UCRL--99893

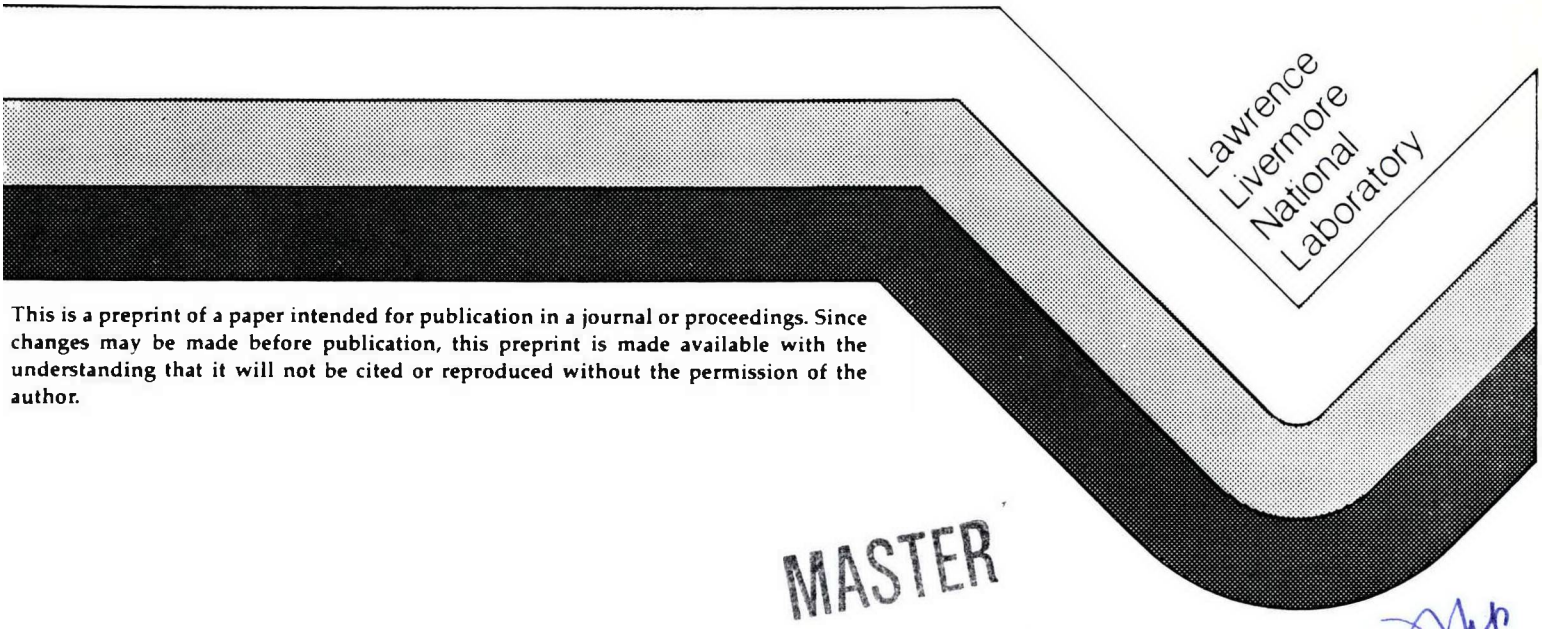
DE89 010223

Partitioning a Finite Difference Code for a Local Memory Multiprocessor[1]

Bryan Lawver

The 4th Conference on HyperCube
Concurrent Computers and Applications
Monterey, CA.
March 6-8, 1989

March 6, 1989



Partitioning a Finite Difference Code for a Local Memory Multiprocessor*

Bryan Lawver, A.J. DeGroot †

March 6, 1989

Abstract

The TSAR code, a 3d finite difference EM model, was partitioned into parallel modules where each processor computed a subset of the 3d mesh. The multiprocessor was arranged in a square mesh of processors with the full range of one dimension in each processor and sub ranges in the other dimensions. Finding the range and checking for independence of array elements over those sub ranges is the main problem of partitioning. In addition with local memory multiprocessors, the array elements are distributed over the processors' local memory and a communications structure must be available which allows non-local elements to be accessed. Element communication must be fast enough or few enough to allow good utilization of the processor compute power. Automated partitioning was not available but recent work provides some hope in this area.

TSAR stands for Time-Domain Scattering and Response software that has been used for many years to model electric and magnetic waves in a three dimensional box. This code was partitioned to run on the SPRINT processor. SPRINT is a Systolic Processor with a Reconfigurable Interconnection Network of Transputers. It incorporates 64 floating point transputers. Results of this partitioning resulted in performance nearly equal to a CRAY XMP for identical problems.

