Speculative Parallelization of Partially Parallel Loops *

William McLendon III   Lawrence Rauchwerger
mclendon@cs.tamu.edu   rwerger@cs.tamu.edu

Technical Report 00-015
Department of Computer Science
Texas A&M University
May 16, 2000

Abstract

Keywords: Parallelization, Strongly Connected Component, Cycle.

*Draft, This research supported in part by NSF CAREER Award CCR-9734471, NSF Grant ACI-9872126, NSF Grant EIA-9975018, DOE ASCI ASAP Level 2 Grant B347886, Sandia National Laboratory and Hewlett-Packard Equipment Grants

1 Motivation

The Boltzmann transport equation can be solved using the discrete ordinates method for problems involving fire simulation, neutron transport, etc. Sandia has developed a solver called SnRad, which is an asychronous, parallel, message-passing algorithm for solving transport problems on general unstructured grids. Unfortunately, if a cycle exists in the dependencies the SnRad solver will hang.

We can precondition meshes for the solver so that they do not contain cycles initially but when shock hydrodynamics is applied in a multiple-timestep problem, it is possible for the mesh to deform in such a way to create cycles. Therefore we needed to develop a code to detect these cycles in parallel as the simulations run.

A strongly-connected-component (SCC) in a directed graph is the maximal subset of vertices containing a directed path from each verted to all others in a the subset. A SCC might contain several cycles which have at least 1 node in common among all of them. We wish to locate the SCCs in the directed graphs representing the dependencies among the elements of the meshes for which we are solving the transport equations.

2 Background

Finding the cycles for a directed graph for serial computers is solved by Tarjan[2]. That algorithm uses a depth first search (DFS) to compute the solution and has O(N) time complexity. Unfortunately DFS for our situation is P-Complete[3] for distributed graphs therefore we are trying other ways to compute this.

Meshes are traversed in transport problems following the direction of the ordinate vector. These angles correspond to the angle that a particle would be streaming along within the material. A particular element of the mesh depends on its upwind neighbors, being placed on a work queue only after every neighbor which it depends on has been executed.

The input to this program is an unstructured finite element mesh consisting of 3D hexahedra and the number of ordinates, Ω, to compute the vectors from. The vectors are computed from the parameter Ω where $nVec = \Omega * (\Omega + 2)$. For a value of $\Omega = 10$ the number of angles becomes $nVec = 120$. The angles are generated at run time. Each vector angle Ω_m represents a different traversal of the mesh therefore there is a different directed graph corresponding to each unique vector angle.

The transformation from the input 3D mesh to a graph is done with the following criteria:

- The building blocks of the mesh, M, are 3D hexahedra called *elements*.
- Each angle in the set of ordinate vectors is represented as Vec_i
- An *element* (*elt*) corresponds to a *node* (n) of the graph
- every element has 6 faces, which are shared by an adjacent element or is a mesh boundary.
- A face, f, that is shared between two elements elt_1 and elt_2 becomes an edge connecting nodes n_1 and n_2 in the dual graph.

- An edge becomes directed according to the angle, Vec_i being checked. The edge between n_1 and n_2 becomes $n_1 \longrightarrow n_2$ if the outward face normal of elt_1 is directed in the same hemisphere as Vec_i. In this case, n_2 is dependent on n_1 for the dual graph of M corresponding to Vec_i, which we will call G_i.

Each angle is tested for cycles simultaneously. In the implementation, We will describe the method used for solving in terms of a single angle. Extending to many angles is not difficult, mostly an extra loop is added to iterate over the angles.

3 Algorithm

We have an implementation of the divide and conquer algorithm for finding SCCs developed at Sandia[1] which uses a series of breadth first searches (BFS) to obtain the strongly-connected-components within directed graphs.

Algorithm 1 divide-and-conquer algorithm[1] for strongly connected components.

$DCSC(G)$
`If` V `is empty then Return`
`Select` v `uniformly at random from` V.
$SCC \leftarrow Pred(G,v) \bigcap Desc(G,v)$
`Output` SCC
$DCSC\ (\langle Pred(G,v) \setminus SCC\rangle)$
$DCSC\ (\langle Desc(G,v) \setminus SCC\rangle)$
$DCSC\ (\langle Rem(G,v)\rangle)$

4 Implementation of pSCC Detection

This code is to be implemented within the SnRad solver at Sandia to detect the SCCs which might occur when hydrodynamics calculations distort the mesh inbetween time steps.

