A Linear-C Implementation of Dijkstra's Algorithm

Chung-Hsing Hsu and Donald Smith and Saul Levy
Department of Computer Science
Rutgers University
LCSR-TR-274

October 9, 1996

Abstract

Linear-C is a data-parallel extension to C. In this report we show, by implementing Dijkstra's algorithm in Linear-C to solve the shortest-paths problem, that (1) data-parallelism in Dijkstra's algorithm can be easily expressed in Linear-C, (2) even a small amount of data parallelism can speed up the whole algorithm substantially, and (3) the algorithm, the data representation, and the efficiency are closely inter-related. Three implementations are provided, each with a different level of data parallelism exploited. The programs are explained, their time complexities are analyzed, and their strength and weakness in efficiency are compared.

1 Introduction

Linear-C [1, 2] is a data-parallel extension to C. It extends C by treating arrays (or more precisely, aggregate values) as first-class citizens. Furthermore, several kinds of association are attached to arrays to control the computation. Activity association, for example, selects particular elements of an array involved in a computation. In addition, Linear-C provides a set of unary operators to manipulate aggregate values. Linear-C has

- a concise representation of programs,
- an explicit data-parallel operation,
- an efficient executables, [1] and
- is easy to debug. [2]

This report describes how the single-source shortest-paths problem can be solved by implementing Dijkstra's algorithm in Linear-C. Various versions of implementation are provided; each of them is an improvement over the other in efficiency. The basic idea is to express data-parallelism

[1] As a direct support for Linear-C, we have developed a new type of memory chip, the CAM2000 [3], that provides efficient ways to compute data-parallel operations.

[2] The property of imperative substitutives allows the C-debugging tools to be used. See Section 3 for details.

exploited in Dijkstra's algorithm as much as possible. The different versions also provide the insight into the inter-relationship between data representation, algorithm, and efficiency.

The report is organized as follows. Section 2 formulates the shortest-path problem, describes the data-parallel Dijkstra's algorithm, and gives an overview of three versions of implementation: one written in C and two written in Linear-C. After that, a subset of Linear-C statements is introduced in Section 3 as a background to interpret the Linear-C programs. We also specialize Dijkstra's algorithm as a common structure for all three implementations in Section 4 to show that even a little data parallelism can substantially improve efficiency. In Sections 5, 6, and 7, those programs are interpreted separately and their corresponding time complexities are discussed. Efficiency concerns are examined again in Section 8, and the inter-relationship between data representation, data manipulation, and efficiency is identified. Section 9 concludes the report in a summary.

2 The Shortest-Paths Problem, Dijkstra's Algorithm, and Its Implementation

The single-source shortest-paths problem is formulated as follows:

> Given an edge-weighted, directed graph $G = (\mathcal{V}, \mathcal{E}, \mathcal{W})$ where
> $\mathcal{V} = \{v_i | 0 \leq i < N\}$, the set of N vertices,
> $\mathcal{E} = \{e_j = (v_i, v_k) | 0 \leq j < E\}$, the set of E edges, and
> $\mathcal{W} : E \rightarrow Z^+$, weight function over edges
> find out the shortest-path length from v_0 to every vertex $v_i \in V$, recorded as MINPL(v_i).

Dijkstra's algorithm is one of the well-known algorithms for solving this problem. It is described in Figure 1. Here, NEWPL(v_i) maintains the upper bound of path length from v_0 to v_i, and EXPAND(v_i) records whether or not v_i has been expanded. Conceptually, Dijkstra's algorithm iteratively refines NEWPL(v_i) according to the new information. In each iteration, a set of vertices, *nearest*, is expanded, and once they are expanded, their NEWPL values, all equal to *nearest_pl*, cannot be refined any further. It is this new information which makes refining NEWPL's of other unexpanded vertices possible.

Statements 1-6 are a sequence of initialization steps. Statements 8-14 are the work to be done in each iteration. Statement 8 marks all vertices in *nearest* to be expanded so that they won't be expanded again in the future. Statements 9-12 refines the NEWPL's of other vertices. Dijkstra's algorithm is smart enough to update only part of unexpanded vertices, those adjacent to vertices of *nearest*. This partial graph traversal makes the implementation more efficient. After updating NEWPL's, Dijkstra's algorithm has to pick up the next set of vertices to be expanded. This is done in statements 13 and 14. Statement 13 finds the minimum value of NEWPL's among unexpanded vertices. If all vertices are expanded already, *nearest_pl* will be reset to ∞ so as to exit the iteration loop. Statement 14 finds the set *nearest* according to this *nearest_pl* information. After statement 14 iterations continue until all vertices are expanded. When all vertices are expanded, statement 16 turns the result into the canonical form, namely, MINPL(v_i). This process is made explicit to

```
1   set up the graph.
2   initialize NEWPL(v_i) ← ∞ for all v_i ∈ V.
3   initialize EXPAND(v_i) ← no for all v_i ∈ V.
4   NEWPL(v_0) ← 0.
5   nearest_pl ← 0.
6   nearest ← {v_0}.
7   while (nearest_pl ≠ ∞) do
8       EXPAND(v_i) ← yes for all v_i ∈ nearest.
9       for {v_i|EXPAND(v_i) = no && e_j = (v_k, v_i) ∈ E where v_k ∈ nearest}
10          NEWPL(v_i) ← min{NEWPL(v_i), nearest_pl + W(e_j)}.
12      end for.
13      nearest_pl ← min{NEWPL(v_i)|EXPAND(v_i) = no}.
14      nearest ← {v_i|EXPAND(v_i) = no && NEWPL(v_i) = nearest_pl}.
15  end while.
16  MINPL(v_i) ← NEWPL(v_i) for all v_i ∈ V.
```

