COMPUTER SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY · STANFORD, CA 94305

An Investigation of the Partitioning of Algorithms Across an MIMD Computing System

(IMAP-1)

Erik J. Gilbert

TECHNICAL NOTE NO. 176

13 May 1980

COMPUTER SYSTEMS LABORATORY
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

This work was performed In conjunction with the software development effort at Stanford University, under a s&contract from Lawrence Livermore Laboratory to the Computer Science Department, Principal Investigator Professor Gio Wiederhold, Contract Number LLL PO#9083403. The S-l hardware and software development has been supported by the Department of the Navy via Office of Naval Research Order Numbers N00014-76-F-0023, N00014-77-F-0023, and N00014-78-F-0023 to the University of California Lawrence Livermore Laboratory (which Is operated for the U. S. Department of Energy under Contract No. W-7405-Eng-48), from the Computations Group of the Stanford Linear Accelerator Center (supported by the U. S. Department of Energy under Contract No. EY-76-C-03-0515). and from the Stanford Artificial Intelligence Laboratory (which receives support from the Defense Advanced Research Projects Agency and the National Science Foundation).

Table of Contents

1 Introduction 1

2 Definitions 2

3 Motivation 3

4 Techniques for partitioning 4

5 Selection of a sample application 6

6 Overview of SIMPLE 8

7 Partitioning SIMPLE 12

8 Multiprocessor SIMPLE simulation 20

9 Analytic speedup computation . . . . , 22

10 Synchronization and communication 27

11 Directions for future study 30

12 Conclusion 31

13 Acknowledgments 32

14 References 33

Abstract:

Multiprocessors offer a new dimension of computing power to user applications with extremely compute-intensive requirements. One mode of using a multiprocessor to this **end is that of** expressing the algorithmic solution to an application problem in some partitioned fashion in order to make effective use of several processors at once. This report documents aspects of progress made to date in the continuing investigation within the S-l project of application partitioning across classic MIMD multiprocessors. The goal of this investigation is to demonstrate the practicality of the partitioned application mode of multiprocessor use for large classes of realistic problems, particularly in the context of a large-scale multiprocessor such as the S-l project has designed and will be implementing. The investigation so far has included a broad spectrum of studies, ranging from general research on multiprocessing issues to specific experiments with algorithms for . particular application problems.

Key words and phrases:

Multiprocessor, application partitioning, large-scale computing, parallel processing, tightly coupled MIMD systems.

1 Introduction

Multiprocessors (strictly speaking, Multiple-Instruction-Multiple-Data processor systems [4]) are potentially extremely attractive systems for realizing greatly enhanced computing capabilities. Potential benefits include significant improvements over both the Single-Instruction-Single-Data and Single-Instruction-Multiple-Data types of uniprocessor systems in the areas of availability, configurability, cost-effectiveness, and raw computing power. The primary concern of this paper is in the area of raw computing power enhancement available from a multiprocessor. Particular reference is made to a classic multiprocessor architecture being explored by the S-i Project [12,131

In order to best realize the computing power increase potentially available from a multiprocessor on a single application problem, it must be possible to express the algorithmic solution to the problem in some partitioned fashion in order to make effective use of several processors at once. The simplest and most obvious, but still useful, scheme for partitioning is to run several different, independent copies of the application algorithm on different sets of data which are of interest to the researcher; such an approach is predicated on the different data sets being totally independent of each other.

However, the more interesting case occurs when the algorithm is structured to take advantage of parallelism inherent in the problem when processing a single set of data. Such an approach admits of possibly very large gains in effective processing speed, and thus potentially allows many more cases of interest to be studied in *serial* order per unit of wallclock time; such an approach is required if subsequent data sets have features determined from computational study of previous ones. It is this particularly useful case to which the present investigation is addressed.

This report documents aspects of progress made to date in the continuing investigation within the S-l project of application partitioning across classic MIMD multiprocessors. The goal of this investigation is to demonstrate the practicality of the partitioned application mode of multiprocessor use for large classes of realistic problems, particularly in the context of a large-scale multiprocessor such as the S-l project has designed and will be implementing. The investigation so far has included a broad spectrum of studies, ranging from general research on multiprocessing issues to specific experiments with algorithms for particular application problems.

This report *covers* several different topics, roughly following the chronological development of the investigation to date. After some definitions and further motivation for application partitioning, there is a brief discussion of generally applicable techniques for partitioning. Next is a historical perspective of the process of selecting a "representative" application for further detailed study. An overview is then given of the algorithm chosen for specific study, followed by a description of the methods used for partitioning that algorithm. After that appears a discussion of some simulation results, followed by some analytic results. Finally, there is a discussion of some of the detailed implications of this study in terms of synchronization and communication mechanisms found to be desirable for support of application algorithm partitioning. The report concludes with a discussion of directions which such investigations may profitably take in the future.

2 Definitions

The term "multiprocessor" will be used in this paper to refer to a generalization of the structure of the S-l multiprocessor. A few important attributes of this generalization are listed here. It is assumed that there is a moderate number (say 2 to ZOO) of extremely fast single processors tightly coupled to a relatively large amount (at least 10 million words) of uniformly accessible global memory. Each processor may also have a moderate quantity of very high performance memory (e.g. cache) local to it, but it must also have high bandwidth (although not necessarily short latency) access to the global memory. Many of the ideas contained herein apply also to other multiprocessor structures (e.g. larger numbers of slower processors), but the S-l structure has been the primary focus for optimization of the partitioning approach developed in this study.

"Problem partitioning" refers to the process of taking a particular application problem and constructing an algorithmic solution for it which can take advantage of the potential for parallel execution available in a multiprocessor. The primary motivation assumed for partitioning a . problem is to substantially decrease the absolute wallclock time taken to run each instance of the application (as opposed to other motivations such as improved reliability and/or recoverability). For partitioning to be realistically useful in this way, the partitioned application must run substantially faster than a uniprocessor version, even when all possible overheads are taken into account, including operating system, multiprocess communication, and synchronization.

The "speedup" of a multiprocessor algorithm is the ratio of wallclock elapsed time for uniprocessor execution to wallclock elapsed time for multiprocessor execution. It is, of course, a function of the number of processors, and possibly other algorithm parameters. The speedup provides a measure of the success with which the problem has been partitioned, indicating greater success as the speedup approaches the number of processors. There are actually conditions in which the speedup can theoretically exceed the number of processors; these will be noted in more detail later.

3 Motivation

Depending on the exact nature of the application, the process of constructing an effectively partitioned solution can vary greatly in difficulty. As mentioned earlier, any uniprocessor code can be immediately run on a multiprocessor in the mode of multiple independent data files; but this is not a partitioned single application as defined here. This mode does serve to characterize a class of applications whose partitioning is trivial. Any application which consists of several already independent computations can be easily partitioned in this way. A simple example (in which each of the independent computations has the same structure) might be a Pascal compiler which has the ability to process multiple input procedures in a "separate compilation" mode.

There is another class of applications which is almost as easy to partition. It is all those which have a basic iterative "outer loop" with perhaps a summary data gathering step at the end of each iteration, but with several otherwise independent computation blocks occurring in each iteration. Examples of this structure of computation may be found in Monte Carlo approaches to simulation [6].

To approach the issue of difficulty of partitioning from another standpoint, it is reasonable to ask for what kinds of applications is a substantial amount of partitioning effort justified. In particular, if an application is hard to partition it could be argued that it is better to run it unpartitioned in timesharing mode along with other user problems in order to still gain the cost-effectiveness benefits of the multiprocessor. However, there are several interesting application areas in which any gains in absolute wallclock execution time are valuable. Classic examples include the weather prediction problem and many types of real-time processing, such as radar signal processing. Also, as the number of processors in the multiprocessor increases, the attractiveness of the partitioning approach increases for more and more problems.

4 Techniques for partitioning

As the number of designed or implemented multiprocessors increases, a few general techniques for problem partitioning are beginning to emerge [7]. Three such techniques which have been considered could be called “synchronous partitioning,” "asynchronous partitioning,” and “pipelining.” From the descriptions below it should become apparent that these techniques are by no means mutually exclusive, and hence may be used in combination in a partitioned application.

