Distribution Category:
Mathematics and Computer
Science (UC-405)

ANL-89/41

ANL--89/41

DE90 006382

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

Parallelizing the SDI ACCESS Algorithm for the Connection Machine-2

By

Thomas F. Ewing
David W. Leibfritz

Edited By

Janet L. Rutkowski
Kelly A. Vogel

Computing and Telecommunications Division

December 1989

ACKNOWLEDGMENTS

We are grateful to those who reviewed drafts of *Parallelizing the SDI ACCESS Program for the Connection Machine-2:* Peter J. Bertoncini, James M. Boyle, Clifford M. Caruthers, Thomas R. Canfield, Wayne R. Cowell, Barry S. Finkel, Donald L. Greenwell, Ewing L. Lusk, David M. Malon, Michael Minkoff, Fred M. Moszur, and Charles J. Mueller.

We have prepared the text with University of Waterloo Generalized Markup Language (GML) and Script (Version 86.1) for printing on the Linotype L300P typesetter.

Parallelizing the SDI ACCESS Algorithm for the Connection Machine-2

Thomas F. Ewing
David W. Leibfritz

ABSTRACT

Argonne National Laboratory (ANL) has developed considerable expertise in developing and optimizing algorithms on a collection of multiprocessor computers. One aspect of Argonne research in parallel computing, funded in part by the Command Center/System Operation and Integration Functions (CC/SOIF) Program of the Strategic Defense Initiative Organization (SDIO), involves the speed and other properties of parallel SDI algorithms.

Various algorithms under study have exhibited speedups resulting from parallelization on shared-memory machines. A weapon-target accessibility algorithm called ACCESS exhibited a high degree of inherent parallelism and has been studied on a wide variety of sequential and parallel multiple instruction multiple data (MIMD) machines. To study ACCESS on a massively parallel single instruction multiple data (SIMD) machine architecture, ANL researchers developed a version of ACCESS on a Thinking Machines Corporation 16K processor Connection Machine-2 (CM-2) located at the ACRF.

ANL researchers wrote the Connection Machine version of ACCESS in C*, a version of C by Thinking Machines Corporation with extensions to accommodate SIMD parallelism. Because of the large number of available physical processors and the ability to create virtual processors on the CM-2, the Connection Machine version of ACCESS was able to process an array of 128 x 1024 tasks in parallel. For the data tested, the CM-2 implementation of ACCESS was faster than both the parallel version run on the Alliant FX/8, the Encore Multimax, and the Sequent Balance and the sequential version run on the ANL Cray X-MP/14. For the benchmark ACCESS problem, the CM-2 at ANL with 16K processors achieved a sustained performance of 400 Mflops. On other larger CM-2 machines, the same problem achieved even higher performance: nearly 1600 Mflops on the Los Alamos National Laboratory 64K processor CM-2. The investigation has demonstrated that achieving optimal performance requires structuring the code carefully to keep all available processors busy and to reduce disruptive communication on the front-end processor.

CONTENTS

TABLES

FIGURES

PARALLELIZING THE SDI *ACCESS* PROGRAM FOR THE CONNECTION MACHINE-2

The Strategic Defense Initiative Organization (SDIO) currently faces the formidable task of developing powerful yet cost-effective SDI applications for use with new computer architectures and software, even as this technology continues to evolve at an accelerating pace. Since such rapid technological development makes it unwise to focus on a particular set of computer models or architectures during the early development of SDI systems, SDIO must develop expertise on a variety of existing leading-edge computers, develop efficient algorithms, methods, and programming techniques for these architectures, and acquire experience necessary to select the most appropriate architecture in a specific context for a given class of applications.

The project defined in this report involved taking a prototype weapon-target accessibility algorithm (ACCESS) previously studied on a variety of shared-memory parallel computers and analyzing the implementation of that algorithm on a Connection Machine-2 (CM-2).[1] *The choice of a CM-2 seemed appropriate because of the near-optimal speed gains achieved on shared-memory machines of modest size (up to 24 processors) and the potentially large task of the accessibility problem. The report provides benchmarks for the algorithm on CM-2 configurations with 16K to 64K physical processors, along with comparative timings from other parallel and sequential implementations.*

ACCESS is a program based on a VAX/VMS Fortran accessibility code, developed by Sparta, Inc., to determine the accessibility of ballistic missile threats by interceptors fired from orbiting satellite platforms. Using data describing the initial track of threats, ACCESS propagates the threat trajectories forward in time (by using Kepler's orbital equations) and determines which satellites, if any, can reach each threat with their interceptors and at what time (see Figure 1). ACCESS can then feed this information to a weapon-target assignment algorithm for targeting. Pseudo code for a simplified sequential ACCESS algorithm appears in Table 1.

The initialization and calculation of satellite positions and velocity vectors accounted for as little as 3% of the execution time on sequential machines. We simplified the initialization by reading in satellite and threat data from a disk and by initializing variables such as the mean anomaly, the orbital velocity, the orbital period, the rotational matrix for the satellite constellations, and the tracking fixes on the threats. These initializations are not characteristic of a typical SDI system, where the data comes from SDI tracking and sensor systems. The SDI ACCESS program also calculates satellite positions and velocity vectors for each time step and stores the results for use in the accessibility calculations. Accessibility calculations run faster when these satellite positions and velocity vectors are precalculated (less time is required to look up precalculated values than to calculate them), so the system has more time for refinements, corrections, and targeting when the application is actually used in real time.

