Work-efficient Techniques for the Parallel Execution of Sparse Grid-based Computations

TR91-042

Jan F. Prins

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

Work-efficient techniques for the parallel execution of sparse grid-based computations

Jan F. Prins

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
prins@cs.unc.edu

Abstract

Computations that consist of repeated applications of a local function to every data item in a grid are easily and efficiently implemented in mesh-connected parallel computers. When grids are large and grid elements requiring recomputation are sparsely distributed, the work performed in such implementations is non-optimal compared to an optimal sequential solution. A technique is described to achieve work-efficient (optimal speedup) implementations for this class of problems that is largely insensitive to the clustering and distribution of the elements participating in the computation.

1. Introduction

This note describes a method for achieving work-efficient parallel execution of certain large grid-based computations. The class of computations addressed is characterized by the repeated evaluation of a local function[1] *f* at every point of data in a grid, and includes finite difference problems, cellular automata simulations and image processing problems. By work-efficient we mean that the parallel algorithm using *P* processors yields optimal speedup with respect to the best sequential algorithm on problems of size O(*P*) or larger. That is, the parallel algorithm using *P* processors runs approximately *P* times faster than the best sequential algorithm for such a problem runs on a single processor .

It is well known that this class of computations is well-suited to speedup through parallel execution on locally-connected parallel computers when the data grid can be decomposed over processors in such a manner that local function evaluation need only use local communication in the machine. Since the performance of local communication can remain essentially constant with increasing numbers of processors (the same can not hold for global communication), such computations can achieve optimal speed-up even with large numbers of processors.

[1]A pointwise function is *local* if its value at any given point on the grid depends only on nearby points.

We are concerned here with extension of optimal speedup to cases where, in a sequential algorithm, an asymptotic improvement can be achieved by avoiding re-evaluation of the local function *f* at locations on the grid that define a local fixed point for *f*. This occurs, for example, in edge-directed diffusion [Beghdadi] where each pixel's value is adjusted by a weighted average of its neighbor's values unless that adjustment would result in a change smaller than some threshold. When a pixel value reaches a local fixed point for the averaging function, it need not be re-evaluated until a neighboring pixel's value changes sufficiently.

An efficient sequential implementation of such a problem recomputes *f* only on "active" grid points in each iteration, while a simple-minded parallel algorithm might re-evaluate *f* at each grid point, regardless of whether it is active. If active grid points are sufficiently sparse, a sequential algorithm performing an iteration in O(active points) time will outperform a parallel algorithm on a *P* processor machine performing an iteration in O(grid points/*P*) time. The technique described here remedies this situation to achieve O(active points/*P*) performance in the parallel case.

2. Example application

We illustrate the application of the technique in the parallel evaluation of the familiar "life" cellular automaton designed by [Conway]. This automaton is described by a particularly simple set of rules on a two-dimensional grid of cells in which each cell's state is either *occupied* or *empty*. The state of a cell evolves each generation according to a local function defined as follows on the state of the current generation:

- an empty cell becomes occupied if it has exactly 3 occupied neighbors.
- an occupied cell becomes empty if it has fewer than 2 or more than 4 occupied neighbors, otherwise it remains occupied.

The application of the rules to a sample configuration is illustrated in Figure 1.

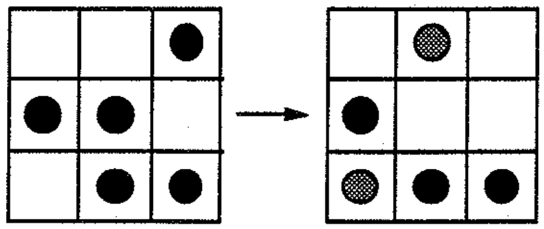

Fig. 1. Application of the *Life* rules to a sample configuration. Solid circles denote surviving occupied cells, shaded circles denote newly occupied cells.

It follows from the rules that the only cells that can change state are occupied cells or their immediate neighbors. All other grid positions define a local fixed point under the rules above (unoccupied cell with unoccupied neighbors), and need not be evaluated in the computation of the next generation. Since each occupied cell has a constant number of neighbors (eight), an O(occupied cells) sequential algorithm can be written for the Life cellular automaton.

Work-efficient parallel implementations of the sparse and dense cases of the life computation were developed for the MasPar MP-1 and their performance quantified. In both cases we consider a square cellular automaton grid of size $\sqrt{N} \times \sqrt{N}$. In the dense case all N cells are updated in each generation; in the sparse case L cells are updated in each generation where L is O(occupied) and $L \ll N$. The timings of the dense case implementation are included to characterize the relative performance of the two approaches.

The MasPar MP-1 is a distributed-memory SIMD machine with processors connected in a 2-D mesh with torus topology. For simplicity we consider the machine to be of size $P = 2^{2m}$, with processors arranged in a square mesh of size $2^m \times 2^m$. All implementations described in this note extend to "rectangular" machines of size $P = 2^k$ for odd k. The key architectural features of the MP-1 used in the implementations are processor addressing-autonomy (each processor can access a locally-specified memory location in a single step) and the toroidal mesh interconnect.