The technique of synchronous partitioning is perhaps the most obvious and most widely applicable of the three. In this technique, either the data structure or the program (or both) is divided up into comparatively independent units, and multiple processes compute in parallel within these units. Occasionally, two or more processes must synchronize with each other in order to maintain data consistency or pass summarizing information among processes.

The technique of asynchronous partitioning [1] is less intuitive and can lead to debugging difficulties due to the lack of exact reproducibility of results, but offers advantages by avoiding the potentially large overheads of frequent process synchronization. This technique is best understood in the context of iterative numerical algorithms. For instance, consider an application containing a large two dimensional matrix of real numbers which are being updated by an iterative algorithm such that each new point value depends in some simple way on previous values of neighboring points. The points may be partitioned into groups among the available processors. If the correctness of the algorithm does not depend on the use of a precisely defined previous iteration value for neighboring points in the updating procedure, and if instead any reasonably recent value will suffice for convergence, then the processes may iterate without synchronization at each iteration. The termination test for convergence is most easily implemented if the error measure is defined so that it can be tested locally in each process, determining process convergence independent of other processes. Thus the only form of synchronization is implicit in the shared point values, which are continually updated in parallel. Note that a pure implementation of this technique has the characteristic that no process is ever in synchronization wait, and so all processes are always actively working towards the solution. However, it is possible for convergence to be slower than in a synchronized solution due to nonuniform use of previous values. The general ideas of this technique have been the subject of research for several years, often appearing under the name “chaotic relaxation” [2].

The pipelining technique is very similar to the pipelined approach in high-performance uniprocessor hardware implementation. In this technique, the computation is divided into several parts, called “stages,” which have the characteristic that the output from one stage becomes the input to the next stage. So, once the computation is well under way, all of the stages can be computing in parallel with the data streaming into the first stage and the results streaming out of the last stage. An example of this approach might be the division of a compiler into scanner, parser, global optimization, and code generation stages.

One note about the interaction between implementations and multiprocessor efficiency and speedup deserves mention here. Some problem partitionings, especially those using pipelining, lead to an implementation which has a fixed maximum speedup, e.g. the number of pipeline stages.

Other partitionings which are parameterized by the number of processors (and possibly some measure of data size) have no obvious fixed maximum speedup, and thus (at least for large data sizes) can continue to benefit from additional processors. Thus, the implementor should be aware that, by requiring a fixed length pipeline or division into a fixed number of parallel processes, a limit on future flexibility for expansion is being imposed.

5 Selection of a sample application

Since the main goal of this investigation is to demonstrate the practicality of partitioned execution of real-world problems, the study includes considering several application areas and specific codes as possible candidates for partitioning. The majority of applications thus far considered may be characterized as large scale numerical physical simulations. These applications have been emphasized because of the potential for substantial benefits to these large consumers of CPU time and because of the large body of expertise in this area available to the S-l project at the Lawrence Livermore Laboratory.

A number of large LLL physics codes were considered as candidates for a detailed partitioning study. Criteria for selection of these codes included such issues as importance to the LLL community and tendency to be representative of currently used techniques for physical simulation. Importance at LLL is often measured by total CPU hours consumed in production use; one of the candidate codes surveyed below consumed over 6000 hours of CDC 7600 CPU time during the past year. The majority of detailed study to date has been concentrated on one particular code, named SIMPLE [3]. Other candidate codes are abstracted here for completeness and as de facto targets of future detailed study.

ZOHAR [9] is an LLL "particle pusher" code for two dimensional electromagnetic plasma simulation. It can also be run in a "two-and-a-half dimensional" configuration, where it maintains two space dimensions and three velocity components. It is expected to be quite amenable to multiprocessor partitioning due to its basic Monte Carlo approach. It will also be an interesting vehicle for studying the utility of S-l special hardware and microcode for Fast Fourier Transforms, since it makes substantial use of two dimensional FFT's; the short S-l floating point data format should also be highly useful for Monte Carlo needs, and makes very high arithmetic bandwidth available.

TARTNP [10] is an LLL Monte Carlo code for calculating the transport of neutrons, photons, and neutron-induced photons. It is fairly large and complex due to the flexibility of the code. For example, the user may divide a problem into a large number of different zones and then specify any one of 17 different methods of calculating particle transport for each zone. The current code version is about 25,000 lines of the LLL dialect of Fortran. Despite its complexity, it is expected to be amenable to partitioning both due to its Monte Carlo natu're and the tendency for users to specify from 25 to 200 completely independent samples in a given run.

WAVE [5] is a Los Alamos Scientific Laboratory code for "particle in cell" (PIC) electromagnetic and electrostatic simulation, supporting both two dimensional and two-and-a-half dimensional configurations. It is also attractive to consider for partitioning, due to its Monte Carlo nature, and it may be somewhat simpler than the related LLL code (ZOHAR). Portions of the code which may be particularly interesting for analysis of S-l implementation include the FFT and tridiagonal linear equation solver sections of its Poisson solver.

LASNEX [14] is an LLL code for studying inertial confinement fusion. It includes a two dimensional Lagrangian formulation of shock hydrodynamics, and models frequency-dependent radiation transport and electron and ion thermal conduction. Of **particular** interest is its "Incomplete Cholesky Conjugate Gradient" (ICCG) subroutine [8], which iteratively solves the large sparse linear systems which arise from the heat conduction difference equations. A substantial fraction of LASNEX CPU time is spent in the ICCG subroutine. LASNEX has many structural and algorithmic similarities to the SIMPLE code.

CORONET is an LLL two dimensional Lagrangian hydrodynamics code. It contains some implicit PDE solution techniques, as well as some explicit ones. Like the LASNEX code, it uses global timestep control and a table-lookup technique for supplying the required equations of state for the fluids being modelled, and it supports irregular boundaries. It does not employ Monte Carlo techniques in its most-used form.

Two other LLL codes have been nominated by LLL scientists as possible candidates for multiprocessor partitioning study. One is MEG, a one dimensional hydrodynamics solver containing some Monte Carlo sections. The other is HEMP, a pure two dimensional hydrodynamics code using explicit solution techniques, which supports irregular boundaries. Both of these seemed to pose less challenge for present purposes than the other major nominees.

As stated above, detailed study so far has been primarily concerned with the SIMPLE code. SIMPLE was chosen for several reasons: (1) it is a close relative of CORONET and LASNEX, two large and complex LLL physics codes; one of the few basic algorithmic differences is SIMPLE's lack of support of regions with irregular boundaries; (2) it seems to be representative of techniques used in many physical modelling codes, in that it contains both explicit and implicit PDE solvers, it uses a two dimensional Lagrangian formulation, and it uses table lookup for the required equations of state of the fluids being modelled; (3) it is sufficiently simplified from a full-scale code to be quite manageable in size (as it consists of about 1800 lines of Fortran); (4) it has been studied by others **in** the academic sector as a candidate application for a number of novel processor architectures, such as data-flow machines.

Large scale numerical simulations such as these form **one significant class of applications for** which multiprocessor partitioning seems to be appropriate. Several other application areas have been suggested and studied by other researchers. One application considered because it is widely used but still fairly self-contained is sorting. Internal (main memory) sorting is fairly CPU intensive but still difficult to partition effectively, since obvious partitionings are often theoretically limited to less than linear speedup [11]. Another general area of application is heuristic search of large tree structures such as those found in artificial intelligence problems. One other application which has been studied in this light is set partitioning integer programming [11].

6 Overview of SIMPLE

The intent of the SIMPLE code is to give a simple, yet realistic, example of computational fluid dynamics and heat flow. It solves the differential equations of inviscid compressible shock hydrodynamics and simple heat conduction using a Lagrangian formulation. It works in two dimensions on a region with a regular boundary. It uses simple table lookup to represent the equations of state of an ideal gas.

The differential equations are reduced to difference equations. The equations for hydrodynamics and for heat conduction are solved in separate sections of the ‘code employing different techniques. The hydrodynamics equations are solved explicitly, while the heat conduction equations are solved implicitly.