[1] *Thomas F. Ewing, "MIMD and SIMD Investigations of an Accessibility Algorithm," presented at the 4th SDI Parallel Computing Group Meeting, Argonne National Laboratory, 10-11 April 1989.*

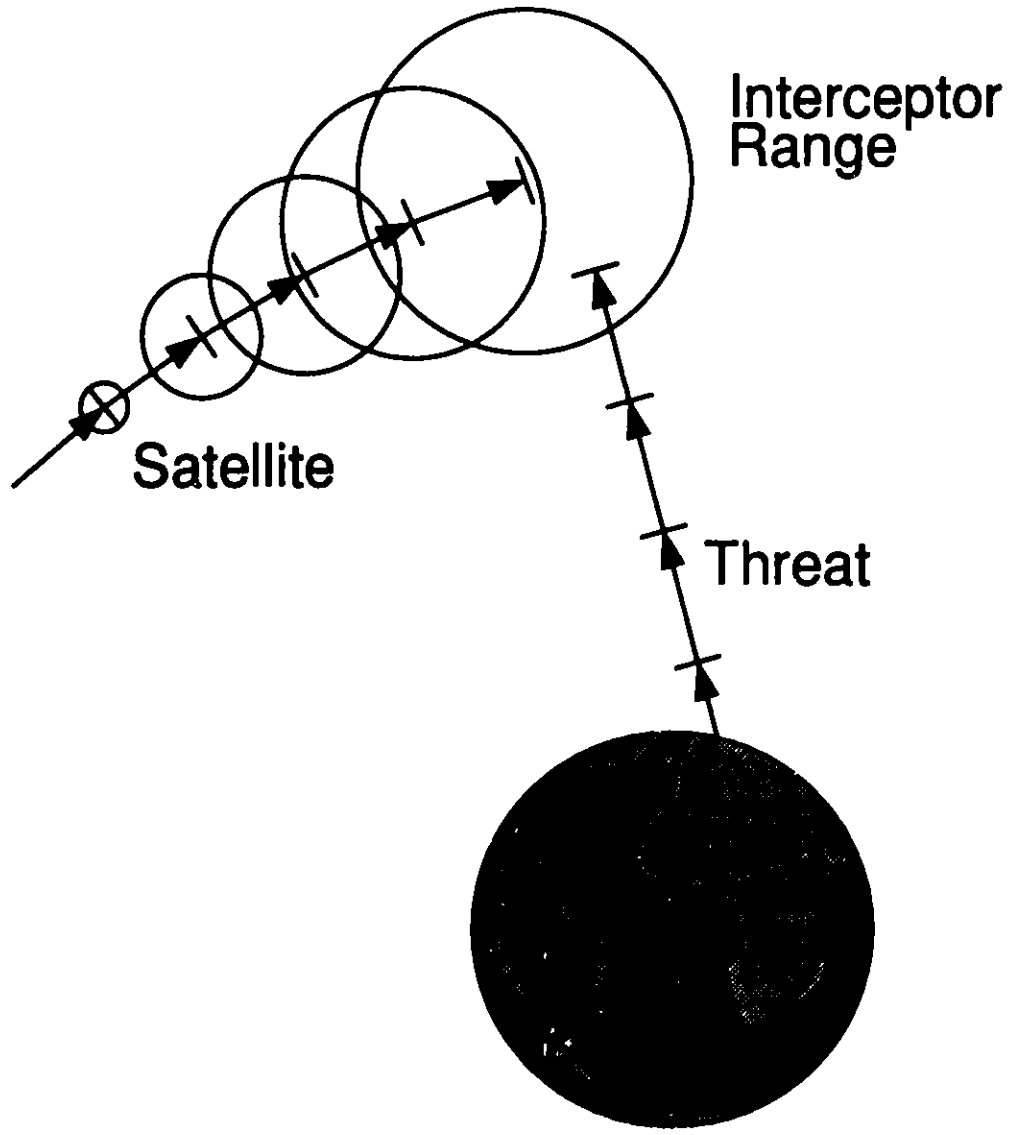

Figure 1: Accessibility of a Threat by Interceptors on Satellite Platforms

Table 1

Pseudo Code for Parallel ACCESS Algorithm

```
{Initialization}

      initialize time intervals, accessibility array, etc.

      read in satellite orbit parameters and initial threat states


{Satellite states calculations}

      do over all time intervals, i

            calculate distance, intercept_dist[i], interceptors travel
            in time interval i

            do over all satellites j

                  calculate satellite states, sat_state[i,j]

            enddo
      enddo


{Accessibility calculations}

      do over all time intervals i

            do over all threats k

                  calculate current state, threat_state, of threat k

                  do over all satellites j

                        calculate distance, sat_threat_dist, between
                        satellite j and threat k

                        if (intercept_dist[i] < sat_threat_dist[i]) and
                        (intercept_dist[i-1] < sat_threat_dist [i-1]) then

                              accessibility[i,j,k] = true

                  enddo
            enddo
      enddo
```

The remaining 97% of time was spent executing accessibility calculations, the focus of our experimentation. For this reason, timing comparisons among MIMD, SIMD, and sequential machines are for the accessibility calculation stage only and do not include the initialization or satellite state calculations.

Accessibility calculations consisted of more than 10^5 independent tasks. We labeled a threat accessible if an interceptor from at least one satellite could reach a threat during both the previous and current time steps. The accessibility of a threat was represented by a three dimensional Boolean cube of time steps, satellites, and threats. A value of 1 (TRUE) at a node denoted access, and a 0 (FALSE) indicated no access. The three dimensional Boolean cube was, therefore, the output of the program.

In the sequential (VAX/VMS Fortran) version of the code, the calculation of accessibility consisted of a triple-nested loop of time steps, threat-state calculations, and satellite-state calculations. The benchmark in this study consisted of 15 time steps, 118 satellites, and 736 threats. For each combination of time steps, threats, and satellites, 1,302,720 (15 x 118 x 736) iterations of the same set of instructions are thus required to determine accessibility. The power of data parallel computing lies in the ability to execute the same set of instructions on multiple sets of data simultaneously, thus reducing the time required to execute all instructions.

The calculation of accessibility is inherently highly parallel because of the independent equations used to calculate the states of each satellite and threat along with the ranges of each interceptor for each time increment. The only dependencies among the equations arise in propagating the satellite and threat state vectors between consecutive time steps. Hence, time should be treated sequentially by the program, whereas the calculation of threat accessibility by satellites is best performed in parallel. Treating time sequentially permits the algorithm to receive and act upon updated sensor data and to pass information to weapon-target assignment modules in a timely fashion. For these reasons, we ran only the inner nested loop of (118 x 736 = 86848) access calculations in parallel for the benchmark case.

Several MIMD machines have already run a parallel version of ACCESS and demonstrated near linear speedup of the accessibility calculation with more than one processor. Among these machines were the Alliant FX/8, the Encore Multimax, and the Sequent Balance 21000. Figure 2 shows the execution speedup curve for the 24 processor Sequent Balance.