3. Dense Implementation

The simplest implementation of the life automaton is obtained when each processor computes the state of exactly one cell, i.e. when $N = P$. Under these circumstances an MP-1 programmed in MPL takes 33µs to compute the next generation. This is independent of the total machine size P, hence the update time per cell $TD(P,P) = 33/P$ µs or about 2 ns/cell for a $P = 16$K processor machine.

remark: The number of occupied neighbors S(i,j) of cell (i,j) can be evaluated in four communication steps on the mesh (instead of eight) by expanding the sum of the eight neighboring occupancy values $A(i\pm1,j\pm1)$ and factoring:

$$T(i,j) = A(i-1,j) + A(i,j) + A(i+1,j)$$
$$S(i,j) = T(i,j) + T(i,j-1) + T(i,j+1) - A(i,j)$$

end of remark.

When the problem size is larger than the number of processors, some sort of virtualization strategy is required for the algorithm based on the decomposition of data to processors. Suppose that $N = kP$, then the strategy that minimizes communication in the implementation is a *hierarchical* decomposition that places a $\sqrt{k} \times \sqrt{k}$ "tile" of neighboring cells at each processor.

To compute the next generation, it is necessary to obtain the state of the $4\sqrt{k} + 4$ cells surrounding the tile from the neighboring cells, followed by k local evaluations of the state change function. Hence the sequential update time per-cell TD grows as:

$$TD(N,P) = \left(\frac{c_{comp}}{P} + \frac{c_{comm}}{\sqrt{NP}} + \frac{c_{ovh}}{N}\right)$$

For P = 16K processors and N = 589824 (N/P = 36 cells per processor) the generation time is 3 ms and *TD(N,P)* is 16 ns/cell. The increase in cost over the single-cell per processor case is due to the control overhead and could be improved substantially with careful attention to the representation of the cells and coding of the program.

4. Sparse Implementation

Next we consider how to adapt the parallel implementation to operate only on active cells only. Maintaining a "worklist" of active cells in each tile of the hierarchical decomposition as defined above is a possible approach but leads to large variations in worklist length over processors since the distribution of active cells is typically not uniform on the grid. In fact, activity in the life automaton typically occurs in clusters as illustrated in figure 2. A computation based on active cells per processor under a hierarchical decomposition will typically exhibit poor load balance.

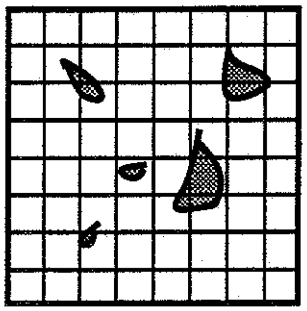

Figure 2. Distribution of active cells (shaded) in a typical large life automaton. Each small square delineates a region of cells placed in a single processor under a hierarchical decomposition.

A better distribution of cells is obtained using a "cut-and-stack" decomposition: processor (i,j) holds all cells (x,y) on the grid satisfying $(i,j) = (x,y) \bmod 2^m$. That is, $i = x \bmod 2^m$ and $j = y \bmod 2^m$. Under this decomposition a clump of nearby cells will tend to be scattered among processors in a uniform fashion. Although the worst-case distribution of active cells can still yield very poor load balance, such distributions become very unlikely as the size of the grid becomes large relative to the number of processors.

The torus topology guarantees that neighboring cells in the automaton will be placed in neighboring processors in the machine under the cut-and-stack decomposition, so that the distribution need not destroy the locality in the update step.

Each processor now has a list of occupied cells B. We represent an occupied cell in grid position (x,y) by entering the value (x,y) div 2^m on list B in processor $((x,y) \bmod 2^m)$. A given cell (x,y) with $(i,j) = (x,y) \bmod 2^m$ and $(r,s) = (x,y)$ div 2^m has an occupied neighbor at $(x, y-1)$ in the grid precisely when B at processor $(i, j\text{-}1) \bmod 2^m$ contains the value (r,s) if $j \neq 0$, or contains $(r,s-1)$, if $j = 0$. Similar definitions hold for neighbors in other directions on the grid.

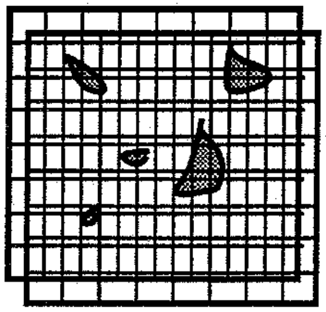

Figure 3. Cut and stack decomposition. Cells in the same processor are located at the intersection points of a single grid. Two grids are shown, corresponding to the cells in two different processors.