The basic data structure in SIMPLE is used in the representation of the mesh covering the problem domain. This consists of 13 two dimensional arrays of real numbers to store the physical quantities involved, plus a few additional arrays for working storage. There are also one dimensional arrays to store the tabular definition of the equation of state, and of course several scalars to store miscellaneous other quantities.

The outer loop structure (after the problem is set up) is a simple iteration as the time value is increased:

```
repeat
    hydrodynamics pass;
    heat conduction pass;
    compute new delta t;
    advance time by delta t;
until done
```

The hydrodynamics pass has the following structure:

```
for each mesh zone, calculate new pressure using EOS lookup;
for each boundary zone, calculate geometry;
for each boundary zone, set up boundary physics;
for each mesh point, calculate new velocities;
for each mesh point, calculate new coordinates;
for each mesh zone, calculate new density and change in specific volume;
for the boundary, sum up the work done on the boundary by hydrodynamics;
for each mesh zone, calculate artificial viscosity and Courant delta t limit;
for each mesh zone, calculate hydrodynamic work and update energy, using EOS;
for all zones, sum up the kinetic energy for the entire problem;
for each mesh zone, calculate new temperature via table lookup;
```

The heat conduction pass has the following structure:

```
for each mesh zone, calculate two material properties;
for the boundary, set the boundary properties to neighboring values;
for each mesh zone, calculate coupling constants in the K direction;
for each mesh zone, calculate coupling constants in the L direction;
for the boundary, set some appropriate initial values;
over the entire mesh, perform a forward and backward sweep in L (see text);
over the entire mesh, perform a forward and backward sweep in K (see text);
for each mesh zone, calculate new energy using EOS, and new delta t limit;
for the boundary, sum up the energy flow across boundaries;
for all zones, sum up a new internal energy for the entire problem;
```

Notice that, with one significant exception, all of the steps in both passes have a very similar structure. A typical step passes over the entire mesh (or maybe just the boundary) making local computations at each mesh zone or mesh point. These local computations typically involve updating one or more quantities at the given place in the mesh, after examining the previous value and perhaps the previous values of a few neighboring elements. Also, of course, computations involving only the boundary contribute much less to the CPU time used than computations over the whole mesh. Below in figure 1 is a pictorial representation of a typical SIMPLE mesh processing step, showing the obvious left to right and top to bottom ordering of mesh element computation. This will be compared in the next section with the multiprocessor partitioned ordering.

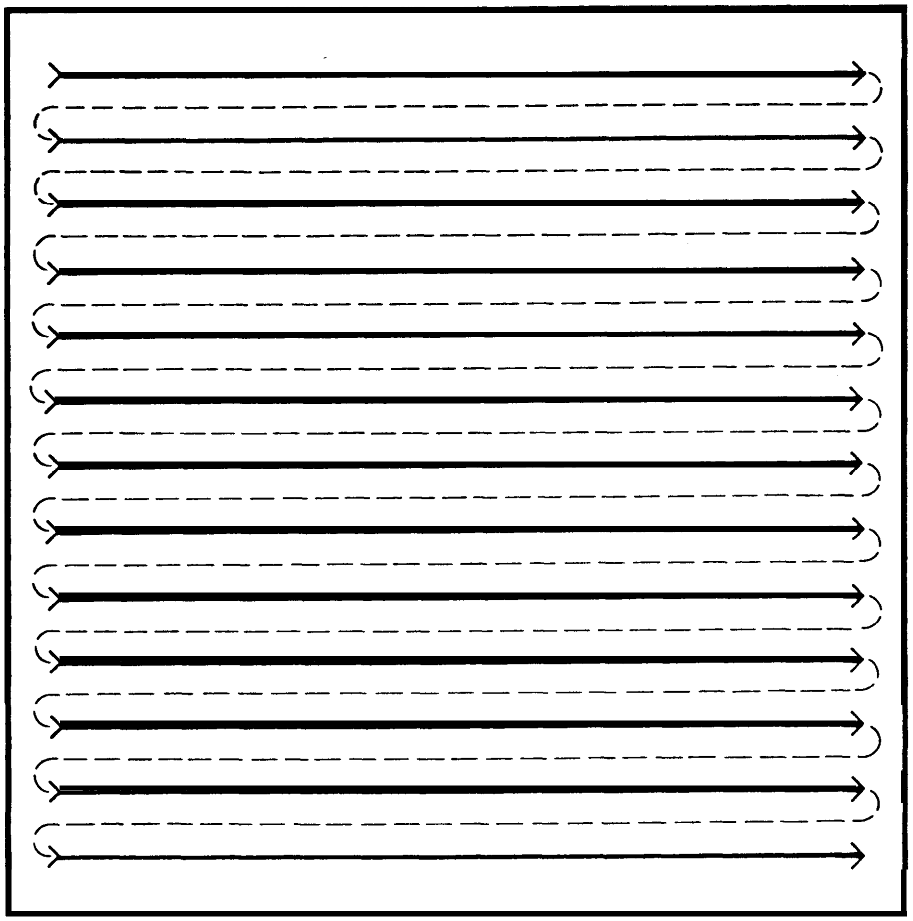

Figure 1: Typical mesh processing order

The one exception to this structure occurs in the steps in the heat conduction pass called "forward and backward sweeps." Superficially, even these steps may appear to have a similar structure. There is one important difference, arising from the implicit nature of the PDE solution technique used. In order to solve a tridiagonal linear system of equations, the sweeps evaluate a recurrence of the form X[I] := A[I]*X[I-1] + B[I] for increasing values of I. The key here is that each new X quantity depends on the new X quantity which was computed in the immediately preceding inner loop iteration. This dependence causes some difficulty in the partitioning of the sweeps, which will be discussed in the next section on partitioning of SIMPLE.

Another algorithmic structure which is used is the table lookup in the EOS and temperature calculations. In both cases this consists basically of locating between which pair of entries in an increasing table of values some physical quantity belongs numerically, and then using the corresponding index into other tables to compute an interpolated function value. The lookup search is a straightforward sequential ordered table search. The only unusual part of the algorithm is that each table index is saved as a starting place for the next search, which reduces the search time assuming that successive uses of the function tend to pass arguments of similar magnitude.

7 Partitioning SIMPLE

Given the basic structure of SIMPLE as mostly performing localized operations fairly uniformly across a large data structure (the mesh), the most reasonable approach seems to be a data-directed synchronous partitioning. Specifically, each of several processes is assigned to operate on some subset of the mesh, computing independently of the other processes whenever possible. Occasional synchronization is required for keeping one mesh section from advancing too far beyond the others, for mesh-wide data summarizing operations, and in the sweep steps (as explained later).

An important factor to consider in partitioning a program which has a large shared data structure like SIMPLE's mesh is the presence of per-processor cache memory on the S-l. Due to the large difference in access time to a word in central shared memory and a word already in a processor's cache, it seems reasonable to select a programming style which has a high degree of per-processor data locality of reference. In a code like SIMPLE, where the computation within the large shared data structure is quite evenly distributed, an easy way to do this is to statically partition the data structure into fixed equal size pieces, with one piece per process. Each process is then responsible for updating its piece, and most of the references to that piece are made by that process, thus assuring locality. Notice that it is also being assumed that there is at least an approximate one-to-one mapping between processes and processors, and that processes do not migrate from one processor to another very often. Otherwise, the advantages of having all recently referenced data in cache would be lost. These assumptions are valid *on* the bare hardware of the S-l, and must be supported by any operating system which is intended to maximally benefit from this type of operation.

For SIMPLE, the chosen static mesh partitioning is into "column groups." Each process is assigned a different fixed subrange of columns of all of the arrays representing the various physical quantities in the mesh. Of course, any process can still access any quantity at any point in the mesh since the entire mesh is in global shared memory. It is just assumed that most of the references within a column group will be by the assigned process, and hence that the column group data will reside largely in the corresponding processor's cache. Below in figure 2 the column grouping of the mesh is shown, along with the ordering within processes of a typical mesh computation, corresponding to the uniprocessor version in figure 1.