Before running the accessibility calculations in parallel on the CM-2, we needed to modify the ACCESS program to fit the unique SIMD CM-2 architecture. Although the version of ACCESS written for the CM-2 is slightly different from the one run on the MIMD machines, we can still fairly compare the timing results. Both the MIMD and SIMD versions of code were optimized for the type of machine, error-checking conditional statements were removed from both parallel versions, and both programs used single-precision floating-point numbers for all floating-point numbers.

PROGRAMMING THE PARALLEL ACCESS PROGRAM ON THE CM-2

The Connection Machine-2 (CM-2) is a massively data-parallel SIMD machine developed and manufactured by Thinking Machines Corporation. In the United States, there are about 24 such machines with a number of processors ranging from 4K to 64K. The CM-2 machine at Argonne National Laboratory has 16K physical processors. At ANL, access to the CM-2 is through a front-end computer, currently a VAX 8250 for Fortran applications and a Sun-4 for C applications. The front-end system provides both a gateway to the CM-2 and to the operating system environment, which performs terminal interaction and file management.

On the CM-2, sequential instructions for the front-end system and parallel instructions for the data processors reside in the memory of the front-end system. The front-end system executes the sequential code and broadcasts instructions to the data processors through a special instruction bus called the front-end bus interface (FEBI). Data may reside in the front-end memory or may be distributed to the local memory attached to each of the data processors. Each data processor has 8K of memory available, and the FEBI allows the front-end system to access the local memories of the data processors a word at a time, for both reading and writing.

Application programs reside physically on the front-end machine of the CM-2 and can be written in C* (a Connection Machine extension of C), *LISP (a Connection Machine extension of LISP), the CM assembly language, PARIS (PARallel Instruction Set, which is callable from C* or Fortran), and Connection Machine Fortran (an implementation of Fortran 8X).

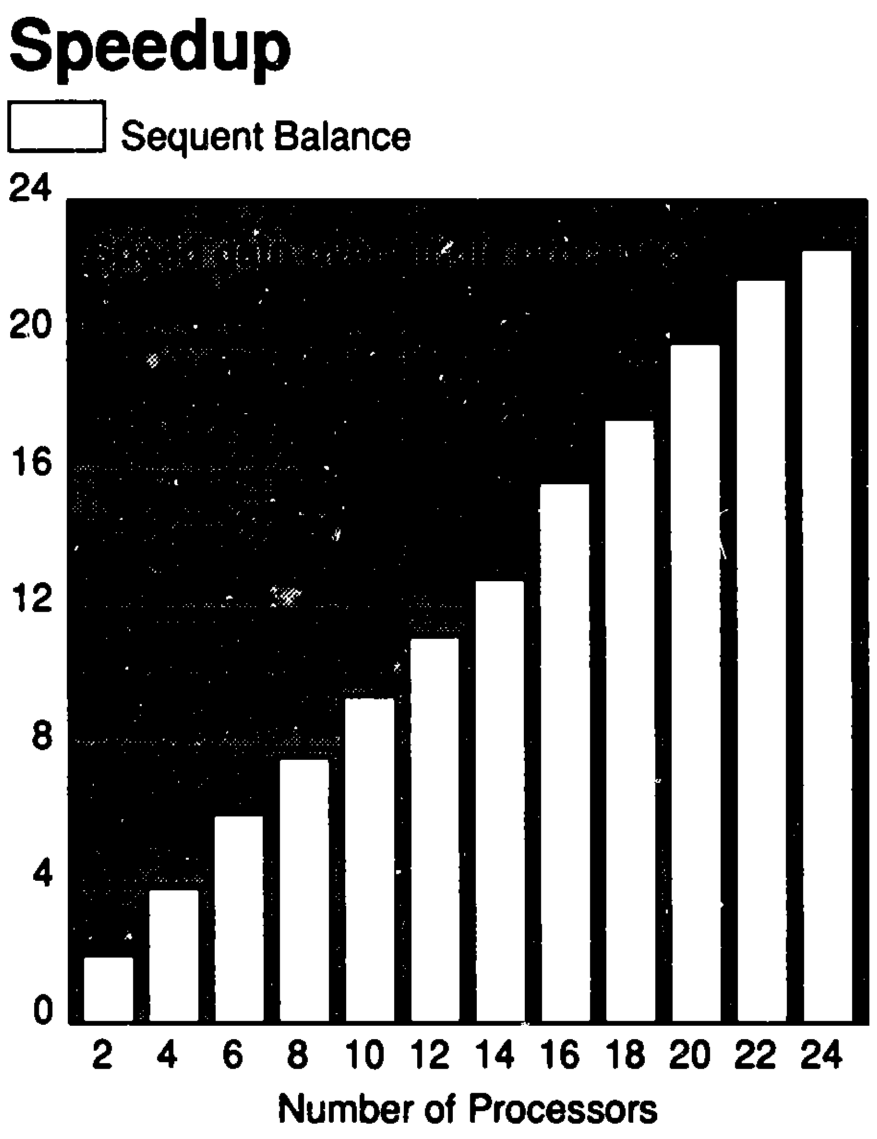

Figure 2: ACCESS Code Speedup on a Sequent Balance

To modify the sequential version of ACCESS to fit the SIMD architecture of the Connection Machine, we translated it into version 5.0.21 of C*. We chose C* because CM-Fortran was still in Beta test form and because a C translation of ACCESS had already been developed. Furthermore, only a few extensions to the C version of ACCESS were necessary to make parallel programming in C* possible. Three language features of C* that make parallel programming possible are:

- A *domain* feature to organize parallel data, similar in syntax and semantics to the C *structure* construct.
- A *poly* type attribute to designate data to be processed in parallel.
- A *selection* statement to activate parallel execution.

The first step in modifying the code for an SIMD machine was to decide how to map the data onto the CM-2 architecture. At ANL, a program written in C* executes on a Sun-4 for the front-end system, where the flow of control is maintained along with all control for interaction with the user. The front-end system gives a program access to the data processors and to data stored in individual processors by treating them logically as memory locations. For the purpose of programming, we could consider the processors to be extensions of the memory addressable from the front-end system.