The Multiprocessor Partitioning Problem

The problem is does a compiler do it? One can hope to drop the old code in one end and have a nicely partitioned multiprocessor drop out the other. For multiprocessors the answer is still mostly *no*. A multiprocessor allows each processor to operate independently of the other processors until synchronization is specified. To minimize communication and synchronization between processors, partitioning should divide the code into large pieces of independent code to be distributed among processors. In many scientific codes and especially the TSAR finite difference code, the large pieces of parallel code are found in loops which iterate over the subscripts of the 3d arrays. The finite difference code updates each element of the mesh to find a new estimate for the six state variables of the model at each time step. By assigning a group of elements to each processor of an N processor multiprocessor, we can hope for an N times speed-up. Finding which part of the loop to partition among processors remains the main challenge of the partitioning problem. Some automated work is now being reported [4,1].

Global memory multiprocessors allow the problem to be distributed to individual processors with a single global copy of the data arrays. As long as two processors don't try to update the same array element then N times speed up is usually possible. Memory capacity, connectivity and performance limit the number of processor a global memory multiprocessor can have. Local memory multiprocesors require the data to be localized to a particular processor with nonlocal accesses handled by a communications system. A communications system does not automatically improve connectivity or performance, but when matched with the appropriate code can be used to connect many more processors. The problem becomes can we find large enough blocks of parallel code that minimizes communications. Secondly can we deterministically move the data such that one processor sends the data when it generates it and the other processor reads the data when it needs it and neither waits unless it hasn't yet received all of its new data. The architecture of the Transputer supports this second part very well.

Partitioning Loops

The *DoAll one to N* loop can be partitioned into N different blocks of code to run on N or some factor of N processors as long as all references within the loop are independent. A procedure call within the loop obscures the process of determining reference independence within the block (figure 1).

This example can not be partitioned until the assignments and references in *Sub* are examined. The method of in-line expansion can sometimes solve this problem, but recursion will defeat this method. The method expands each procedure call in-line so a single block can be checked for data dependency. Another method developed by Li and Yew [4] to examine procedure calls within loops is

*This work was performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore Laboratory under contract W-7405-Eng-48. UCRL-99893

†Lawrence Livermore National Labs PO 808 , L-156 Livermore, Ca. 94550 415-422-6234

```
Do      i = 1, N
        Array(i) = ...
        ⋮
        CallSub(A, i)
        ⋮
endDo
```

Figure 1: Loop with procedure call

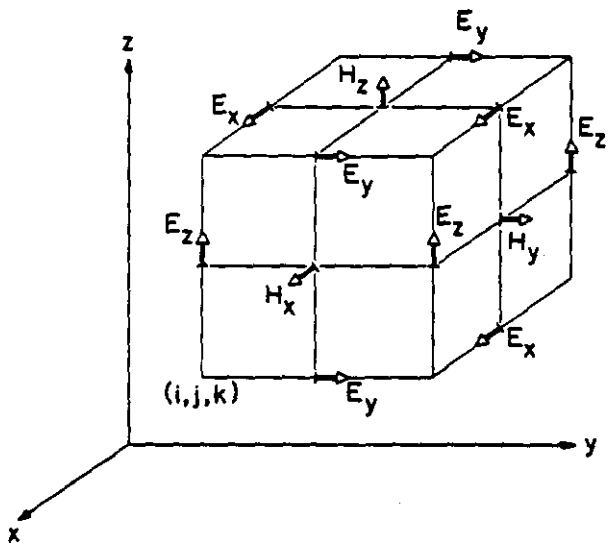


Figure 2: Positions of Field components about a unit Cell.

called subscript propagation. It attempts to extract out of procedures the range of referrences based on loop variables. With this information a partitioning of the loop can be found if the referrences are independent. Some preliminary work on a compiler has been done at the Center for Supercomputing Research and Development using this method.

Partitioning TSAR

A non automated form of subscript propagation was used to develop a partitioning of the TSAR code. For most of the loops it was easy to observe that the loop indices were used as subscripts in the field arrays. Two loops were not but they were sufficiently separate from the other loops that parallelization and partitioning were easy.

The code updates electric fields (E) from surrounding magnetic fields (H) in all three dimensions.[3] For each cell the edges of a face represent field components which contribute to a field emanating from the center of the face see figure 2. The H cells are offset by 1/2 cell in each direction so that the E fields emanate from the center of these cells also. Numbering of edges drops the 1/2 cell offset from the indices. This yields three equations for E fields one for each direction and three more for the H fields. Additional equations estimate field values along the boundaries of the mesh. These equations only compute the E fields and there are 6 face estimates and 12 edge estimates for each of the three E field directions. This produces 60 equations to estimate all of the field equations. There was also another 8 equations to estimate the source conditions.

