Parallel Solutions of Simple Indexed Recurrence Equations

Yosi Ben-Asher[1] and Gady Haber[2]

[1] Dep. of Math. and CS. Haifa University 31905 Haifa, Israel,
yosi@mathcs.haifa.ac.il
[2] IBM Science and Technology, Haifa, Israel,
haber@haifasc3.vnet.ibm.com

Abstract. We consider a type of recurrence equations called "Simple Indexed Recurrences" (SIR) wherein ordinary recurrences of the form $X[i] = op_i(X[i-1], X[i])$ $(i = 1 \ldots n)$ are extended to $X[g(i)] = op_i(X[f(i)], X[g(i)])$, such that op_i is an associative binary operation, $f, g : \{1 \ldots n\} \mapsto \{1 \ldots m\}$ and g is distinct.[1] This extends our capabilties for parallelizing loops of the form: for $i = 1$ to n { $X[i] = op_i(X[i-1], X[i])$ } to the form: for $i = 1$ to n { $X[g(i)] = op_i(X[f(i)], X[g(i)])$ }. An efficient solution is presented for the special case where we know how to compute the inverse of op_i operator. The algorithm requires $O(\log n)$ steps with $O(n/\log n)$ processors. Furthermore, we present a practical and a more improved version of the non-optimal algorithm for SIR presented in [1] which uses repeated iterations of pointer jumping. A sequence of experiments was performed to test the effect of synchronous and asynchronous message-passing executions of the algorithm for $p << n$ processors. This algorithm computes the final values of $X[]$ in $\frac{n}{p} \cdot \log p$ steps and $n \cdot \log p$ work, with p processors. The experiments show that pointer jumping requires $O(n)$ work in most practical cases of SIR loops, thus forming a more practical solution.

1 Introduction

Ordinary recurrence equations of the form $X_i = op_i(X_{i-1}, X_i)$ $i = 1 \ldots n$ can be generalized to what we refer to as *Indexed Recurrence* (IR) equations. In IR equations, general indexing functions of the form $X_{g(i)} = op_i(X_{f(i)}, X_{g(i)})$ replace the $i, i-1$ indexing of the ordinary recurrences. Efficient parallel solutions to such equations can be used to parallelize sequential loops of the form:

for $i = 1$ to n { $X[g(i)] = op_i(X[f(i)], X[g(i)])$ }

if the following conditions are met:

1. $op_i(x, y)$ is any segment of code that is equivalent to a binary associative operator and may be dependent on i.
2. the index functions $f, g : \{1..n\} \mapsto \{1..m\}$ do not include references to elements of the $X[]$ array.

[1] This paper is a continuation of the work on IR equations presented in [1].

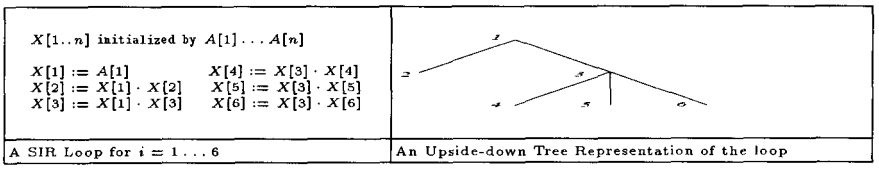
3. the index function g is distinct, i.e., for every indexes i, j if $i = j$ then $g(i) = g(j)$.

In practice, such solutions can be used to parallelize sequential loops within given source code segments.

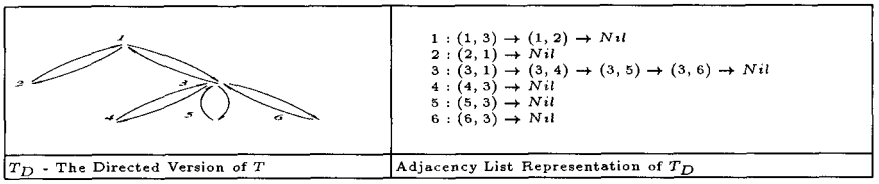
2 An Efficient *SIR* Algorithm for the Case Where the Inverse of op_i is Known

In this section we describe an efficient parallel algorithm for SIR loops, providing we know how to compute the inverse operation op_i^{-1}. The algorithm requires $O(\log n)$ steps with $O(n/\log n)$ processors.

