Two Graph Algorithms
On an Associative Computing Model

Mingxian Jin
Department of Mathematics and Computer Science
Fayetteville State University,
Fayetteville, NC 28301
Email: mjin@uncfsu.edu

Johnnie W. Baker
Department of Computer Science
Kent State University
Kent, OH 44242
Email: jbaker@cs.kent.edu

Abstract - The MASC (for Multiple Associative Computing) model is a SIMD model enhanced with associative properties and multiple synchronous instruction streams (IS). A number of algorithms have been developed for this model and some have been implemented using its associative programming language. In this paper, we present two graph algorithms for the restricted MASC model with one IS (called the ASC model). These are an ASC version of Kruskal algorithm for the minimum spanning tree (MST) problem and an ASC version of Dijkstra algorithm for the single source shortest path (SSSP) problem. Both of our algorithms have worst case time of $O(n)$ which provide speedups of at least $n$ over their sequential versions. Also, our second algorithm is cost optimal. These algorithms have advantages of being easy to program, highly scalable and having small overhead costs. They provide a better understanding about the power of the ASC/MASC model in terms of effectiveness of algorithms it supports.

Keywords: Parallel algorithms, associative computing, SIMD algorithms, minimum spanning tree, single source shortest path

1. Introduction

The MASC (for Multiple Associative Computing) model is a SIMD model enhanced with associative properties and multiple synchronous instruction streams that can execute and coordinate data parallel threads (i.e., a multi-SIMD model). It was proposed by Potter and Baker in 1994 [16], motivated by the STARAN computer built in 1970s by Goodyear Aerospace. It has been actively studied at Kent State University as a practical model for years.

The MASC model possesses certain constant time global properties such as constant time broadcasting, constant time global reduction operations, and constant time associative search, all of which have been justified in [9]. These properties make it distinguishable from a number of other parallel models, in that it solves not only general parallel processing problems effectively but also solves problems in special areas such as real-time air traffic control in an extremely efficient manner [14]. The MASC model has been compared using simulations to other well-known parallel models [2]. A standard associative language that supports the one IS version of MASC, called ASC, has been implemented across a number of platforms. A number of MASC algorithms have been developed to solve problems in various areas [1, 6, 16].

In this paper, we present two graph algorithms for the ASC model, namely one for Kruskal algorithm for the minimum spanning tree (MST) problem, and one for Dijkstra algorithm for the single source shortest path (SSSP) problem. By taking advantages of constant time associative operations on the model, both of our ASC versions of the algorithms take $O(n)$ time, which is optimal and provides an $O(n)$ speedup when compared to their sequential versions in worst case. Since the MST and SSSP problems both have numerous practical applications, this work provides additional information about the power of the ASC and MASC model in terms of effectiveness of algorithms they support.

The paper is organized as following. Section 2 reviews the sequential algorithms for Kruskal algorithm and Dijkstra algorithm. Section 3 outlines the ASC/MASC model and its properties. Sections 4 and 5 present the ASC version of Kruskal algorithm for the MST problem and the ASC version of Dijkstra algorithm for SSSP problem. Section 6 provides a summary.

2. Problems and Their Sequential Algorithms

2.1 The MST Problem

The MST problem is described as follows. Given an weighted, undirected connected graph $G=(V, E)$ while $V$ is the set of vertices and $E$ is the set of edges. The problem is to find a tree $T=(V, E')$ such that $E' \subseteq E$ and the weight of $T$ is minimized.

Classical algorithms to solve this problem are Prim algorithm and Kruskal algorithm, as described in
An ASC version of Prim algorithm has been developed previously [16]. In this paper, we present an ASC version of Kruskal algorithm in Section 4.