Figure 1: Dijkstra's algorithm

enforce the consideration of time complexity of other result forms. As we will see later, one of our implementations generates another result form.

The description in Figure 1 makes explicit several instances of data parallelism in Dijkstra's algorithm. First of all, a set vertices can be expanded at the same time. Secondly, refining NEWPL's of other vertices can be done in parallel. Finding the *nearest_pl* and the corresponding *nearest* can also be done more efficiently. Finally, copying NEWPL(v_i) to MINPL(v_i) can be done in parallel for all v_i. Though the algorithm reveals much data parallelism, it still depends on the machine model to be fully implemented. Ideally, the most powerful machine model for this algorithm will make the running time $O(N)$, the number of iterations.

Linear-C is a data parallel extension to C. It can perform the operation on all elements of an array in parallel. In C, only one operation on an object is allowed at a time. Therefore, to perform the same operation on all objects of an array, C has to iterate over all objects, doing operations one by one. The time of completion is equal to the size of the array. In contrast, Linear-C can perform the same operation on all objects of an array in constant time. If we have a fixed number of operations on objects to be done, the more objects operated in parallel, the less time for the whole computation.

Three versions of implementation are provided: C-code, LC-N-code, and LC-E-code. C-code is written in C and is thus expected to be the slowest. LC-N-code is written in Linear-C as an array of vertices. LC-E-code is written also in Linear-C but as an array of edges. Usually a graph contains more edges than vertices, i.e., $E > N$, which implies that, LC-N-code is expected to be faster than C-code, but slower than LC-E-code. Figure 7 in page 14 gives the time complexity of these three versions and verifies the above expectation.

3 Linear-C: A Subset

An array is a variable in which multiple scalar values are stored. In C, a scalar value in an array is retrieved by indexing. In contrast, Linear-C considers all these values of an array as an *aggregate value*, without indexing involved. In short, Linear-C extends C by treating aggregate values as first class values.

The operational semantics of a Linear-C statement can be described in terms of a sequence of C statements, which we term *C-simulation*.

Linear-C	C-simulation
a Linear-C statement	{ a block of standard C statements }

C-simulation is only a simulation because it imposes the unnecessary sequential nature on the semantics of a Linear-C statement. In other words, an operation on the entire array is simulated by the iteration over all elements of an array, through indexing. On the other hand, a Linear-C statement and its corresponding C-simulation is consistent on the state before and after the statement is executed. This property is called *imperative substitutive*, in the sense that a Linear-C statement and its C-simulation can be substituted for each other without changing the state semantics of a program. Note that, a Linear-C statement may have many different C-simulations.

In the following, we will take a look at different types of Linear-C expressions and statements used in LC-N-code and LC-E-code. Note that, Linear-C extends C's conditional operator ? to control the activity of a computation. For example, an array assignment is considered as a set of scalar assignments in C's view. But Linear-C is capable of selecting (or activating) a subset of these assignments to be performed. Linear-C calls this feature *activity control*. Note also that Linear-C introduces a new operator /\. This operator is used for finding the minimum scalar of an aggregate value [3].

Syntax	Intention
s	a scalar
A	the target array to be assigned into
B	the source array involved in a computation
ACT	the *activity* array controlling a computation
$\oplus$	any binary operator in C, e.g., +
A = s	assign s into all elements of A
A = B	assign each element of B into corresponding element of A
A = B $\oplus$ s	assign the result of B $\oplus$ s into A
A = (ACT ? B)	assign *part* of B into A selected by ACT
/\ B	find the minimum scalar from B
/\ (ACT ? B)	find the minimum from *part* of B, selected by ACT

All these operations are considered primitive in Linear-C, and each of them can be done in constant time on a CAM2000. Operations can also be combined into more complex expressions, evaluated still in constant time. The whole picture of Linear-C is explored in [2].

[3] The order of scalars are as in C.

Statement A = s means broadcast the value of scalar s to every element of array A. Or, for every element of array A, get the value of scalar s. Therefore we call it *broadcasting* semantics or *for-all* semantics.