The first step in the algorithm is to translate the SIR loop into its upside down tree representation T, in which every vertex i $(1 \leq i \leq n)$ represents the $g(i)$ index and every edge $e = \{i, j\}$ represents the dependency $op_i(X_{g(j)}, X_{g(i)}$ where $g(j) = f(i)$ and $j < i$. For example consider the following upside-down tree representation of a SIR loop:



We then apply the *Euler Tour* circuit on T in order to compute all the prefixes of the vertices in T. An Euler Tour in a graph is a path that traverses each edge exactly once and returns to its starting point. The Euler tour circuit which operates on a tree assumes that the tree is directed and that it is represented by an Adjacency List. In the directed tree version T_D of T, each undirected edge $\{i, j\}$ in T has two directed copies - one for (i, j) and one for (j, i). For example, consider the directed version of the tree for the above example, and its Adjacency List representation:



Our main concern when coming to construct the adjacency list of T_D, is finding the list of all the children of each vertex i in T_D efficiently; i.e., finding all vertices $j < i$ which satisfy that $g(j) = f(i)$. Sorting all the pairs $< f(1), g(1) >, \ldots, < f(n), g(n) >$ using $f(i)$ as the sorting key, will automatically group together all vertices which have the same parent. The sorting process can be done efficiently using the Integer Sort algorithm [3] which operates on keys in the range of $[1..n]$.

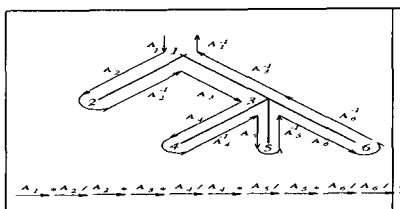
After the adjacency list of T_D is formed, we can construct the Euler tour of T_D in a single step according to the following rule:

$$EtourLink(v, u) = \begin{cases} Next(u, v) \text{ if } Next(u, v) \neq Nil \\ AdjList(u) \text{ Otherwise} \end{cases}$$

by allocating a processor to each pair of edges $(v,u),(u,v)$ in T_D.

Once the Euler tour of T_D is constructed, in the form of a linked list, we can compute the prefixes of all the vertices of T_D, using the efficient parallel prefix algorithm on the linked list [2], after initializing all the edges of the tree with the following weights:

- The weight of each forward edge $< f(i), g(i) >$ in T_D is initialized by $A_{g(i)}$, where A is the array of initial values of the X array of SIR.
- The weight of each backward edge $< g(i), f(i) >$ is initialized by the inverse value $A^{-1}_{g(i)}$.



Euler Tour of T_D with initial weights

$X[1] = A[1]$
$X[2] = A[1] \cdot A[2]$
$X[3] = A[1] \cdot A[2]/A[2] \cdot A[3]$
$X[4] = A[1] \cdot A[2]/A[2] \cdot A[3] \cdot A[4]$
$X[5] = A[1] \cdot A[2]/A[2] \cdot A[3] \cdot A[4]/A[4] \cdot A[5]$
$X[6] = A[1] \cdot A[2]/A[2] \cdot A[3] \cdot A[4]/A[4] \cdot A[5]/A[5] \cdot A[6]$

The Prefixes for each vertex

3 Parallel Solution for SIR by Using Pointer Jumping

Consider for example the SIR problem $A'[2i] := A'[i+1] \cdot A'[2i]$; (for $i = 1, 2..n$) where $g(i) = 2i$ and $f(i) = i+1$. The final values of each element $A'[i]$ is a product of a varying number of items. We refer to this sequence of multiplications of every element i in $A'[]$ as the **trace** of $A'[i]$. In general the trace of each element $A'[g(i)]$ satisfies that for all $i = 1 \ldots n$: $A'[g(i)] = A[f(j_k)] \oplus \ldots \oplus A[f(j_1)] \oplus A[g(i)]$ such that:

- $j_1 = i$.
- for $t = 2 \ldots k$ the indices j_t satisfy that $j_t < j_{t-1}$ and $g(j_t) = f(j_{t-1})$.
- j_k is the last index for which $g(j_t) = f(j_{t-1})$, i.e., there is no $1 \leq j_{k+1} < j_k$ such that $g(j_{k+1}) = f(j_k)$.

The above rule suggests a simple method for computing $A'[g(i)]$ in parallel. Let $A^{-t}[g(i)]$ denote the sub-trace with $t+1$ rightmost elements in the trace of $A'[g(i)]$, i.e., $A^{-t}[g(i)] = A[f(j_{k-t})] \oplus \ldots \oplus A[f(i)] \oplus A[g(i)]$, and now consider the **concatenation** (or **multiplication**) of two "successive" sub-traces: $A^{-(t_1+t_2)}[g(i)] = A^{-t_1}[g(j)] \oplus A^{-t_2}[g(i)]$ where $g(j) = f(j_{k-t_2})$ and $A[f(j_{k-t_2})]$ is the last element in $A^{-t_2}[g(i)]$.

The proposed algorithm (as shown in [1]) is a simple greedy algorithm that keeps iterating until all traces are completed. In each iteration all possible concatenations of successive sub-traces are computed in parallel where the concatenation operation of sub-traces $A^{-t_1}[g(j)], A^{-t_2}[g(j)]$ can be implemented as follows:

1. The value of a sub-trace $A^{-t}[g(i)]$ is stored in its array element $A[g(i)]$.
2. A pointer $Next[g(i)]$ points to the sub-trace $A^{-t_1}[g(j)]$ to be concatenated to $A^{-t_2}[g(i)]$ (to form $A^{-(t_1+t_2)}[g(i)]$). Hence, $A[Next[g(i)]]$ contains the value of the sub-trace $A^{-t_1}[g(j)]$.

Following is the code of an improved and a more practical version of the algorithm for which we ran our experiments with $p << n$ processors. The new algorithm improves the above algorithm in the following points:

- It works with compressed array of size n (the number of loop iterations) instead of the original array size, which is of size $m > n$, by mapping every $A'[g(i)]$ element to an auxiliary array $X[i]$.
- The index function $f(i)$ is transformed to a decreasing function which never refers to future indexes of $X[]$. This reduces the total number of iterations required by the algorithm from an order of $\log n$ to $\log p$, improving the execution time to $\frac{n}{p} \log p$. Suppose each processor computes $\frac{n}{p}$ traces and uses a sequence of passes wherein it calculates the traces sequentially one after another. Since $f()$ always points backwards to earlier elements in the input array, then after each step i we already have the final values of the first 2^i elements, requiring a total of $\log p$ steps in order to complete the entire array of size n. The same argument however, does not hold for the case where $f()$ can refer to future elements in the array.

```
Input: Initialized Arrays A[1..m], g[1..m], f[1..m], ⊕.   (where ⊕ is a binary associative operator)

Output: Array X[1..n] where X[i] will hold the value of A[g[i]]
   at the end of the IR loop.

Initialize Auxiliary Arrays X[1..n], G[1..m], Next[1..n] to zero.

forall i ∈ {1..n} do in parallel
   G[g[i]] := i; (The auxliary array G represents the inverse of g, i.e. G[i] ≡ g^-1(i))
end parallel-for

forall i ∈ {1..n} do in parallel
   u := G[f[i]]; (The iteration where A'[f(i)] was modified by the loop).
   if (u ≥ i or u = 0) { (The trace of A'[g(i)] is of size two.)
      X[i] := A[f[i]] ⊕ A[g[i]]; (i.e., A'[g(i)] = A[f(i)] ⊕ A[g(i)].)
      Next[i] := 0;
   } else {
      if (f[u] ≥ f[i] or f[u] = 0) { (The trace of A'[f(i)] is of size three.)
         X[i] := A[f[u]] ⊕ A[f[i]] ⊕ A[g[i]];
         Next[i] := 0;
      } else { (The trace of A'[g(i)] is of size greater than three.)
         X[i] := A[f[i]] ⊕ A[g[i]];
         Next[i] := f[u];
      }
   }
end parallel-for
```