4.1 Data Structures

We tried to use as much of the structures and routines that were available to us for loading and traversing the meshes. There were some additional data structures that were introduced to the code for the recursive strongly connected component detection.

A new data structure was created to store the information about the submesh which we are interested in. This was required because at each level of recursion, we are only interested in a particular part of the mesh. We only store lists of the elements in any given active set. From these lists we reference into the global mesh data structure to obtain the element / processor adjacency information.

```
struct mesh_ord {
  int   pivotidx;                       /* index of element to be pivot */
```

```
  int   iord;                                      /* ordinate number */
  int   size;                         /* Number of Elements in submesh */
  int   nowork;                     /* FLAG: TRUE if no more work to do */
  int   nummarked;                 /* Tracks a count of 'seen' elements */
  int   *eltid;                          /* List of global element IDXs */
  short *eltlabel;                    /* Marking Label for PIVOT sweeps */
  short *seen;                         /* mark visited elts inside sweeps */
  short *nup;                      /* # upwind dependencies for each elt */
  short *ndown;                    /* # dnwind dependencies for each elt */
}
```

We create an array of *struct mesh_ ord*, where *mesh_ ord*[i] corresponds to the i^{th} ordinate.

4.2 Algorithm Implementation

The code consists of two major operations. The function pSCC-Search() initializes the data structures and determines the angles to be checked.

```
/* pSCC-Search() : Initializes the mesh for the recursive
 * SCC detection routine                                              */
pSCC-Search( M, A )
  {
    get ordinate pairings P_i = {A_i, - A_j}
    L = pick 1 angle from each P_i to search
    G = Set of graphs G_i for every L_i
    rec-pSCC-Search( G )
  }

/*  rec-pSCC-Search() : Implements the recursive SCC
 *  detection scheme devised in [1]                                   */
rec-pSCC-Search( G )
{
  FOR every i {
    initialize each element G_i
    initialize global2local element mapping
  }
  submesh_edgefix( G )                         /* remove invalid edges  */
  subcomm_nowork( G )                          /* flag ords with no work*/
  make_subcomm( G )                            /* which procs have work */
  p_scctrim( G, DOWNWIND )                     /* Downwind sweep        */
  flag = query_workexists()                    /* was something unseen? */
  if ( flag == TRUE ) {
    make_subcomm( G )
    p_scctrim( G, UPWIND )                            /* backwards trim */
    pivot = randompivot( G )                          /* select pivot   */
    setflag_nowork( G )
    make_subcomm( G )
```

```
    p_sccmark( G, pivot, PREDECESSORS )
    p_sccmark( G, pivot, SUCCESSORS )
    getSCC( G )                                   /* {G_pred ∩ G_succ} = SCC */

    /* create submeshes for the 3 disjoint pieces of G */
    For every G_i s.t. G_i ∈ UNMARKED, G_O ← G_i
    G_P = G_pred
    G_S = G_succ
    deallocate G

    /* Recursively search the submeshes */
    IF ( |G_P| > 1 ) { rec-pSCC-Search( G_P ) }
    IF ( |G_S| > 1 ) { rec-pSCC-Search( G_S ) }
    IF ( |G_O| > 1 ) { rec-pSCC-Search( G_O ) }
  }
  else {                                          /* trim visited all of G */
    reset global2local mapping to -1
    deallocate G
  }
}
```

5 Performance Measurements

Measurements were made on a HP V-Class SMP with 16 PA-Risc 8200 processors. Grids of varying sizes and complexity in terms of number of cycles were tried.

5.1 Test Meshes

Figure 1 shows some information about the meshes which were used for testing. The total number of cycles is reported on this graph. The meshes range from small to relatively large (although meshes used on transport sweeps may have millions of elements). We could Ring and cube50 have no cycles, therefore the initial pass will visit every node in the graph and determine that no cycles existed.

Mesh	Elements	Ordinates	Angles Checked	# SCCs	Total SCC Size
s20	43984	120	60	4	80
brick2	1391	120	60	193	1075
brick3	8305	120	60	18	108
sphere2	36712	120	60	1	8
ring	47160	120	60	0	0
cube50	125000	120	60	0	0

Figure 1: Mesh sizes, number of angles checked and SCC data from tests.

As the graph has more cycles, the performance of the code will suffer from a serialization. The worst possible case would be where the entire mesh is composed of small 3 or 4 element SCCs. We believe that that nightmare case is unlikely to happen for the meshes we are checking, but improving on this degeneracy is definitely something to improve upon in the future.