As described in [5], the sequential version of Kruskal algorithm uses the disjoint-set operations to iteratively examine all edges which are pre-sorted in non-decreasing order by weight. Initially each of vertices is made a set. The algorithm selects an edge \((u, v)\) with the smallest weight and checks if it is a safe edge. A safe edge is defined to be an edge that can be safely added to the minimum spanning tree without making a cycle or breaking its property of preserving a minimum weight. If the edge \((u, v)\) is safe, the set containing \(u\) and the set containing \(v\) are merged to a larger set. If the edge is not safe, then it is discarded. This procedure is iterated until all edges are examined.

The sequential Kruskal algorithm requires sorting all edges. The running time of the algorithm is \(O(|E| \log |E|)\). Since \(|E| \leq |V|^2\), this time is equal to \(O(|E| \log |V|)\). If \(n = |V|\) and \(m = |E|\), this is \(O(m \log n)\) or \(O(n \log m)\) in worst case.

Parallel versions of this algorithm on different models have been studied. Most of them are on the PRAM model and many still require sorting edges, unless they use another method [17]. For examples, the author of [10] developed a parallel Kruskal algorithm on a CRCW PRAM in \(O(\log m)\) with \(mn^2\) processors. The authors of [18] gave another parallel version of this algorithm on CRCW PRAM in \(O(\log m)\) using \(O(m+n)\) processors. Although our algorithm takes more time, its overall cost is less than the above examples, particularly in worst case, as we use fewer processors. Also, unlike other versions of the algorithms, our algorithm eliminates a need of sorting edges due to constant time associative operations of the MASC model.

### 2.2 The SSSP Problem

The SSSP problem is described as follows. Given a weighted, directed graph \(G = (V, E)\) where \(V\) is the set of vertices and \(E\) is the set of edges. The weight of a path from vertex \(u\) to vertex \(v\) is defined to the sum of the weights of all included edges from \(u\) to \(v\). A path is a shortest path if its weight is minimal. The problem is to find a shortest path from a given source vertex \(s \in V\) to each other vertex \(v \in V\).

Dijkstra algorithm is a classical algorithm to solve this problem. As described in [5], the basic idea is the use of the greedy strategy. While the algorithm proceeds, the distance of each vertex from the source vertex \(s\) is updated. The current shortest distance from source \(s\) to vertex \(u\) is denoted to \(d(u)\). Initially \(d(u)\) for each vertex \(u\) is set to be infinite except \(d(s)\) is zero. A priority queue is used to select the next vertex \(u\) with the shortest distance to \(s\) among all of the vertices in the queue. After \(u\) is extracted from the queue, \(d(v)\) for each adjacent vertex \(v\) of \(u\) is examined. If the weight of the edge \((u, v)\) is \(w(u, v)\) and \(d(v) > d(u) + w(u, v)\) then \(d(v)\) is set to \(d(u) + w(u, v)\), which represents the updated current shortest distance from \(s\) to \(v\). The procedure is iterated until the queue is empty.

The running time of Dijkstra algorithm is \(O(|V| \log |V| + |E|)\) or \(O(n \log n + m)\) if \(n = |V|\) and \(m = |E|\). This is \(O(n^2)\) time in worst case when \(m = n^2\). However, using MASC, we do not need to maintain the priority queue to find the minimum value. Instead, the MASC associative search can find this data item in constant time. Therefore, the MASC version of Dijkstra algorithm that will be presented in Section 5 takes \(O(n)\) time in worst case.

### 3. The MASC Model

The MASC model is an enhanced multi-SIMD model with associative operations. As shown in Figure 1, MASC consists of an array of processing elements (PEs) and one or more instruction streams (ISs). A MASC with \(n\) PEs and \(j\) ISs is written as MASC\((n, j)\). Sets of PEs assigned to each IS are disjoint and form a partition of the set of all PEs. This partition is dynamic in the sense that the assignment of PEs to ISs can change during execution. Each PE, paired with its local memory, is called a cell. (In this paper, we use the terms PE and cells interchangeably when there is no conflict.) Instructed by its assigned IS, a PE acts as an ALU to perform basic arithmetic and logic operations. A PE neither stores a copy of the program nor participates in decoding this program.