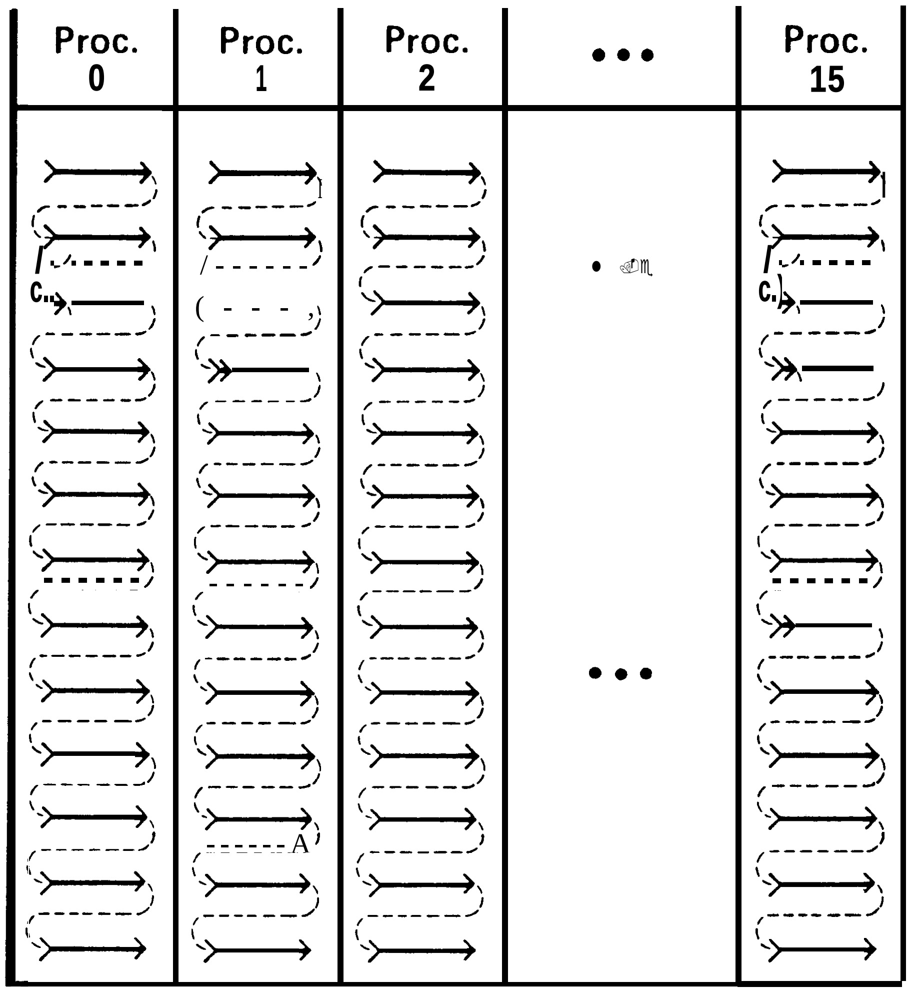

Figure 2: Partitioned mesh processing order (independent mesh computations)

The presence of the caches has another interesting performance implication, on the theoretical speedup achievable for a program like SIMPLE. For some reasonable mesh sizes, it is quite possible that all of the mesh data will not fit in a single processor's cache, but that it will all fit in all of the caches combined. In this case, the uniprocessor execution of the program could continually cause cache misses and corresponding lengthy delays while cache lines are transferred to and from main memory. However, the multiprocessor version, with properly partitioned references to the data, would be able to retain the entire mesh distributed in all of the caches, thus causing cache missing to be insignificant. In this way, if the efficiency of processor utilization is high enough, the speedup over the uniprocessor version could actually exceed the number of processors executing the program!

Here are some details of the process of partitioning a mesh-processing part of SIMPLE other than the difficult sweep steps. One simplified typical code fragment might appear as follows (where K is the row index and L is the column index):

```
for L := LMN to LMX do
    for K := KMN to KMX do
        begin
        A[K,L] := (X[K,L]+Y[K,L])*(Z[K,L-1]-Z[K-1,L]);
        B[K,L] := B[K,L] + Z[K,L]*Y[K-1,L+1];
        end;
for L := LMN to LMX do
    for K := KMN to KMX do
        begin
        P[K,L] := P[K,L] + A[K-1,L]*A[K,L];
        Q[K,L] := Q[K,L] + B[K,L-1]*B[K,L];
        end;
```

Notice that the computation at each mesh point is in terms of other quantities at the same mesh point or at a neighboring mesh point, thus maintaining the desirable locality mentioned above. The only references outside of local column groups occur when L is in the first or last column of a group and an off-column reference like Z[K,L-1] or Y[K-1,L+1] is made. Also notice that within each loop pair the computations at each mesh point are completely independent of each other, and so they may be performed in parallel with no interprocess synchronization needed. However, the second loop pair is dependent on the results of the first loop pair, so synchronization is needed to insure that the second loop pair is not executed by some process before the A and B values needed have been stored by perhaps a different process. An easy way to insure this is to insert a “synchall” synchronization call between the loop pairs. This call forces each process to wait at that point of execution until all processes have arrived there, and then they are all allowed to proceed. Since each process is performing essentially the same amount of work on its column group as any other process, all processes may be expected to complete the first loop pair at about the same time and *not* cause very much overhead wait time at the synchall point.

So, the partitioned version of the code fragment might appear as follows (where PR is the index of the process executing the code, and LMN and LMX have been expanded into arrays specifying the column boundaries of the column groups):

```
for L := LMN[PR] to LMX[PR] do
    for K := KMN to KMX do
        begin
        A[K,L] := (X[K,L]+Y[K,L])*(Z[K,L-1]-Z[K-1,L]);
        B[K,L] := B[K,L] + Z[K,L]*Y[K-1,L+1];
        end;
SYNCHALL;
for L := LMN[PR] to LMX[PR] do
    for K := KMN to KMX do
        begin
        P[K,L] := P[K,L] + A[K-1,L]*A[K,L];
        Q[K,L] := Q[K,L] + B[K,L-1]*B[K,L];
        end;
```

Another code fragment worth considering is one which includes a summary data gathering of some sort, such as the result of a summation or a maximum over some function of the mesh points. Such a computation requires a complete pass over the mesh with a single scalar output, rather than updated mesh values. A typical step of this sort in SIMPLE might appear as follows:

```
TOTAL := 0.0;
for L := LMN to LMX do
    for K := KMN to KMX do
        begin
        TOTAL := TOTAL + A[K,L]*X[K,L];
        end;
```

The obvious approach to partitioning this code fragment is to let each process compute a total for its column group, and then to have one process compute a grand total at the end. If the number of processes is sufficiently large, the simple grand total computation should perhaps be replaced by a multiprocess version which could compute pairwise subtotals, eventually reducing the number of totals to one grand total. So, a partitioned version of this code fragment could appear as follows (where MAXPROC is the number of processes):

```
PTOTAL[PR]:= 0.0;
for L := LMN[PR] to LMX[PR] do
    for K := KMN to KMX do
        begin
        PTOTAL[PR] := PTOTAL[PR] + A[K,L]*X[K,L];
        end;
SYNCHALL;
if PR = 1 then   (* processor I computes the grand total *)
    begin
    TOTAL := 0.0;
    for P := 1 to MAXPROC do
        TOTAL := TOTAL + PTOTAL[P];
    end;
```

One more code segment which should be discussed is the table lookup in the EOS and temperature calculations. As mentioned previously, these code segments are essentially straightforward sequential ordered table searches, which can be executed independently by several processes in parallel with no synchronization since they are computing function values from read-only data. The only exception to this is the mechanism for retaining the search index from the previous search for use as a starting point the next time. The obvious way of partitioning this mechanism is to retain the previous search index on a per process basis, so that processes executing in unrelated portions of the mesh do not try to use each other's previous search indices.

Finally, some consideration must be given to the somewhat more difficult problem of partitioning the forward and backward sweeps in the heat conduction pass. It was noted previously that the difficulty arises from the recurrence inherent in the loops, in which each inner loop iteration is dependent on results computed in the previous iteration. Even this structure would not be difficult to partition if such iterations only traveled up and down columns, and hence were evaluating each recurrence only within a single process. Unfortunately, recurrence iterations are performed both up and down columns and across rows. So, some of the recurrences must be evaluated across process boundaries, requiring some form of synchronization at very frequent intervals (once per process boundary crossed, i.e. several times per row of the mesh in a single sweep). All previously discussed partitionings of SIMPLE required only about one synchronization per computation over the entire mesh.