When designing an algorithm for the CM-2, we needed to partition the problem between the front-end system and the SIMD processors. This partitioning should exploit the strengths of the front-end system and the parallel processors. The front-end system supports the user interface and is superior to the parallel processors in manipulating files and controlling I/O, while the parallel processors are more efficient in processing loops containing many numeric operations. The differences between the front-end system and the parallel processors suggest that portions of the code that are computationally intensive should be run on the processors, while other operations should be run on the front-end system.

If the program size requires more processors than the number of physical processors available, a programmer can divide each physical processor into more than one virtual processor. The system logically divides the memory of each physical data processor into equivalent virtual processors and processes each parallel instruction n times, once for each virtual processor. The execution time of a single physical processor is thereby distributed among the n virtual processors. The overhead involved in using virtual processors reduces the memory capacity of each virtual processor to $\frac{8K}{n}$ bytes of available memory.

Because of the large number of processors on the Connection Machine, it can process a vast amount of data in parallel. The Connection Machine exploits the inherent parallelism of data intensive problems, since each data element of a data structure is associated with a single processor on the CM-2. A data structure can be any combination of objects, such as integers, real numbers, characters, arrays, records, and structures. A programmer must select, combine, and place these data objects into the memory of the data processors. An example of selecting, combining, and placing these data objects would be spreading the *for* loop below over a set of data processors:

```
for(i=0;i <= 1000;++i)
    C[i] = var + B[i];
```

Here a unique element from array B and C would be placed in each of the 1,000 allocated data processors, along with a copy of *var*. Each data processor would then run in parallel by executing the same instruction (without the index i), and the *for* statement would drop out.

After designating which data should be processed in parallel, we decided which data should be stored in each processor. Ideally, one would store all primary data variables referenced by the parallel portion of the code into each data processor to avoid referencing the front-end system. However, limited space in the memory of each processor sometimes makes the local storage of all primary data variables impossible. Data variables stored within each processor reside in an area called the *domain*.

The *domain* resembles a standard C structure, in which each variable is listed along with its data type. Much of the parallel power of the language comes from applying existing C operators and statements to the C* domain data type. The domain for the access code appears in Table 2. Note that the variables within the curly brackets reside on each of the data processors.

All variables declared within the domain are by default of the type *poly*. A poly-type declaration places a local copy of the variable into the memory of each processor. Poly-type variables are also temporarily created when a member function is called. All local variables declared within the member function automatically become type poly during the execution of the function. After the function has executed, however, the poly-type variables declared in the member function cease to exist.

To activate a domain so poly-type variables can be used in parallel, a programmer must use the *selection* statement (see Table 3). The selection statement specifies which domain to activate and lists the instructions to be performed on variables stored within this domain.

Using the domain to distribute array elements to the processors eliminated time-consuming loops that referenced arrays. Although code within the parallel domain retained the same syntax as ordinary sequential C code, the new parallel configuration of data required a restructured version of the ACCESS algorithm to fit the massively parallel architecture of the CM-2. Since the CM-2 is an SIMD machine, we had to coordinate operations on multiple sets of data mapped onto many parallel processors with a single set of instructions. This coordination required an approach different from one used to program on sequential machines. While the instruction set dominated the design approach for the sequential ACCESS algorithm, the data within the domain needed to dominate the design approach for the SIMD algorithm.

On an SIMD machine, all processors simultaneously run the same set of instructions on data stored in their local memories. To operate only on data within selected processors, therefore, a programmer first needs to analyze how a parallel instruction will affect data within each processor and then turn each processor on or off with a conditional. Using this technique, different instructions can execute selectively on different processors.

An example of using the selection statement to assign values to a Boolean parallel variable for a designated set of processors appears in Table 3.

Table 2

Access Code Domain

```
domain sat_threat_data
  {int current_access, previous_access, threat_num, sat_num, on;
  float threat_sat_dist, sat_state_vect[7], threat_state_vect[7],
  object_state_vect[7], reentry_time, time_object_state_vector,
  access[7];}
  [NUM_THREATS] [NUM_SATS];
```

Table 3

Selection Statement Example

```
[domain sat_threat_data].{on = (current_time <= reentry_time);}
```

Here *on* is set to either 0 (False) or 1 (True) in each processor, depending on how the current time interval and the reentry time compare in the member function for the domain, sat_threat_data. For each particular threat, the reentry time is constant. Specifically, the reentry time represents the point in time when a threat reenters the atmosphere and is no longer capable of being intercepted from orbiting satellites. If, for a given processor, the current time interval comes before the reentry time, there is still opportunity to intercept the threat, and the processor remains on. However, if the current time interval comes after the reentry time, the threat is beyond interception, and the processor is turned off. Once a processor is off, it remains off.

When modifying ACCESS to run on the CM-2, we tried to avoid referencing the front-end system, which breaks the flow of execution on the parallel processors, forcing serialization. Since the CM-2 is an SIMD machine, an instruction that references the front-end system will cause each processor to successively access front-end values, while the others must stop and wait. This break in the flow also occurs when executing conditional statements, such as an *if* statement.

To reduce the time required to process data, a programmer should place all the variables used within the parallel portion of code into the local memory of each processor to avoid serialization. A processor accessing a 32-bit floating-point variable from its local memory requires 26.26 microseconds, whereas a processor accessing the same variable from the front-end system requires 588.76 microseconds because of the communication overhead.

A technique we used to avoid conditional statements was to remove the conditional statement and assign a dummy Boolean parallel variable to each processor (as was done in the example above with the Boolean variable *on*). When the processor executed the instruction set following the condition, each value was multiplied by the dummy variable, which had a value of zero or one. The logical result was the same as using the conditional but was faster because the program did not need to reference the front-end system, even though every processor had to execute the instruction set.

The final algorithm differed from classical sequential algorithms but was more natural to the problem. The pseudo code in Table 4 describes the parallel algorithm implemented on the Connection Machine.

The nested loop over satellites and threats fell out naturally because of the data parallelism. We ran the accessibility procedure, which covered every combination of satellites and threats in the domain, only once for each time step.