Meshes s20, brick3, and sphere2 have few cycles relative to mesh size. These cases are the ones that are expected to occur more frequently in real problems. The speedups for these meshes scale well with processors, since the problem size was not increased with processors, the performance tails off some when we get to 16 processors on this machine.

Processors	s20	brick2	brick3	sphere2	ring	cube50
1	1.00	1.00	1.00	1.00	1.00	1.00
2	2.05	1.45	2.04	2.11	2.01	2.03
4	4.35	1.65	4.11	4.32	4.27	3.99
8	8.02	1.18	7.78	8.77	8.38	7.70
16	14.05	0.67	11.53	16.02	15.85	13.55

Figure 2: Measured speedups on Hewlett Packard V-Class SMP with 16 PA-RISC 8200 processors.

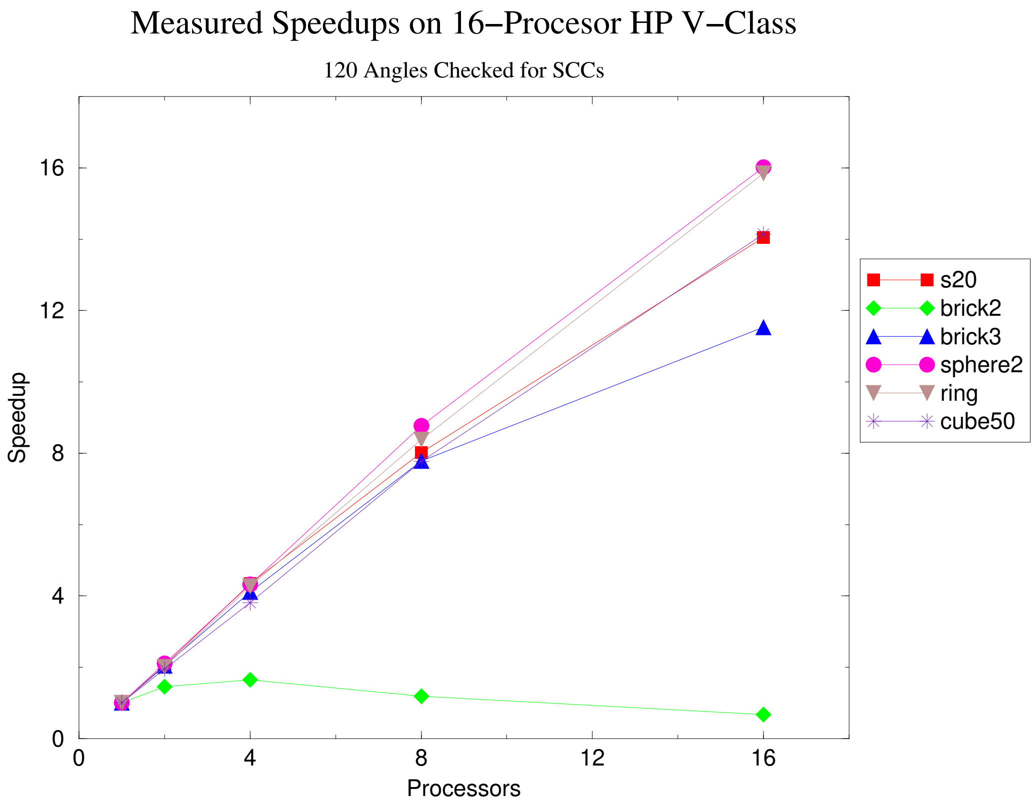

Figure 3: Speedup graphs showing performance of pSCC detection.

The code serializes for the brick2 mesh. This is somewhat expected in this implementation due to the fact that brick2 contains many cycles.

6 Future Work

In the future, We plan to put this code up on the Intel Tflops MPP at Sandia National Laboratory and tune it for work on large machines. Performance on our 16-processor V-Class gives some promising numbers, giving poor performance usually when there are a large number of small SCCs present in the grid. That case is a known instance where the algorithm currently serializes. We will be working on this degeneracy as well as streamlining the code to gain more parallel efficiency.

We are looking into other techniques which might also allow parallel detection of SCCs and possibly avoid the nightmare case for this algorithm. We also are investigating other graph oriented problems which might benefit from parallel techniques such as these.

References

[1] Lisa Fleischer, Bruce Hendrickson, and Ali Pinar, *On Identifying Strongly Connected Components in Parallel*, Proceedings of Irregular'2000

[2] R. E. Tarjan, *Depth first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146-160.

[3] J. H. Reif, *Depth-first search is inherently sequential*, Inform. Process. Lett., 20 (1985), pp. 229-234