![Figure 1 The MASC model](image_url)
With its broadcast/reduction network, the MASC model supports basic associative operations in constant time (assuming word length is a constant) and broadcast from an IS to its active cells in constant time. These associative operations include the global reduction operations of OR and AND, the maximum and minimum of integer or real values. In addition, the MASC model supports a constant time associative search due to the fact that data in the local memories of the PEs are located by content rather than by address. The cells whose data value matches the search pattern are called responders and the unsuccessful ones are called non-responders. The IS can activate either the responders or the non-responders. Each IS can determine if there is a responder ("any responders") and can select (or "pick one") a responder from the set of active cells in constant time. An IS can instruct the selected cell to place a value on the bus and all other cells in its set receive this value in constant time. The feasibility of these assumptions has been justified in [9] and more details can be found there.

The number of ISs is naturally smaller than the number of PEs. ISs are coordinated using control parallelism and communicate using the IS network [3]. In this paper, however, we only use the MASC model with one instruction stream.

A standard associative language that supports the one IS version of MASC, also called ASC, has been implemented across a number of platforms. It provides true portability for parallel algorithms [15]. As mentioned earlier, the MASC model is able to support general algorithm development and analysis. A wide range of different type of ASC algorithms and several large programs has been implemented using the ASC language. A few examples are given in [1, 6, 16].

The MASC model has been compared and related to that of other well-known parallel computational models such as PRAM, MMB, and RM. Some constant time simulation results have been obtained [2]. These simulations and comparisons provide methods to evaluate the power of the MASC model. On the other hand, it is also useful to develop more efficient and effective algorithms for this model, and these algorithms provide information about its power.

A tabular representation is used to store data for the MASC model. Using an associative search, data can be located by it content rather than by its address. For example, Figure 2 shows how a directed graph can be represented in tabular form with each PE storing one vertex of the graph. The five parallel variables $aS$, $bS$,
$c$, $d$, and $e$ store the weights of outgoing edges from this vertex, respectively. Figure 3 shows how an undirected graph can be represented with each PE storing one edge of the graph. The variables $u$ and $v$ represent the two vertices of the edge. Other variables will be discussed in later sections.

Note that a “$\$” sign is used as a suffix of a parallel variable in order to distinguish it from an IS variable, which is also called a scalar variable. Since only one IS is used in our algorithms, we refer to them as ASC rather than MASC algorithms. These ASC algorithms are presented in the following sections.

4. ASC Version of Kruskal Algorithm

We use the graph representation illustrated in Figure 3 as the data structure in our algorithm. That is, the information about each edge is stored on one PE. In addition to the three variables (or fields) in Figure 3 for the weight of the edge (edge$\$) and two end vertices (u$\$ and v$\$), four additional variables are defined, i.e., $setu_rep\$ and $setv_rep\$ for storing the representatives of the two disjoint-sets containing $u$ and $v$, respectively, $safe\$ for identifying a safe edge, and $visited\$ for tracking edged that have been visited.

Initially, all $setu_rep\$ and $setv_rep\$ are set to the same values as $u$ and $v$, i.e., $setu_rep\$ = $u$ and $setv_rep\$ = $v$. The algorithm follows the idea of the sequential Kruskal algorithm and selects an edge with minimum weight, but there is no need to first sort all edges in the MASC algorithm. This is because the MASC model can use its minimum field operation to find an edge with the minimum weight in constant time. When such an edge is selected, the two sets containing its two adjacent vertices are checked. If the two sets are different, the edge is safe and can be added to the resulting tree. The IS reads the values of the two set representatives, i.e., $setu_rep\$ or $setv_rep\$. The numerically or lexicographically smaller (i.e., smaller using dictionary ordering) representative is stored in the scalar variable new$\_rep$, and the other representative is stored in old$\_rep$. Also, $visited\$ is set to “Yes” for this edge. The IS instructs all PEs to replace all the value of old$\_rep$ with the value of new$\_rep$ in their fields $setu_rep\$ and $setv_rep\$. After updating, all PEs with $setu_rep\$ equal to $setv_rep\$ are deactivated and will not participate in future processing. The above steps are iterated until all PEs are inactive. At the last step, all PEs with $visited\$ = “Yes” represent the edges of a minimum spanning tree.