As previously described, the inner nested loop of accessibility calculations (118 satellites x 736 threats = 86,848) was the focus of the parallelization effort. Note that the timings consistently represent the accessibility portion of the calculations exclusive of the initialization portion. Fortunately, the problem size was nearly ideal for testing the speed of the SIMD ACCESS algorithm on the CM-2. The CM-2 software interface requires processors to be allocated in powers of two. However, in designing a parallel algorithm for this code, we allocated each particular combination of satellite and threat to a single processor. Since there were only 118 satellites and 736 threats in the test problem, we mapped the processors using 128 potential satellites and 1,024 potential threats to attain the power of two. This total of 128 satellites and 1024 threats required 128K (131,072) data processors. Since the CM-2 at Argonne has only 16K physical processors, each physical processor was divided into eight virtual processors. This division resulted in 1K of memory per virtual processor. Less than 75% of this memory was available, however, because of compiler overhead, so the amount of local memory available was actually 750 bytes per virtual data processor. This was enough memory for the parallel portion of the accessibility calculation.

We used the following techniques to create a parallel accessibility procedure for the CM-2. To avoid using the conditional, we created the Boolean poly variable *on* to designate which threat processors should be activated. Almost every variable within the accessibility procedure was of the type *poly* except for two index variables that were logically required to execute on the front-end system. We also removed two procedure calls from the accessibility procedure and replaced them with the procedures themselves to reduce calling overhead. An I/O statement used for debugging was commented out. Two error checking conditional statements were removed from the accessibility procedure (they were never executed for the data tested and had already been removed in MIMD benchmarks). All floating-point numbers were in single-precision rather than double precision to take advantage of the single-precision floating-point hardware and to conserve the limited space of each data processor. We calculated the satellite states during each time step because limited processor space prevented the precalculation and storage of the satellite states for each time step. We replaced a *while* statement with a *for* statement that looped for the maximum number of iterations. To achieve the same logic, a poly Boolean variable was introduced that had the value of the condition within the *while* statement. This poly Boolean variable was then multiplied by each value in the loop to achieve the same result as a conditional statement. Using the above techniques, the entire accessibility procedure ran in parallel on the CM-2.

Table 4

Pseudo Code for CM-2 Implementation of the ACCESS Algorithm

```
  {Initialization}
    initialize time intervals, accessibility array, etc.
    read in satellite orbit parameters and initial threat states
    create array of 1024 threat x 128 satellite (128K total) processors
    distribute copies of control data to each data processor

{Satellite state calculations}
    do over all time intervals i
          calculate distance, intercept_dist[i], interceptors travel in
          time interval i
          do over all satellites j
                calculate satellite states, sat_state [i,j]
          enddo
          call Parallel Access Procedure
    enddo

{Accessibility calculations}
    Parallel Access Procedure for all processes:
          set parallel Boolean variable on for selected threat processors
          calculate current state, threat_state, of threat
          calculate distance, sat_threat_dist,
          between each satellite and threat
          if (intercept_dist[i] < sat_threat_dist[i]) and
          (intercept_dist[i-1]) < (sat-threat-dist[i-1]) and
          (processor is on) then
                accessibility = true
```

MEASURING THE PERFORMANCE OF THE CM-2

Measuring the speed of the CM-2 was complicated task because a program spends time on both the front-end system and the CM-2. We could not simply add the CPU times of both machines together for an accurate measure of total execution time because there is an overlap of time when both the front-end system and the CM-2 execute. For this reason, we used a timing routine that computed elapsed wall clock time. Even though this approach may be slightly conservative, it is the only true measure of real time performance of the CM-2 and assures an upper-bound time measurement.

Another challenge in measuring the speed of the CM-2 involved correcting for the effect of user load on the front-end system. Timings fluctuated greatly with user load. To minimize the effect of user load on timings, we timed the execution of the program when user load was minimal. Then, to filter out the impact of external loads, we ran the accessibility procedure many times in succession and used the lowest time. Keeping the procedure in memory for each successive run also limited memory paging.

Optimizing the parallel access procedure by using the above techniques reduced the CM-2 timing from 8.56 seconds to 4.05 seconds, an improvement of almost 53%. Most of the improvement can be attributed to eliminating the I/O statement used for debugging. The CM-2 does not handle I/O efficiently, because each parallel processor must sequentially access the front-end system to print a value. Removing the I/O statement from the accessibility procedure eliminated 3.82 seconds of time, which accounted for 44.6% of the total loop execution time. For a fair comparison among the machines, we also removed the I/O statement from the MIMD code run on the shared-memory machines and from the sequential code run on the Cray. Conditional statements were also bottlenecks for the CM-2 because these instructions needed to execute at the front-end system and wait for every processor to return before continuing with the next instruction. For the ACCESS code, removing the three error-checking conditional statements resulted in a 2.3% speed gain in performance, and replacing the *while* statement resulted in a 3.4% gain in speed.

COMPARING THE CM-2 WITH OTHER MACHINES

Figure 3 shows how accessibility calculation timings on the Connection Machine compare with other sequential and parallel machines for the sequential and parallel versions of ACCESS. Narrow black bars represent execution times of the sequential ACCESS version run on a single processor; while white bars represent the parallel ACCESS version.

For every MIMD machine, the sequential result was slightly faster (2-8%) than the single processor parallel result, since there was additional interprocessor communication overhead in the parallel version of code. Note that the ANL/Caltech 16K processor Connection Machine benchmark was nearly twice as fast as the best MIMD (Alliant) machine result.

To investigate the relationship between the number of physical processors and execution speed for this problem, we ran the ACCESS code on a 32K Connection Machine located at Syracuse University. With twice the number of physical processors as the 16K Connection Machine, each physical processor on the Syracuse 32K Connection Machine was divided into four instead of eight virtual processors. As we expected, the Syracuse 32K Connection Machine ran the benchmark problem nearly twice as fast as the ANL/Caltech 16K CM-2. Similarly, we also ran the ACCESS code on a 64K Connection Machine located at the Los Alamos National Laboratory. With four times the number of physical processors as the 16K Connection Machine, each physical processor on the Los Alamos 64K Connection Machine was divided into two instead of eight virtual processors. As we expected, the Los Alamos Connection Machine ran the benchmark problem nearly four times as fast as the ANL/Caltech 16K CM-2. The timing results appear in Table 5.