In order to achieve constant time evaluation of the rules in the neighborhood of each occupied cell, a total order is introduced on representations of occupied cells, and the list B at each processor is kept in sorted order according to that relation. A simple total order is the natural lexicographic ordering on the tuples in B. To apply the rules at a given processor, that processor computes the merge M of all lists B in its own and its neighboring processors. Eight merge operations can be reduced to four merge operations using a factoring analogous to the factoring of the neighbor sum in section 3. Since the lists are kept in sorted order each merge operation takes constant time per element. The only elements that appear in the lists are occupied cells, consequently the total merge time is easily seen to be O(active cells) in the neighborhood of each processor.

In computing the merge at processors on the four edges of the mesh, we increase by (0,1) all tuples arriving from the west into processor column 0, and decrease by (0,1) all tuples arriving from the east into processor column 2^m-1. Similarly we increase by (1,0) all tuples arriving from the north into the top row of processors and decrease by (1,0) all tuples arriving from the south into the last row of processors. With this adjustment, the number of times that a value (r,s) appears on a list M on processor (i,j) gives the number of occupied cells at or neighboring the grid position corresponding to (r,s) on processor (i,j). Hence a single traversal of M and B is sufficient to generate the occupied cells in the next generation in sorted order (see Figure 4). The cell represented by (r,s) at processor (i,j) will be occupied in the next generation precisely when:

(r,s) occurs three times in M at (i,j),

<u>or</u>

(r,s) occurs four times in M at (i,j) and once in B at (i,j).

In the implementation constructed, an additional rule is included that prohibits the creation of cells on the edges of the computational grid.

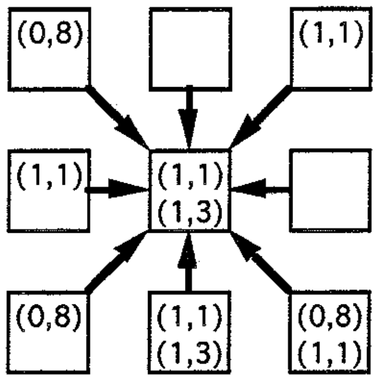

Figure 4: Each pair represents an occupied cell. Merging all nine lists of occupied cells yields the list [(0,8),(0,8),(0,8),(1,1),(1,1),(1,1),(1,1),(1,1),(1,3),(1,3)] at the middle processor. Application of the rules yields the list [(0,8)] at the middle processor in the next generation.

The update time per generation is linear in A_{max}, the largest number of occupied cells in any processor (i.e. the maximum length of of B over all processors). For a given number L of occupied cells distributed over P processors, the expected size of A_{max} can be bounded with high probability by some function $\phi(L,P)$. Thus the per-active-cell update time is given by

$$\mathrm{TS(L,P)} = \frac{\mathrm{c}\bullet\phi(\mathrm{L,P})}{\mathrm{L}}$$

Where c is the per-element time for the merge and update calculations. Empirically, $\phi(L,P) \leq 2L/P$ when $L/P > 50$. On a 16K MP-1, a large clustered life simulation varying between 280,000 and 4,000,000 active cells ($17 \leq L/P \leq 244$) achieves an average per-active-cell update time of 125 ns. Simulations of large numbers of randomly occupied cells typically lead to per-cell update times on the order of 250 ns.

Although these times are an order of magnitude larger than the per-cell update times in the dense case, it must be remembered that as long as the active cells are sparse (less than 1 in 10 cells active on average), the active-cell simulation will be faster. In particular, for the large simulation cited above, the average generation time is 48ms, while in the dense case the generation time (in this case the automaton size is 16384 x 16384) is nearly 5 seconds or about one hundred times slower.

A sequential implementation of the active cell strategy requires complex data structures to represent the active cells and to organize the evaluation of all active cells and their immediate neighbors. Efficient sequential implementations, however, use word-parallel logical operations in the computation that can offset this overhead. Nevertheless, for large simulations we found a 4K MP-1 typically 10 times faster than a DECstation 5000 using the efficient sequential algorithm described.

4. Conclusion

We have adapted an iterative local computation on a grid to update only active elements in the computation given that these elements may be irregularly distributed and sparse. The general approach was to use a cut-and-stack decomposition to obtain load balancing and to carefully choose a representation of the active points that makes application of the local computations simple and efficient.

The technique presented can be used in a large class of grid-based iterative problems, even some that at first appear not to exhibit local fixed points. For example, simulations that require different time scales at different places on the grid can be viewed as operating on the finest time scale and dynamically introducing local fixed points for those grid points that may be updated in larger time steps.

Bibliography

[Beghdadi] A. Beghdadi, A. Le Negrate, "Contrast Enhancement Technique Based on Local Detection of Edges", *CVG&IP* 46, 1989.

[Fox] G. Fox et al. *Solving Problems on Concurrent Processors,* Vol 1, Prentice–Hall 1988.

[Wolfram] S. Wolfram, *Theory and Applications of Cellular Automata*, World Scientific Publishing, 1986.