Algorithm ASC-KRUSKAL-MST (G, T)

1. Set all PEs to active
2. Set $setu_rep\$ = $u$, $setv_rep\$ = $v$, $safe\$ = ‘No’, and $visited\$ = ‘No’
3. For all PEs, repeat the following steps until no PE is active
   3.1. Find the minimum edge$\$  with $visited\$ = ‘No’
   3.2. Pick one PE from responder(s) (if there are multiple responders)
   3.3. Set $safe\$ = ‘Yes’, set $visited\$ = ‘Yes’
   3.4. Let $old_rep\$ = Max($setu_rep\$, $setv_rep\$) and $new_rep\$ = Min($setu_rep\$, $setv_rep\$)
   3.5. Update $setu_rep\$ and $setv_rep\$ for all PEs by replacing the value of old$\_rep$ with the value of new$\_rep$
   3.6. Deactivate all PEs with $setu_rep\$ = $setv_rep\$

End.

The correctness of the algorithm can be justified as follows. Initially each vertex makes an individual set. Each round of iteration examines an edge with the minimum weight if its two adjacent vertices are in two different sets. If not, it is a safe edge. Then, we merge the two sets and set the new set representative to the vertex that has a smaller number (or lexicographically smaller). All occurrences of the old set representatives in the $setu_rep\$ and $setv_rep\$ fields are updated with the value of this new set representative. At the end of the iteration, all PEs with edges having two adjacent vertices in the same set are deactivated and do not participate future processing. This avoids unnecessary processing. Figure 3 shows variables after two iterations.

Since each iteration takes constant time, the running time of the above algorithm is $O(|V|)$ or $O(n)$ if $|V| = n$. It represents a speedup of $O(m \log n)$ compared to the sequential version of the algorithm if $|E| = m$. In worst case, this is $O(n \log n)$, which is substantial. Moreover, we use $m$ PEs to store edges while one edge per PE. They can be easily scaled to a large data set without any additional overhead costs.

5. ASC Version of Dijkstra Algorithm

We slightly modify the graph representation in Figure 2 for the data structure used in our algorithm. The vertices are still stored one per PE. Instead of storing data of its outgoing edges, we store data of its incoming edges. The modified data structure is shown in Figure 4. On each PE, in addition to the $n$ fields to represent the graph (five fields in Figure 4), three
variables are defined. They are, dist$ for the current shortest distance from the source, parent$ for the parent (or predecessor) vertex in the shortest path, and visited$ for tracking vertices that have been visited. On the IS, three scalar variables are defined, i.e., source for the source vertex, $u$ for the vertex currently being processed, and dist$_u$ for the current shortest distance from s to $u$.

The algorithm initially sets all PEs but the source with dist$ to the value in the source’s field, parent$ to the source if dist$ is not infinite otherwise to $\emptyset$, and visited$ to “No”. For the source vertex, set dist$ to 0, parent$ to $\emptyset$ and visited$ to “Yes”. Then a loop is executed until all vertices have been visited. During each round of iteration, find a vertex with the minimum dist$ and set it visited to “Yes”. The IS reads the vertex to the scalar variable $u$ and its dist$ value to another scalar variable dist$_u$. Then the IS broadcasts these two values to all other PEs. All PEs with visited$ = “No” check if its dist$ > dist$_u + the value in u’s field. If yes, update dist$ with dist$_u + the value in the u’s field and its parent$ is set to u. The loop continues until no more vertex with visited = “No”. It can be described as follows.