Table 5

CM-2 Timings (118 Satellites X 736 Threats)

ANL/Caltech 16K CM-2	*4.05s*
Syracuse 32K CM-2	*2.05s*
Los Alamos 64K CM-2	*1.07s*

These timings demonstrated an almost ideal two-fold gain in speed on the 32K Syracuse CM-2 and a four-fold gain in speed on the 64K Los Alamos CM-2.

To compare timings between the Cray X-MP/14 and the CM-2 running an optimal size problem, we increased the dimension of the problem to occupy all the virtual processor space available on the Connection Machine (128 satellites and 1024 threats) by adding more satellites and threats. With the optimal Connection Machine problem size, the ANL/Caltech 16K CM-2 ran the accessibility calculation almost 36% faster than the Cray X-MP/14. The Cray X-MP/14 timings for this case appear in Table 6.

Table 6

16K CM-2 and Cray X-MP/14 Timings (128 Satellites X 1024 Threats)

ANL/Caltech 16K CM-2	*4.10s*
ANL/Cray X-MP/14	*6.39s*

The CM-2 accessibility calculation time changed only slightly when the problem size changed from the benchmark case depicted in Figure 3 to an optimal problem size, since, in both cases, the entire array of satellite x threat tasks were performed in parallel. Both the benchmark and the optimal size problem must use a number of processors that is a power of two and run the equivalent of the full-size (128 x 1024) problem.

By extracting the PARIS code, we could count the number of floating-point operations on a single processor. To accommodate for the SIMD parallelism, we multiplied this number by the number of virtual processors (128K). This procedure included counting floating-point operations on processors with pseudo data to implement the 736 by 118 satellite case. The ANL/Caltech 16K CM-2 ran the parallel access procedure of the benchmark problem using approximately 103.68 million floating operations per time step, resulting in a speed of 409 Mflops. The Syracuse 32K CM-2 ran the parallel access procedure at a speed of 809 Mflops, almost double the speed of the procedure run on the 16K CM-2, and the Los Alamos 64K CM-2 ran the parallel access procedure at 1550 Mflops, almost four times the speed of the procedure run on the 16K CM-2.

After optimizing the problem size, we tested the maximum potential of the ACCESS code by increasing the number of instructions executed within a given time interval and iterating through them as needed to achieve the desired number of instructions. Increasing the floating point instruction count within the accessibility procedure had almost no effect on the performance of the code, a fact which demonstrated that our code was running close to its maximum potential speed.

To test how the number of processors affected execution time on the ANL/Caltech 16K CM-2, we modified the problem size to match the desired number of processors by changing the number of satellites and compensating for lost satellites by performing additional iterations of the accessibility procedure (e.g., for an 8K run, we used 8 satellites and iterated over the accessibility procedure 16 times on each processor, and for a 16K run, we used 16 satellites and iterated over the accessibility 8 times on each processor). In essence, each processor performed the work of more than one satellite sequentially. The results of this test appear in Figure 4. Note that the "vp" in parentheses in Figure 4 represents the virtual processors as opposed to the physical processors. As the number of processors approached the maximum number of available physical processors, the speed increased in linear fashion. As noted from the Syracuse CM-2 result and the Los Alamos CM-2 result, the linear behavior of Mflops versus the number of physical processors continues to hold reasonably well to a 32K and 64K processor CM-2 (see Figure 5).

In Figure 4 , the curve representing the number of processors versus execution speed is linear until the number of processors used by the program equals the number of physical processors. Past this point, the rate of performance gain decreases. Despite the inverse relationship between performance gain and number of virtual processors, the execution speed continued to increase with the use of more virtual processors. In fact, an additional 20% improvement in Mflops was possible by using virtual processors when the problem size exceeded the number of physical processors. This demonstrated the pipelining effect of using virtual processors to increase the efficiency of the physical processors. However, specifying a number of virtual processors greater than the number required by the problem size adversely affected performance, because of wasted processors.