A partitioned forward mesh sweep recurrence is diagrammed below in figure 3. In the figure, the mesh rows have been grouped into blocks of three rows each; row blocking is not used in the code below, but it will be discussed later in the section on analytic study. The vertically circled column group boundaries show points at which synchronization must occur. The diagonally circled column group portions represent a single time snapshot of how much computation can proceed in parallel, due to the skew enforced by the left to right recurrence. As time proceeds, more and more processes become actively executing in parallel. The average degree of parallelism depends on the “angle of attack” of the diagonal part, which is determined by the amount of row blocking, the mesh dimensions, the number of processors, and the synchronization overhead. These quantities will be studied in detail later, in the analytic section.

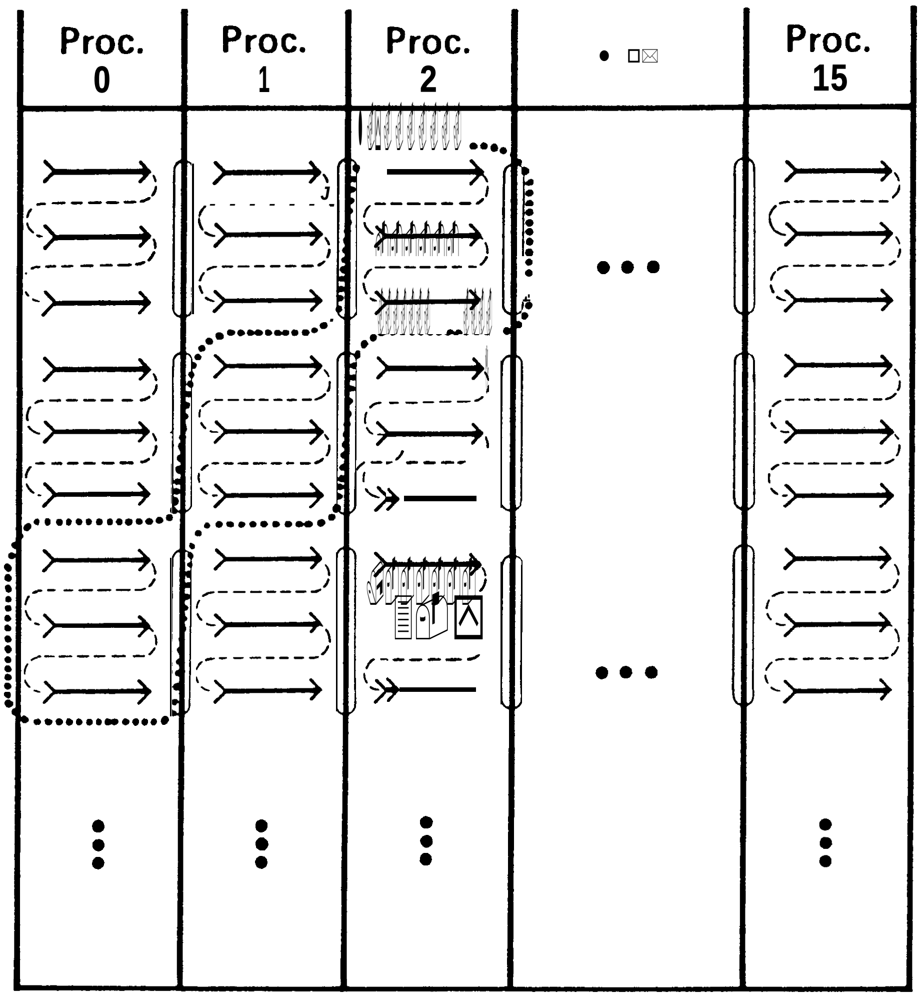

Figure 3: Partitioned forward sweep processing (with row blocking)

For reference, a slightly simplified version of the unpartitioned troublesome sweep code fragment appears below:

```
for K := KMN to KMX do
    begin
    for L := LMN to LMX do
        begin
        A[K,L] := Q[K,L] / A[K,L-1];
        B[K,L] := (Q[K,L-1]*B[K,L-1]) / A[K,L-1];
        end;
    for L := LMX downto LMN do (* note stepping by -1 *)
```

```
        begin
        T[K,L] := A [K,L]*T[K,L+1]+B[K,L];
        end;
    end (*for Kt);
```

For partitioning this code fragment, there must be a somewhat more detailed synchronization mechanism than the synchall call used previously. Let A WAIT(n) and SIGNAL(n) correspond roughly to Dijkstra-style semaphore operations P(SEM[n]) and V(SEM[n]). So, AWAIT(n) will be used to await a signal on channel n, and SIGNAL(n) sends a signal on channel n. Notice that the signal channels contain counters, so more than one signal may be outstanding on a channel. In this example, AWAIT(n) will be used to wait for a signal from process n that it is finished with the next row's worth of column group. Given these definitions, the partitioning discussed above might be expressed in this code fragment as follows:

```
for K := KMN to KMX do
    begin
    if PR > 1 then AWAIT(PR-1);
    for L := LMN[PR] to LMX[PR] do
        begin
        A[K,L] := Q[K,L] / A[K,L-1];
        B[K,L] := (Q[K,L-1]*B[K,L-1]) / A[K,L-1];
        end;
    if PR < MAXPROC then SIGNAL(PR);
    end (*for K*);
for K := KMN to KMX do
    begin
    if PR c MAXPROC then AWAIT(PR+1);
    for L := LMX[PR] downto LMN[PR] do (* note stepping by -1 *)
        begin
        T[K,L] := A[K,L]*T[K,L+1]+B[K,L];
        end;
    if PR > 1 then SIGNAL(PR);
    end (*for K*);
```

There exists an alternative to the above frequently synchronizing structure for partitioning the sweeps. It would be possible to transpose the mesh quantities needed, perform the sweeps in the "easy" direction (up and down columns}, and then transpose back. This unwieldy sounding approach could actually be quite feasible in practice when compared to the high overhead method outlined above, if the problem of efficiently transposing a matrix on the multiprocessor can be solved. At the moment this problem appears to be quite complicated, since it must attempt to keep all of the processors busy at the same time as avoiding delays from simultaneous access to any single central memory unit. Some further analysis of how much time the high overhead method spends in waiting will be presented below in the section on analytic speedup computation. Also, a new hardware-supported mechanism will be proposed in the section on synchronization and communication which should eliminate most of the overhead associated with loops like this one, thus obviating the need for such a transpose mechanism.

The above examples tend to blur the distinction between variables which are shared by all processes and variables which are private to each process. In any actual implementation, of course, this distinction must be explicitly specified by the user to the system software. For SIMPLE, shared variables include the mesh quantities, the EOS lookup tables, and miscellaneous globally known scalars. Private variables include loop indices and temporaries.

8 Multiprocessor SIMPLE simulation

As part of this study, a modest simulation of the SIMPLE code running on a multiprocessor has been implemented. One of the major goals of this simulation was just to force the process of considering the entire code line by line, to be sure there were no major conceptual problems in partitioning it for a multiprocessor. Another goal was to study in general the effectiveness of the previously discussed approaches to partitioning, with particular emphasis on the viability of a static mesh partitioning. The simulation is accurate in the sense that it still actually solves exactly the same problem as that solved by the uniprocessor code, but it is incomplete in its consideration of the complexities of the multiprocessor environment.

The basic approach of the simulation was to begin with the code of SIMPLE (translated into Pascal from Fortran), and to start by considering how to partition each stage for multiprocessor execution. However, each code segment which was intended to run independently in different processes is actually enclosed in a loop which executes the code segment successively for each process, varying the process number over all possible values. Variables which were private to each process (and had a useful lifetime long enough to justify keeping the values across major processing steps) were changed into arrays indexed by the process number.