Initialization Stage

```
for t = 1 to log n do (The algorithm performs, at most, log n iterations)
   forall i ∈ {(2^(t-1) + 1)..n} do in parallel
      if (Next[i] > 0) then { (Apply the concatenation operation followed by the pointer updating)
         X[i] := X[Next[i]] ⊕ X[i];
         Next[i] := Next[Next[i]];
      }
```

The code for the iteration phases

4 Experimental Results for Message Passing Sync and Async SIR

For the message-passing version of SIR we assume that the initial values $A[1..m]$ are partitioned between the processors, with each processor holding $\frac{m}{p}$ elements.

Thus the first step is for each processor to fetch the $O(\frac{n}{p})$ elements of $A[]$ that it will use during the initialization stage. Each processor will sort the requested elements according to their processor destination, and then fetch them using p "big-messages" (i.e., using message packaging). Two variants of this algorithm have been tested:

Sync SIR: where all processors compute one iteration of the main loop synchronously.

Async SIR(k): where each processor performs l iterations of the main-loop before passing the control to the next processor. The number l is chosen randomly (every time) from the range $1 \ldots k$, where k is the parameter of the algorithm. Thus, by choosing different values of k we can change the amount of asynchronous deviation of the execution.

We simulated different numbers of processors, $p = 4, 8, 16, 32, 64, 128, 256, 512, 1024$ with $n = 500,000$. The simulation of the algorithm was made on a sequential machine, since this allowed us an exact and simple measure of the total size of generated messages (communication), the work (in units of C instructions) and the execution time (also in units of C instructions). All diagrams contain an artificial curve (e.g., $f(p) = \frac{n}{p} \cdot \log n$) for comparison. We first chose to test a SIR loop where $g(i) = i$ and $f(i) = i - 1$, as this maximizes the length of each trace and consequently the expected work. For *Sync SIR*, we already know the results: execution time approximately $\frac{n}{p} \cdot \log p$, work $n \cdot \log p$, total of $p \cdot \log p$ big messages and communication (total number of data items that have been sent) of $p \cdot \log p$. The results of the execution time in fig. 1 verify this computation and show that increasing k cost of *Async SIR*(k) can reduce the speedups significantly. Another observation regarding *Async SIR*(k) is that for a relatively small number of processors the impact of k is much larger than with a large number of processors. According to fig. 1,the difference between the various *Async SIR*(k) results, becomes constant (independent on p) for values greater than 32. This is because of the relatively small number of $A[]'s$ elements that are allocated to each processor ($\frac{n}{p} \approx p$). The same results are obtained for the work of the algorithms as described in fig. 1. The communication and number of big messages is exactly $p \cdot \log p$, as expected. These experiments were repeated for a random setting $f(i) = random(1 \ldots n)$. In general, we hoped to reach optimal performances of execution time $\frac{n}{p}$ and work $O(n)$. The execution time has improved, and *sync SIR* (see fig. 2) is now between $\frac{n}{p}$ and $\frac{n}{p} \cdot \log p$. In addition, the effect of k in *Async SIR*(k) on the execution time is reduced, and for $k = 1, 5, \sqrt{p}$ we get an execution time that is less than $\frac{n}{p} \cdot \log n$. In particular, we can see that the work (fig. 2) behaves like $O(n)$ rather than $n \log p$. The communication (fig. 2) is below n. Unlike the case $f(i) = i$, the effect of k in *Async SIR*(k) on the communication is negligible (all the results for $k = 1, 5, \sqrt{p}$ are the same). In addition, asynchronous execution seems to improve the communication compared to *sync SIR*. An asynchronous execution might, for example, cause the middle processor to advance its $Next[i]$ pointers before other processors started to work. This might reduce the communication that would have occured between