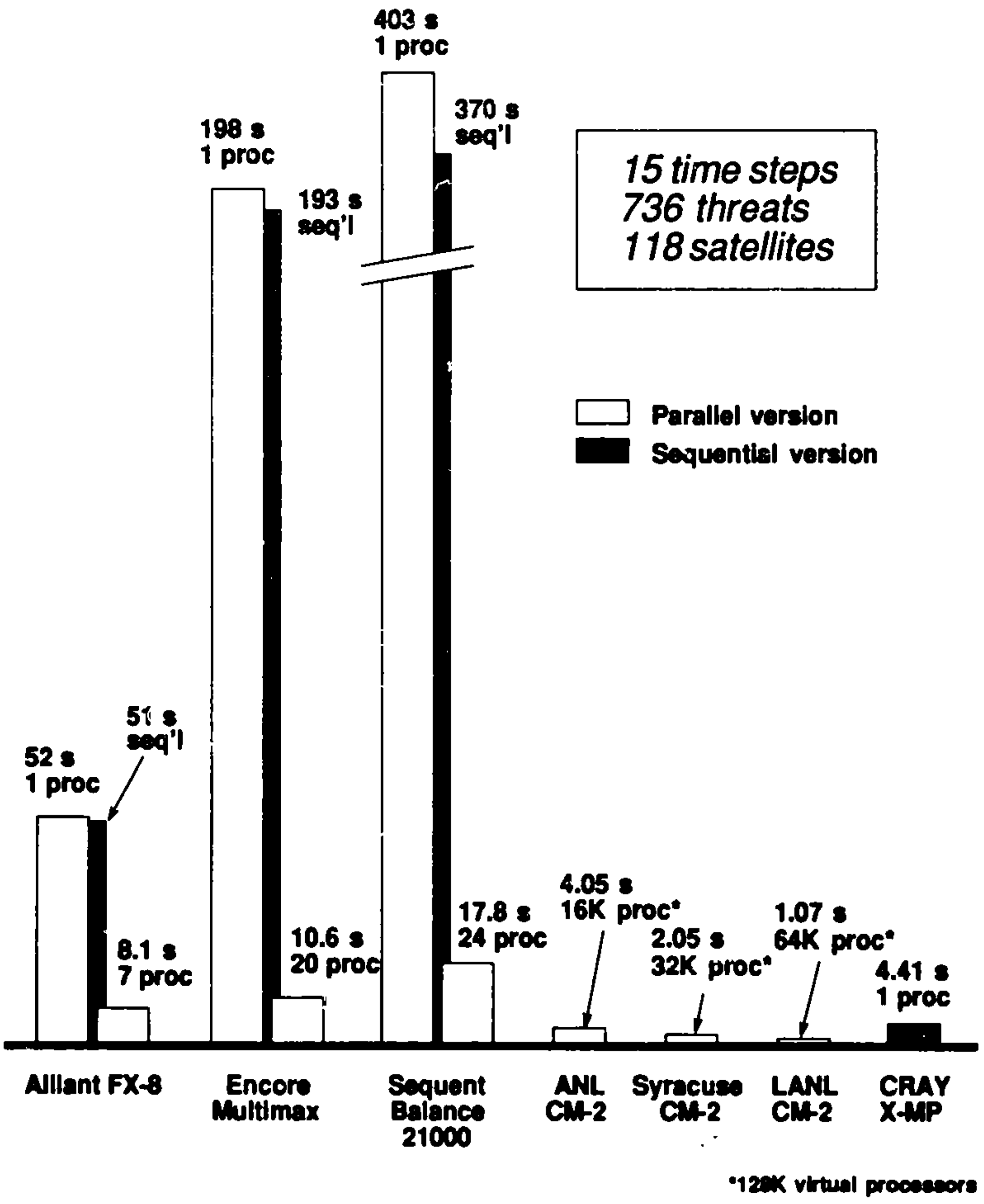

Figure 3: Comparison of ACCESS Execution Times on Different Machines

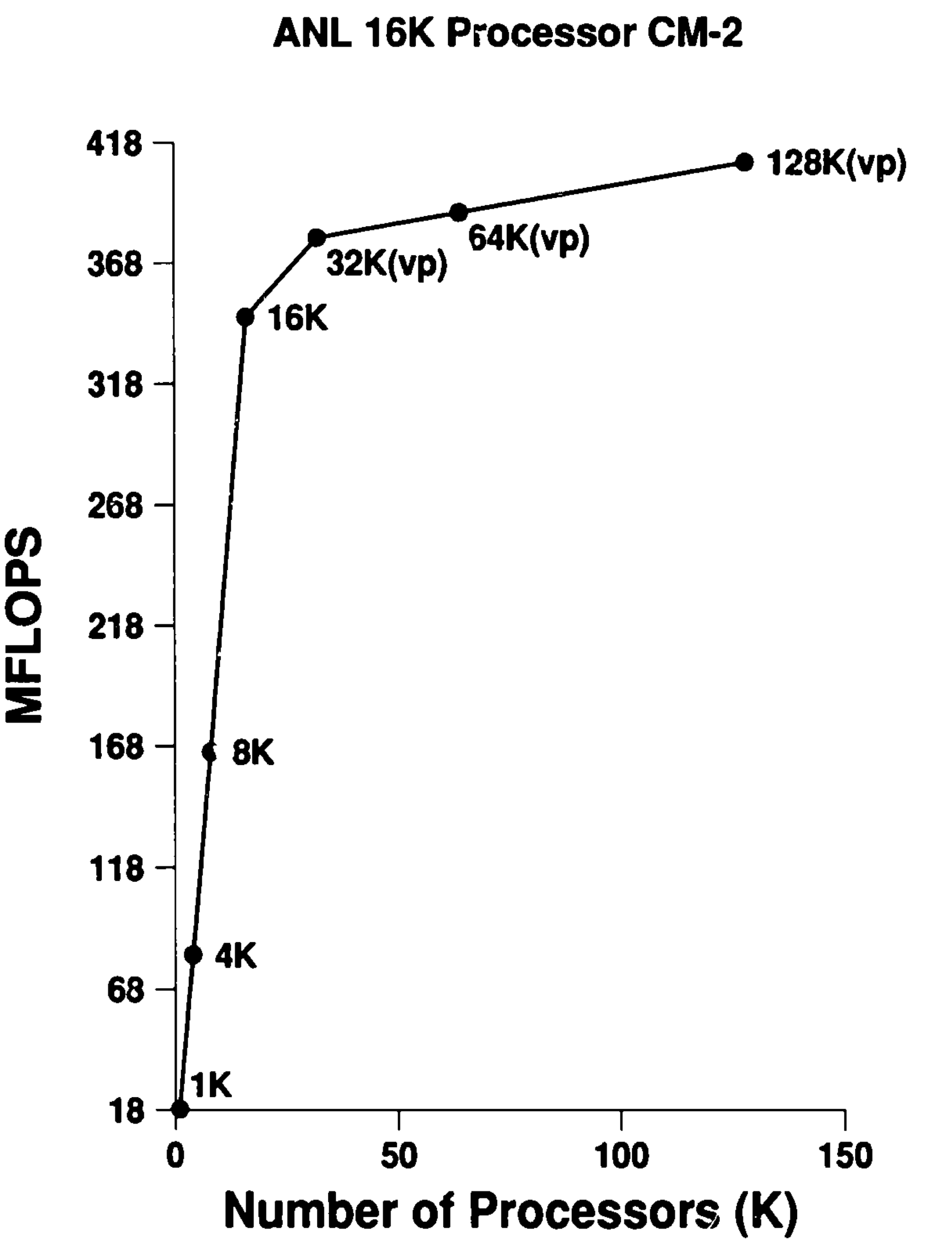

Figure 4: Connection Machine Speed (MFLOPS) Versus Number of Processors

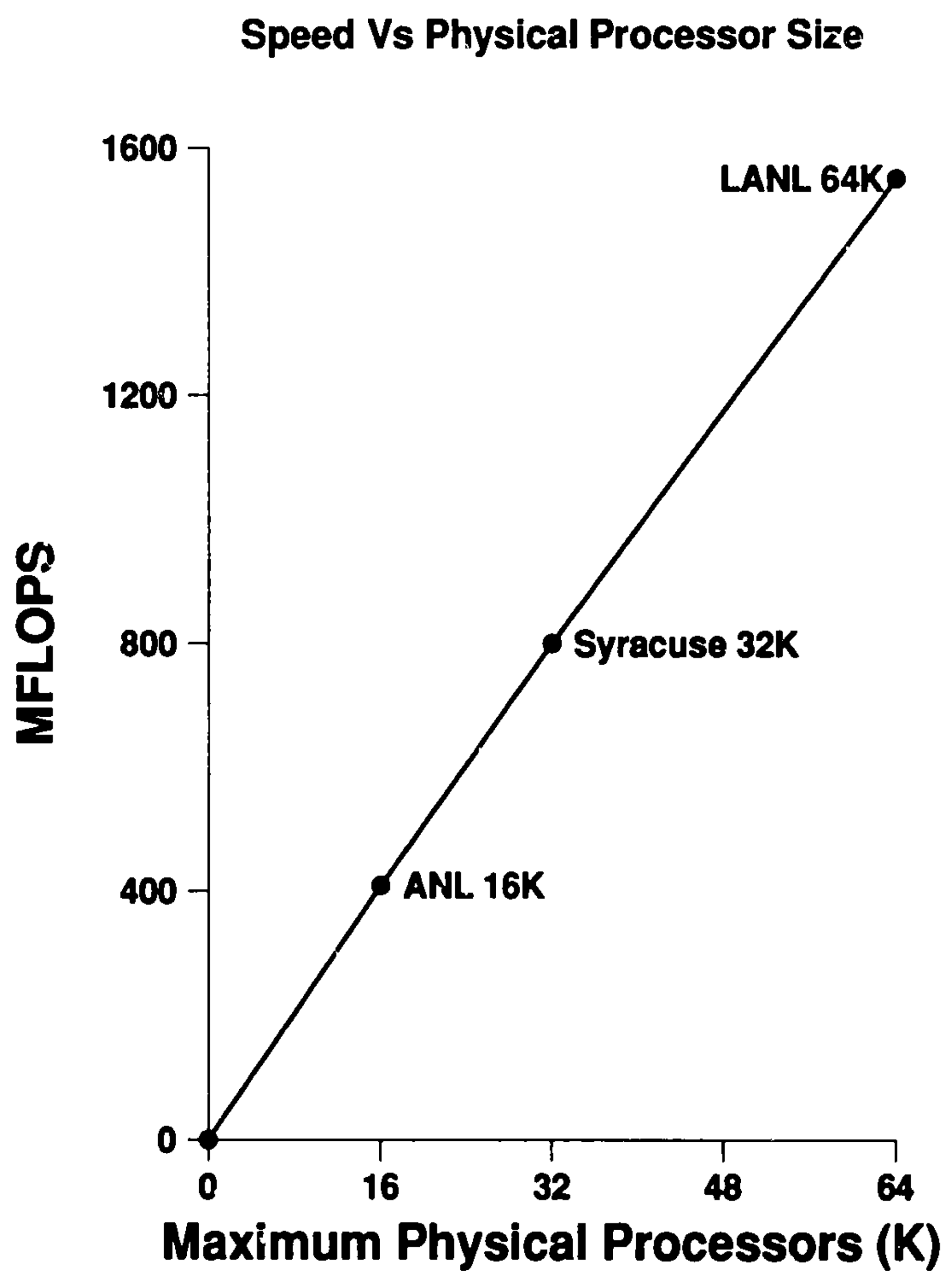

Figure 5: Connection Machine Speed (MFLOPS) Versus Maximum Physical Processor Size

CONCLUSIONS

By parallelizing the ACCESS program for a massively parallel SIMD Connection Machine, we have demonstrated substantial speedups over several parallel MIMD machines and the sequential ANL Cray X-MP/14 supercomputer. Even though the versions of ACCESS used for each machine differed slightly, we believe a fair comparison of timings was achieved. Because of the lack of dependencies in this problem, and the vast amounts of data, the CM-2 was able to sustain the best performance. The natural style of programming in C* easily allowed the programmer to store various data types on the parallel processors and to select when these processors should be activated. As these bit processors performed operations simultaneously on the parallel domain, the speedup increased linearly as the number of physical processors increased. The only degradation in performance occurred when the processors were forced into serialization in referencing the front-end system. Through experimentation we have shown that we can replace these statements with logically equivalent parallel instructions.

The virtual processor feature of the CM-2 proved to be particularly useful for a problem of this nature, where the precise size of the satellite-threat array is not known in advance and may, in fact, exceed the number of physical processors. The use of virtual processors is more efficient for large problems that exceed the physical processor space than for performing the computations sequentially N times, where N is the ratio of the number of satellite-threat pairs to the number of processors, times in code. Moreover, no coding changes are necessary to use virtual processors.