To this structure *were* added timing, synchronization, and statistics gathering functions. The main timing function is assignment of CPU time spent in mesh computation to the simulated process which is spending that time. This is done by surrounding each code segment with calls to start and stop charging of CPU time to a specified process. The only synchronization function simulated at present is the synchall function described earlier. It is simulated by a procedure call to update timing statistics at each synchall point. The most interesting statistic is of course the speedup achieved. It is computed by assuming that wallclock time advances at the same rate as the maximum CPU time used by any process at each stage. Again, this assumes essentially that each process has its own dedicated processor. Other statistics gathered include per-process CPU usages, which may be examined to determine how successfully the workload is being balanced among the processes.

The results of sample runs of the simulation were quite encouraging. The per-process CPU usage was very well balanced, indicating that the static mesh partitioning appears to be a reasonable choice. The speedup reported for a small mesh on a 16 processor system varies between 9.7 and 14.5, depending on how it is chosen to account for CPU time which was spent but not attributed by the simulation to any particular process. Both the accuracy of the simulation and the speedup value are expected to increase as the size of the mesh increases.

There are a number of ways in which the simulation to date is incomplete, and so future improvements could increase the accuracy of the simulation. One minor improvement would be to accurately model the subtotal accumulation part of each summary data gathering step; at present these parts are assumed to be negligible and are not included. Also lacking is a detailed study of exactly which synchall points are absolutely necessary; at the moment they are scattered liberally throughout the code wherever there is any possibility that global resynchronization might be needed. The influence of the caches was included in some analytical study (discussed later), but the simulation assumes uniform access to all of shared memory. The critical points in SIMPLE where cache misses will happen due to column group boundary crossing have been isolated but not yet included in the simulation. Probably the most important omission in the present simulation is accurate accounting for the complicated interactions in the heat conduction forward and backward sweeps. At present the simulation assumes that a no cost mesh transpose is done; this is obviously unrealistic.

9 Analytic speedup computation

The simulation studies of SIMPLE to date have ignored the implications on memory access times imposed by the per-processor caches of the S-l. The presence of the caches is quite important to consider due to the possibility of a more than tenfold increase in access time for a word not in cache over the access time for a word in the proper cache. In particular, accessing a word in cache takes only about 50 nanoseconds, whereas accessing a word from the cache of another processor will probably take about 300 nanoseconds (averaged assuming all words of a cache line will be accessed, corresponding to a cache line access time of 4 to 5 microseconds).

To augment the simulation results, some analytic study has been done of potential speedup of portions of the SIMPLE code, allowing for the presence of the caches. The portions chosen for analytic study are the sweep steps in the heat conduction pass and a time-consuming nested loop representative of the hydrodynamics pass. The sweep analysis is simplified by only considering the overhead implied by cache misses and cache line transfers, and not considering any overhead associated with process synchronization. The next section of this report proposes a mechanism which can reduce both types of overhead.

The sweep analysis will be presented for the forward sweep only. The forward sweep part of the slightly simplified code fragment which appeared earlier is repeated below for reference:

```
for K := KMN to KMX do
    begin
    for L := LMN to LMX do
        begin
        A[K,L] := Q[K,L] / A[K,L-1];
        B[K,L] := (Q[K,L-1]*B[K,L-1]) / A[K,L-1];
        end;
    end (*for K*);
```

It is assumed that the two dimensional arrays are stored by columns, i.e. that element A[1,1] is followed in memory by element A[2,1]. Thus each S-l 16 word cache line contains 16 elements of a column of an array. Since cache transfers happen in units of 16 word lines rather than single words, it is reasonable to assume that the overhead would be less if each process computes the above recurrence on a block of rows within its column group before letting the next process start on those rows, rather than synchronizing on each single *row*. This blocked approach allows more than one word to be used from each cache line each time it is transferred across from one processor to another at a column group boundary. For simplicity, the unit of time used here will be the length of time it takes one processor to execute a single loop iteration with no cache misses.

Define the following parameters:

B = blocksize = number of rows in a block
W = time to compute the recurrence over one block of one column group
P = number of processors

R = number of rows (KMX-KMN+l)
C = number of columns (LMX-LMN+l)
TSP = elapsed time for entire forward sweep on single processor
TMP = elapsed time for entire forward sweep on multiprocessor

The speedup for this section of code is then defined by:

$$\text{Speedup} = \frac{TSP}{TMP}$$

By the definition of the unit of time,

$$TSP = R \cdot C$$

Similarly, notice that since a block is B rows high and C/P columns wide, W would be equal to B•C/P in the absence of cache misses.

To formulate the value of TMP, the exact sequence of the multiprocessing sweep execution must be observed. Each of the P processors computes the recurrence at each element in all R rows in its assigned column group of C/P columns. In other words, each processor computes over R/B blocks, taking time W•R/B for the whole computation. If all the processors could execute for the whole sweep fully in parallel, W•R/B would also be the elapsed time of the entire computation. However, no processor can begin its computation until the previous processor has finished computing on its first block. So, processor P must wait for P-l block computations until it can start on its first block. From then on all processors can run in parallel, assuming that each block computation takes the same amount of time. Thus, accounting for the delayed startup of processor P, the total elapsed time is:

$$TMP = W \cdot (P\text{-}1 + \frac{R}{B})$$

In formulating W, processor to processor cache line moves must be accounted for, in addition to the basic iteration compute time. The basic iteration time (of the real code in SIMPLE) is estimated at about one microsecond, and a cache line move takes 4-5 microseconds, so it seems a reasonable estimate that a cache line move takes about the same time as 4 basic iterations. Assuming that a previous step computed values for the array Q, causing its data to reside in the caches of assigned column group processors, the read-only use of Q[K,L-1] in each iteration causes each processor to participate in two cache line moves (one from the previous processor and one to the next processor) every 16 rows. So, the contribution to each block computation of accessing Q[K,L-1] is twice B/16 times the cache line move time, i.e. 2•B/16•4.

Each iteration also uses the values of A[K,L-1] and B[K,L-1], but *not* in a read-only fashion, i.e. each value used was written on a recent earlier iteration. So, the cache lines containing these values at column group borders must be moved between processors (twice) for every block which is processed, not just every 16 rows. In other words, when processor p finishes computation on a block, the cache line containing the last column of that block must be moved to processor p+1, and if the next **block to be computed by processor p also contains any part of that cache line it must be moved** back to processor p. So, where $\lceil x \rceil$ (“ceiling of x”) is the smallest integer greater than or equal to x, the contribution to each block computation of accessing both A[K,L-1] and B[K,L-1] is twice $2 \cdot \lceil B/16 \rceil$ times the cache line move time, i.e. $2 \cdot 2 \cdot \lceil B/16 \rceil \cdot 4$.

Therefore, the final formula derived for W is:

$$\begin{aligned} W &= \frac{C \cdot B}{P} + 2 \cdot \frac{B}{16} \cdot 4 + 2 \cdot 2 \cdot \lceil \frac{B}{16} \rceil \cdot 4 \\ &= \frac{C \cdot B}{P} + \frac{B}{2} + 16 \cdot \lceil \frac{B}{16} \rceil \end{aligned}$$

From all of the above, the speedup can be expressed:

$$\begin{aligned} \text{Speedup} &= \frac{R \cdot C}{\left(\frac{C \cdot B}{P} + \frac{B}{2} + 16 \cdot \lceil \frac{B}{16} \rceil\right) \cdot \left(P\text{-}1 + \frac{R}{B}\right)} \\ &= \frac{P}{\left(1 + \frac{P}{2 \cdot C} + \frac{16}{B} \cdot \lceil \frac{B}{16} \rceil \cdot \frac{P}{C}\right) \cdot \left(1 + \frac{B \cdot (P\text{-}1)}{R}\right)} \end{aligned}$$

Notice that this formula has the expected quality that as the number of rows and columns in the mesh approaches infinity, the speedup approaches the number of **processors.**

For determining some numeric values of the speedup formula, some interesting parameter values can be substituted. Specifically, by letting $P=16$, choosing sample values for R and C, and then maximizing over B, the following speedups are obtained:

R	C	speedup max	occurs at B =
128	128	7.0	4
128	1024	11.4	2
1024	1024	14.1	4

Now, a time-consuming nested loop representative of the hydrodynamics pass will be analyzed. The loop chosen performs the function listed earlier in the SIMPLE overview as “for each mesh point, calculate new velocities.” This loop forms the majority of a subroutine **which** uses 397. of the CPU time used **in** the hydrodynamics pass, and 26% of the total CPU time in SIMPLE. It is also in the class of easily partitioned loops in SIMPLE, since it requires no potentially costly synchronization calls within the loop body. Thus, the major factor which might limit speedup for this section is the overhead of cache misses due to shared array access. For reference, the exact text of the loop in question appears below:

```
for L := LMN to LMX do
    for K := KMN to KMX do
        begin
        AU := (P[K,L]+Q[K,L]) * (Z[K,L-1]-Z[K-1,L]) +
              (P[K+1,L]+Q[K+1,L])*(Z[K+1,L]-Z[K,L-1]) +
          (P[K,L+1]+Q[K,L+1])*(Z[K-1,L]-Z[K,L+1]) +
          (P[K+1,L+1]+Q[K+1,L+1])*(Z[K,L+1]-Z[K+1,L]);
   AW := (P[K,L]+Q[K,L]) * (R[K,L-1]-R[K-1,L]) +
          (P[K+1,L]+Q[K+1,L]) * (R[K+1,L]-R[K,L-1]) +
          (P[K,L+1]+Q[K,L+1]) * (R[K-1,L]-R[K,L+1]) +
          (P[K+1,L+1]+Q[K+1,L+1]) * (R[K,L+1]-R[K+1,L]);
   AUW := RHO[K,L]*AJ[K,L]+RHO[K+1,L]*AJ[K+1,L]
            +RHO[K,L+1]*AJ[K,L+1]+RHO[K+1,L+1]*AJ[K+1,L+1];
   AUW := 2.0/AUW;
   AU    := -AU*AUW;
   AW    := AW*AUW;
   U[K,L] := U[K,L]+DTN*AU;
   V[K,L] := V[K,L]+DTN*AW;
   if ABS(U[K,L]) <= VCUT then U[K,L] := 0.0;
   if ABS(V[K,L]) <= VCUT then V[K,L] := 0.0;
   end (*for L,K*);
```

Define the following parameters:

P = number of processors
R = number of rows (KMX-KMN+1)
C = number of columns (LMX-LMN+1)
K = number of cross-cache references within one row of a column group
S = time for a single inner loop iteration with no cache misses
V = time to move one word from one cache to another
T = total time spent on all iterations on single processor

First, observe that:

$$T = R \cdot C \cdot S$$

Now, there are K cross-cache references within one *row* of a column group. There are R rows and P column groups. Each cross-cache move takes time V. So, the total time spent moving cache words on the multiprocessor is $K \cdot V \cdot P \cdot R$. But, this time is divided evenly among the P processors, so the cache word moving overhead contribution to the elapsed time is $K \cdot V \cdot R$. Assuming this is the only overhead and that the normal iteration time is also divided evenly among the processors, the speedup can be expressed as:

$$\text{Speedup} = \frac{T}{\frac{T}{P} + K \cdot V \cdot R} = \frac{P}{1 + \frac{K \cdot V \cdot P \cdot R}{T}}$$

$$= \frac{P}{1 + \frac{K \cdot V \cdot P}{C \cdot S}}$$

For the above code fragment, K can be computed by simply counting the number of different accesses of adjacent columns, i.e. column L-l or L+1. In this case, K = 12 (not counting duplicate references to the same off-column element). The average value of V was estimated earlier to be about 300 nanoseconds. The value of S for this loop could be about 1200 nanoseconds. So, the speedup can be estimated:

$$\text{Speedup} = \frac{P}{1 + \frac{3 \cdot P}{C}}$$

Now, again letting P-16, and choosing the same sample values for R and C as for the sweep analysis, the following speedup estimates are obtained:

R	C	speedup
128	128	11.6
128	1024	15.3
1024	1024	15.3

And finally, a speedup estimate for the entire code can be computed, assuming that the sweep speedup is a good estimate of the heat conduction pass speedup and that the sample hydrodynamics loop speedup is a good estimate of the hydrodynamics pass speedup. The heat conduction pass consumes about 30% of SIMPLE CPU time, and the hydrodynamics pass consumes about 70%. The speedups are combined using the equation:

$$\text{Speedup} = \frac{100}{\frac{\text{percent1}}{\text{speedup1}} + \frac{\text{percent2}}{\text{speedup2}}}$$

This yields the following entire code speedup estimates:

R	C	speedup
128	128	3.7
128	1024	13.9
1024	1024	14.9

10 Synchronization and communication

The above studies have pointed out that a variety of process synchronization and communication mechanisms may be desirable for use under varying circumstances. The most obvious form of communication between processes on a processor like the S-l is through the use of shared memory, which is implemented on the S-l multiprocessor as several shared main memory modules and a cache coherence algorithm to keep the state of main memory and local caches consistent throughout all read and write accesses.

Shared memory does not necessarily directly implement the desired synchronization primitives, however. The (statically) most frequent synchronization primitive used in partitioned SIMPLE is the synchall call described earlier. Recall that it forces each process to wait at a given point of execution until all processes have arrived, after which all processes may continue. Synchall can be easily implemented in terms of classic Dijkstra-style P and V semaphore operations. For example, letting MAXPROC be the number of processes, if SLEEPINGPROCS is of type integer and MUTEX and SLEEP[1..MAXPROC] are semaphores, the following code can be used to implement synchall on process number PR:

```
(* Initially SLEEPINGPROCS=O, MUTEX=0, SLEEP[1..MAXPROC]=0 *)
P(MUTEX);
SLEEPINCPROCS := SLEEPINCPROCS + 1;
if SLEEPINCPROCS = MAXPROC then
    begin
    for I := 1 to MAXPROC do V(SLEEP[I]);
    SLEEPINGPROCS := 0;
    end;
V(MUTEX);
P(SLEEP[PR]);
```

The performance of this code in practice would of course depend very greatly on the underlying implementation of the P and V primitives. Also, it is important to note that in this code one process (the last one to execute the synchall) is responsible for issuing the V's that wake up *all* of the other processes. If the CPU time required for executing a V primitive is large enough compared to the CPU time between synchalls, and if the number of processes is large enough, this can be a severe performance bottleneck.

For allowing the implementation of synchronization primitives, the S- 1 architecture contains "conditional move" instructions. One such is the MOVCSF ("move conditionally, skip on failure") instruction. This instruction tests to see if the values of its first and second operands are equal. If so, the contents of the first operand are replaced by the contents of register %12 (decimal). If not, the first operand is left unchanged and a skip is taken to the skip destination. The instruction operates indivisibly, so that nothing can change the value of the first operand before it is (conditionally) replaced.

Synchall can also be implemented in terms of the MOVCSF instruction. The following example implementation is written in S-l assembler code. It is implemented at a very low level, without any operating system calls such as might be desired for a more general implementation - all waiting is busy-wait looping. Notice that the process local index SW is used to toggle between the first and second words of SLEEPINGPROCS on successive synchalls, to avoid race condition trouble if one process reaches its next synchall before another process has realized it is time to wake up from the previous synchall.

```
;;; Initial ly(SW)=0, (SLEEPINGPROCS)=0, (SLEEPINGPROCS+4)=0
INCSLEEP: MOV A,SLEEPINGPROCS(SW)
          INC X12.,A
          MOVCSF SLEEPINGPROCS(SW),A, INCSLEEP   ;Increment SLEEPINGPROCS indivisibly
          SKP.NEQ %12.,MAXPROC,SLEEP
          MOV SLEEP1NGPROCS(SW),#0               ;If incremented to MAXPROC, zero it
SLEEP:    JMPZ.NEQ SLEEPINGPROCS(SW),SLEEP       ;Wait for SLEEPINGPROCS = 0
          SUBV SW,SW,#4                          ;Switch: SW:=4-SW
```

For some kinds of synchronization and communication, it appears that a mechanism other than simple shared memory is very desirable. The cache line size of 16 words requires a substantial amount of overhead per cache line moved from one processor to another. This overhead can be amortized over the 16 words if the memory access pattern causes most of the 16 words to be used before the cache line must be moved again. This type of amortization is the reason that SIMPLE arrays were assumed to be stored by columns, and then the rows were processed in blocks in the heat conduction sweep analysis. In a straightforward non-blocked implementation, the sweeps in SIMPLE would require that about 4 words per C/P microseconds be transferred between processors. Especially for small numbers of columns, the "bulky" 16 word cache moves can be a significant bottleneck.