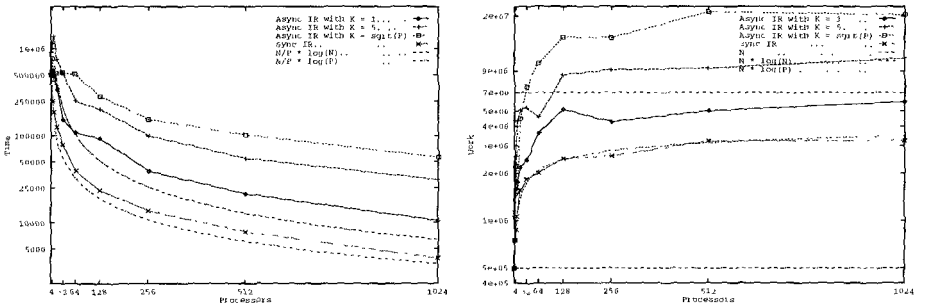


Fig. 1. *Execution time and Work for* $f(i) = i - 1$.

these processors if the middle processor hadn't advanced its pointers. The total number of messages was not affected by the random setting and (as is described in fig. 2) is around $p \log p$.

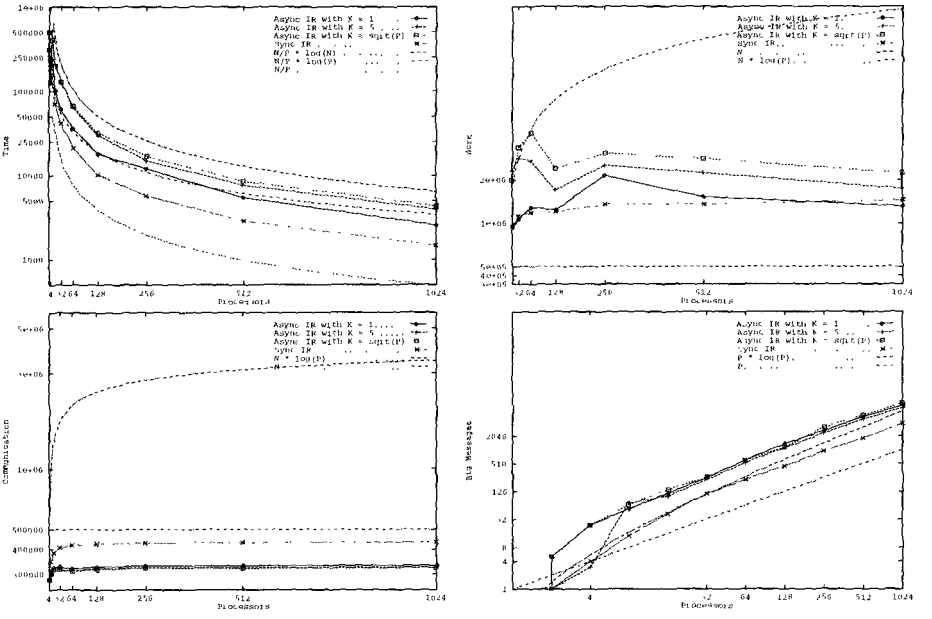


Fig. 2. *Execution time, Work, Comm. and big-messages for* $f(i) = random(1..n)$.

References

1. Y. Ben-Asher and G. Haber. Parallel solutions of indexed recurrence equations. In *In Proceedings of the IPPS'97 conference, Geneva*, 1997.

2. L. Rudolph Kruskal C. P. and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, 34(C):965–968, 1985.
3. S. Rajasekaran, S. Sen. On parallel integer sorting. Technical Report To appear in ACTA INFORMATICA, Department of Computer Science, Duke University, 1987.