The ANL/Caltech 16K processor CM-2 (with a virtual processor size of 128K--the processor size that most nearly fitted the problem size for this machine) achieved a sustained performance of over 400 Mflops. For the same problem size, the Syracuse 32K processor CM-2 and the Los Alamos 64K processor CM-2 achieved sustained performances of over 800 Mflops and nearly 1,600 Mflops, respectively. The Los Alamos CM-2 executed more than four times as fast as the ANL Cray X-MP/14.

We should not expect all software components of an SDI system to achieve gains in speed (through parallelization) comparable to those demonstrated by the accessibility problem. Indeed, data dependencies would increase communication overhead among parallel processor nodes and might favor a machine architecture with fewer, faster processors for other SDI functions, such as the tracker-correlator. Further comparative studies between machine architectures for various SDI software components will thus be necessary. It is likely that optimal performance will be achieved from an SDI system composed of coupled multiple-machine architectures, each tailored to specific functions.

BIBLIOGRAPHY

"C* User's Guide." In *Using the Connection Machine® System* (ANL/MCS-TM-118, Revision 1). Mathematics and Computer Science Argonne Division: Argonne National Laboratory, May 1989.

"C* Reference Manual." In *Using the Connection Machine® System* (ANL/MCS-TM-118, Revision 1). Mathematics and Computer Science Argonne Division: Argonne National Laboratory, May 1989.

"Model CM-2 Technical Summary." In *Using the Connection Machine® System* (ANL/MCS-TM-118, Revision 1). Mathematics and Computer Science Argonne Division: Argonne National Laboratory, May 1989.

Ewing, Thomas F. "MIMD and SIMD Investigations of an Accessibility Algorithm." Presented at the 4th SDI Parallel Computing Group Meeting, Argonne National Laboratory, 10-11 April 1989.