Also, timing cache line mesh data moves only includes communication overhead, and does not account for any synchronization overhead (mentioned in the "partitioning SIMPLE" section as A WA IT and SICNA L primitives). So, it is reasonable to propose a new general purpose mechanism which combines the functions of communicating small packets of data at high bandwidth and providing synchronization between the processes sending and receiving the data.

The new proposal is a simple inter-processor message sending mechanism. Messages are transmitted on one-way "links," which are allocated in I/O memory space much like normal I/O mechanisms. The I/O memory allocation is performed by the operating system, so that transparent reallocation can be done if it becomes necessary to move a process from one processor to another. Once the link is set up, the user processes can use it at high speed via special instructions without substantial operating system intervention.

The user instructions are called SNDMSC and RCVMSG. They are specified to operate on small messages (doublewords) at very low overhead per message transmission. The hardware contains a small amount of buffering for smoothing the message flow, but both instructions have failure returns, indicating that either the buffers are momentarily full (for SNDMSC) or empty (for RCVMSG). It is expected that both instructions can execute in the 100-200 nanosecond range, with a message latency between processors limited largely by physical factors such as interprocessor cable lengths.

As an example, a possible implementation of the forward sweep part of the slightly simplified SIMPLE code fragment which appeared earlier using AWAIT and SIGNAL is included below:

```
for K := KMN to KMX do
    begin
    if PR > 1 then RCV2WORDS(LINK[PR-1],AKLM1,BKLM1) else
        begin
        AKLM1:=A[K,LMN[1]-1];
        BKLM1:=B[K,LMN[1]-1];
        end;
    for L := LMN[PR] to LMX[PR] do
        begin
        A[K,L] := Q[K,L] / AKLM1;
        B[K,L] := (Q[K,L-1]*BKLM1) / AKLM1;
        AKLM1 := A[K,L];
        BKLM1 := B[K,L];
        end;
    if PR < MAXPROC then SEND2WORDS(LINK[PR],AKLM1,BKLM1);
    end (*for Ku);
```

11 Directions for future study

In a broad ranging study such as this, there will always remain interesting problems to be addressed. There is more work to be done in each of the areas discussed in this report, and there are also many other related areas requiring study.

The simulation of partitioned SIMPLE is still **incomplete in several ways** mentioned earlier, especially in simulating the overhead time required for cache line moves and/or synchronization primitive execution. Also, a higher-level simulation could **be** done **which** does **not** actually **solve the** physics equations, but still models the multiprocessor behavior of **the** code for various mesh sizes and other parameters. The analytic studies of SIMPLE could be continued **in** several directions, including detailed analysis of other code sections, or studying previously analyzed sections under different assumptions, such as the **use** of SNDMSC and RCVMSC primitives. A more quantitative statement about the sensitivity of the **speedup** of various code **segments** to variations in the mesh size would also **be** useful.

Further detailed study of synchronization and communication mechanisms **is** desirable. Such **mechanisms should be as easy to use and as general as possible, but must not sacrifice performance.** It has already **been** observed that a variety of mechanisms with **a variety of functional** and timing characteristics is likely to be needed. In conjunction with these **mechanisms, more** study **should be done on** general techniques for partitioning of applications. **The special issues arising in debugging** a multiprocess implementation are particularly important. More tools need to be developed for **evaluating the** effectiveness of alternative implementations.

Another important dimension of study is the range of applications chosen for partitioned implementation. Study of partitioning in detail should be done (as it was for SIMPLE) for several **other** real-world applications, such as those mentioned in the section on “selection of a sample **application.” Also,** several entirely different non-numerical **areas of application should be** considered in more detail for multiprocessing feasibility.

One final area of investigation needed is the implications of trying to use the S-i multiprocessor hardware as cost-effectively as possible. **A** major topic is **the** interaction of **the** partitioned multiprocessing approach with the powerful vector processing capabilities of the S-1 Mark IIA. One other topic mentioned earlier is researching the possible implementation of a very efficient multiprocessor matrix transposition algorithm, for possible use in situations **where** matrix processing does not efficiently align with the chosen data partitioning of matrices.

12 Conclusion

This study to date has added to the evidence in favor of the partitioned application mode of multiprocessor use. It has demonstrated that applications representative of real-world large scale problems can reasonably be considered for multiprocessor partitioning. Some simulation and analytic estimates of code speedup have been obtained. Some general methodologies for partitioning have been suggested, and some specific mechanisms for multiprocess cooperation have been proposed.

It seems certain that general purpose multiprocessors will play a large role in the future spectrum of the world's computing needs. Part of this role will be assumed by large scale multiprocessors executing some of the most compute-intensive applications, partitioned across multiple processors to gain valuable increases in raw computing power per wallclock hour.

13 Acknowledgments

I am indebted to many of my fellow staff members of the S-l Project for encouraging this research, and particularly to Mike Farmwald, whose stimulating advice and helpful comments from beginning to end of this work have been most warmly appreciated. Professor Cio Wiederhold also provided much useful guidance. I am also grateful to the Fannie and John Hertz Foundation for a Hertz Fellowship, which provided personal support during much of the time that this research was conducted.

The S-l Project is sponsored by the Naval Material Command, through the Office of Naval Research and the Naval Electronic Systems Command. The Lawrence Livermore Laboratory is operated by the University of California for the US Department of Energy under Contract W-7405-Eng-48.

14 References

[1] Baudet, Gerard M., The Design and Analysis of Algorithms for Asynchronous Multiprocessors, Ph.D. thesis, Carnegie-Mellon University, April 28, 1978.

[2] Chazan, D. and W. Miranker, Chaotic Relaxation, *Linear Algebra and Its Applications, 2,* 1969, 199-222.

[3] Crowley, W. P., C. P. Hendrickson, and T. E. Rudy, The SIMPLE Code, Lawrence Livermore Laboratory report no. UCID-17715, February 1, 1978.

[4] Flynn, M. J., Very High-Speed Computing Systems, *Proceedings of the IEEE,* 54(12), December 1966, 1901-1909.

[5] Forslund, D. W., and C. W. Nielson, Vectorized PIC Simulation Codes on the Cray-I, *Proceedings of the 1978 Workshop on Vector and Parallel Processors,* Los Alamos, New Mexico, September 20-22, 1978.

[6] Hammersley, J. M., and D. C. Handscomb, *Monte Carlo Methods,* Wiley, N. Y., 1964.

[7] Jones, Anita K. and Peter Schwarz, Experience Using Multiprocessor Systems: A Status Report, technical report, Carnegie-Mellon University, October 14, 1979.

[8] Kershaw, David S., The Incomplete Cholesky-Conjugate Gradient Method for the Iterative Solution of Systems of Linear Equations, Journal *of Computational Physics,* 26(1), January 1978, 43-65.

[9] Langdon, A. Bruce and Barbara F. Lasinski, Electromagnetic and Relativistic Plasma Simulation Models, *Methods in Computational Physics 16, 327-366.*

[10] Plechaty, Ernest F., and John R. Kimlinger, TARTNP: A Coupled Neutron-Photon Monte Carlo Transport Code, Lawrence Livermore Laboratory report no. UCRL-50400 vol. 14, July 4, 1976.

[11] Raskin, Levy, Performance Evaluation of Multiple Processor Systems, Ph.D. thesis, Carnegie-Mellon University, August 1978.

[12] S-l Project Staff, Advanced Digital Processor Technology Base Development for Navy Applications: The S-l Project, Lawrence Livermore Laboratory report no. UCID-18038, 1978.

[13] Widdoes, L. C., High-Performance Digital Computer Development in the S-l Project, *Proceedings of IEEE CompCon,* Spring 1980, in press.

[14] Zimmerman, George B., Numerical Simulation of the High Density Approach to Laser Fusion, Lawrence Livermore Laboratory report no. UCRL-74811, October 17, 1973.