**Algorithm ASC-DIJKSTRA-SSSP(G, s)**

**Begin**
1. Set all PEs active
2. Set visited$ = “No”
3. Set dist$ to the value in the source’s field
4. Set parent$ to source if dist$ is not infinite otherwise to $\emptyset$
5. Set visited$ of source to “Yes”
6. Repeat the following steps until no PE remains with visited$ = “No”
   6.1. Find the minimum dist$ with visited$ = “No”
   6.2. Pick one PE from the responder(s) (if there are multiple responders)
   6.3. Read $u$ and dist$_u$ from vertex$ and dist$ of the PE picked in the above step
   6.4. Set visited$ of $u$ to “Yes”

**End.**

The correctness of the algorithm can be justified as follows. Starting from the source s, we select a vertex $u$ with the shortest distance dist$_u$ to s. Then we compare the current shortest distance dist$ for each of its adjacent vertices with the distance if the path would go through $u$ (i.e., dist$_u + the weight of the adjacent edge). The current shortest distances of its adjacent vertices are updated accordingly. Inductively, assume that during the $k$th iteration, the shortest path from the source to each vertex is found, subject to the restriction that the path can only go through the $k$ closest vertices to the source. This greedy strategy used in this algorithm assures the shortest path from the source to each vertex is found, subject to the restriction that the path can go through either the initial $k+1$ closest points to the source. This assures that for $k = n-1$, the shortest path from each vertex to the source is found since this path is allowed to go through any of the other points.

Since each iteration takes constant time, the total run time of the above algorithm is $O(n)$ time if $n = |V|$. This represents a speedup of $O(\log n + m/n)$ over the sequential version of Dijkstra algorithm if $m = |E|$. In worst case when $m = n^2$, the speedup is $n$, which is linear. Moreover, this algorithm is cost optimal since the optimal running time for the sequential algorithm in the worst case is $n^2$.

Parallel versions of Dijkstra algorithm have also been studied by other researchers. However, in the literature we have searched, they did not provide a clear complexity analysis (see examples in [12, 17]). In contrast, the complexity of our algorithm is clear. Similarly, our algorithm uses $n$ PEs, which is cost
optimal. It also has advantages of easy programming and high scalability.

6. Summarizing Remarks

We have presented two graph algorithms for the ASC model, i.e., the ASC version of Kruskal algorithm for the MST problem and the ASC version of Dijkstra algorithm for the SSSP problem. Both of them take \(O(n)\) time. Both have achieved a significant speedup of at least a factor of \(n\) in the worst case, compared with their sequential versions. Also, our ASC version of the Dijsktra algorithm is cost optimal. The results can be summarized in Table 1, along with the results of similar research work on other models.

<table>
<thead>
<tr>
<th>Table 1 Sequential version vs. ASC version of the algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of processors</td>
</tr>
<tr>
<td>Sequential Kruskal Algo.</td>
</tr>
<tr>
<td>ASC Kruskal Algo.</td>
</tr>
<tr>
<td>CRCW PRAM Kruskal Algo. in [10]</td>
</tr>
<tr>
<td>CRCW PRAM Kruskal Algo. in [18]</td>
</tr>
<tr>
<td>Sequential Dijkstra Algo.</td>
</tr>
<tr>
<td>ASC Dijkstra Algo</td>
</tr>
</tbody>
</table>

In addition to the speedups gained, the ASC algorithms have advantages of easy programming, highly scalability, and having small overhead costs. By taking advantages of the constant time associative operations on the ASC model, we eliminate a need of sorting edges in the MST algorithm. These two algorithms can also be efficiently implemented on other types of SIMD computers, though the running time may increase depending on the efficiency with which the SIMD can support the additional basic operations assumed for ASC.

Our work provides a better understanding of the ASC/MASC model in terms of effectiveness of algorithms it supports. It also contributes more ASC/MASC versions of classical algorithms to the collection of algorithms for this model.

Further interesting work includes using multiple ISs to see how much the running time for these algorithms can be improved. A possible direction for future work is to implement the above algorithms on an ASC simulator using the ASC language, on a SIMD, or a COTS SIMD (e.g., WorldScape’s CSX601). This would provide data to further evaluate the effectiveness of these algorithms.

References