When distributing cells to processors, most cells are completely contained in just one processor The 68 equations that might apply to the cell are unmodified. But if the partitioning of field values to processors divides a cell in half in either of the two dimensions which processors divide three space then some or all of the equations have to be modified. Rather than rewriting all of the equations it was decide to extend the local memory in the x and y dimensions by one cell so that the partitioned cell exists in both processors. One processor always has enough information to update the field while the neighbor which needs the field value also has a copy. The overlap was identified by examining the 60 equations and by moving planes or vectors of data from one processor to another. Each precessor could be a beginning edge, a middle part or the end edge. These three cases times the 68 equations produced approximately 160 different cases to examine. This yielded 6 planes of data to move for overlap between the processor computing the field value to the neighbor which needs the data to compute its field values. Also 40 vectors of data were identified for moving at the boundary to estimate field values. The planes were move between all processors except when there was no processor in that direction whereas the vectors of data were used to estimate boundary conditions so if the processor has only internal cells with no outside boundary then no vector data moves.

SPRINT System

The SPRINT system [2] includes 64 Transputers arranged in a square mesh. TSAR is a three dimensional finite difference code. Two dimensions of the TSAR code were distributed across the mesh of Transputers. Each Transputer was connected to nearest neighbors in two dimensions. Each index spans a row or column of processors. At the hardware level a Transputer connects to each of four neighbors except those which are on the edge of the mesh. The connection allows data to pass over a physical link. Each link is supported by an on chip protocol which verifies the data is read before a new datum is sent.

We were able to start with a modular well written code with approximately 50 routines. Of the routines, 10 were pre- and post-processing routines which remained unmodified on the host. New routines were provided for the host to broadcast the initialized common blocks and to distribute initial state data to the individual processors. The remaining routines were used to build the partitioned code with very minor parameterization changes made to the start and stop indices of loops. A couple of new routines interface with the host's broadcast and distribution routines. Two new routines were created to move the 6 planes of overlap data and 40 vectors. One routine moves

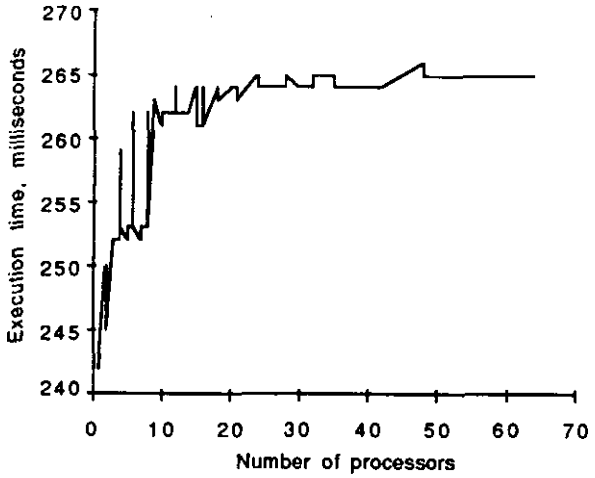


Figure 3: Performance N processors each with a 16x16x6 mesh.

E field overlap after the update to E fields and one moves H field overlap after the H fields update.

Performance

The original SPRINT system had 64 processors with enough memory for each processor to contain a 16 by 16 by 6 mesh of cells. Figure 3 shows the SPRINT needing 250 ms to complete one update of the mesh where 64 processors update 128 by 128 by 6 cells. The equivalent size mesh when updated on a Cray XMP requires 218 ms. This produces an equivalent performance of 87% for the 64 processor SPRINT. At this size problem one in eight points are part of the overlap cells which move each step.

Figure 3 also shows a nearly linear speed-up. Some of the smaller configurations are faster because of less communications, but they also compute fewer cells because not all of the boundary equations were in place in this version. For larger configurations than 64 Transputers we predict linear speed-up.

More memory is planned for the SPRINT so that larger and more interesting size problems can be run on the SPRINT. In addition all of the boundary and input options of the code require more memory to hold the program. Shortly we expect to run problems which are 100 by 100 by 60 in size with the proportionally longer time required.

Conclusion

Propagation of subscripts was independently developed for this partitioning of TSAR. The Parafrase work of Li and Yew generated tables to resolve the data dependence problem. We used hand generated tables and used a spread sheet for storage and analysis. The code has been run on the 64 Transputer system and the performance was measured to be 87% of a Cray XMP.

References

[1] Micheal Burke and etal. Automatic discovery of parallelism. In *ACM/SIGPLAN PPEALS Conference Proceedings*, New Haven, Ct., July 1988.

[2] A.J. De Groot, E.M. Johansson, and S.R. Parker. Sprint- the systolic processor with a reconfigurable interconnection network of transputers. In *IEEE Fifth conferrence on Real-Time Computer Applications in Nuclear Particle and Plasma Physics*, San Francisco, Ca., May 1987.

[3] Mur G. Absorbing boundary conditions for the finite-difference approximation of the time-domain electromagnetic-field equations. *IEEE Transactions on Electromagnetic Compatibility*, EMC-23(4):377–382, November 1981.

[4] Zhiyuan Li and Pen-Chung Yew. Efficient interprocedural analysis for program parallelization and restructuring. In *ACM/SIGPLAN PPEALS (Parallel Programming: Experience with Applications, Languages and Systems) Conference Proceedings*, New Haven, Ct., July 1988.