Syntax	Intention	C-simulation
A = s;	{ A[i] = s; | for all i }	{ int i; for (i = 0; i < size(A); i++) A[i] = s; }

In contrast, statement A = B means get the aggregate value of array B and assign it into the corresponding position of array A. Or, for each element of array A, get the value of its corresponding element of array B. Therefore it is called *corresponding* semantics or *for-each* semantics. Here, correspondence means two elements having the same index. In addition, this statement is performed only when the two arrays are of the same size. [4]

Syntax	Intention	C-Simulation
A = B;	{ A[i] = B[i]; | for all i }	{ int i; for (i = 0; i < size(A); i++) A[i] = B[i]; }

The interpretation of statement A = B $\oplus$ s is similar: compute the resulting aggregate value from expression B $\oplus$ s and assign it into array A. Note that we call expression B $\oplus$ s *mixed-mode* since it combines an aggregate value and a scalar value.

Syntax	Intention	C-simulation
A = B $\oplus$ s;	{ A[i] = B[i] $\oplus$ s; | for all i }	{ int i; for (i = 0; i < size(A); i++) A[i] = B[i] $\oplus$ s; }

An array assignment can be done partially. Statement A = (ACT ? B) means associate the value of ACT as activity control with the aggregate value of B, and then assign this activity-controlled aggregate value into array A. As a result, only those scalar assignments in A = B activated by the corresponding scalar of array ACT will be performed. If all elements of ACT are non-zeros, the statement A = (ACT ? B) is exactly the same as A = B. On the other hand, if all elements are zeros, it means none of the scalar assignments is active, and the whole statement is just an empty statement, no effect at all.

Syntax	Intention	C-simulation

A = (ACT ? B);	{ A[i] = B[i]; | ACT[i] $\neq$ 0 for all i }	{ int i;
		for (i = 0; i < size(A); i++)
		if (ACT[i]) A[i] = B[i]; }

[4] Assignment between arrays of different sizes complicates the operational semantics of the language and produces anomalies. It is not allowed in Linear-C. See [2, Section 2.5] for more details.

Besides extending C operators to deal with aggregate values, Linear-C provides a set of new unary operators for aggregate values. Operator /\ is one of them which finds the minimum scalar of an aggregate value. Therefore, statement s = /\ B means find the minimum scalar from the aggregate value of B and assign it into scalar s.

Syntax	Intention	C-simulation

s = /\ B;	min{ B[i] | for all i }	{ int i;
		int minval = Inf;
		for (i = 0; i < size(B); i++)
		if (B[i] < minval)
		minval = B[i];
		s = minval; }

By combining activity control and minimum-finding operator, we are able to find the minimum scalar from a *part* of the aggregate value. Statement s = /\ (ACT ? B), therefore, is interpreted as: associate the value of ACT as activity control with the value of B, and then find the minimum scalar from this activity-controlled value, and finally assign it into s. As a result, the minimum-finding operator only operates on those elements of B activated by ACT, and assign the result into scalar s. If none of the elements is active, scalar s won't be modified.

Syntax	Intention	C-simulation

s = /\ (ACT ? B);	min{ B[i] | ACT[i] $\neq$ 0 for all i }	{ int i;
		int ac = 0, minval = Inf;
		for (i = 0; i < size(B); i++)
		if (ACT[i]) {
		ac = 1;
		if (B[i] < minval)
		minval = B[i]; }
		if (ac)
		s = minval; }

4 Specialization of Dijkstra's Algorithm

Dijkstra's algorithm, shown in Figure 1, exploits much data parallelism. Unfortunately, none of it can be implemented in C since C is a sequential language. To be fair in comparison with C-code, or even in favor of it, we restrict Dijkstra's algorithm in several ways. This specialized version of the algorithm is then used as a common structure for all three programs: C-code, LC-N-code, and LC-E-code. As a result, LC-N-code and LC-E-code do not fully utilize the capability of Linear-C in implementing all data parallelism of Dijkstra's algorithm. Nonetheless, they show that even a few statements implemented in data parallel will speed up the whole algorithm substantially.

There are four major guidelines in this specialized Dijkstra's algorithm.

```
1  set up the graph.
2  initialize NEWPL[i] ← ∞ for all v_i ∈ V.
3  initialize EXPAND[i] ← no for all v_i ∈ V.
4  NEWPL[0] ← 0.
5  nearest_pl ← 0.
6  nearest ← 0.
7  while (nearest_pl ≠ ∞) do
8      EXPAND[nearest] ← yes.
9      for {i | EXPAND[i] = no && e_j = (nearest, i) ∈ E}
10         if NEWPL[i] > nearest_pl + W(e_j)
11             NEWPL[i] ← nearest_pl + W(e_j).
12     end for.
13     nearest_pl ← min{NEWPL[i] | EXPAND[i] = no}.
14     nearest ← min{i | EXPAND[i] = no && NEWPL[i] = nearest_pl}.
15 end while.
16 MINPL[i] ← NEWPL[i] for all v_i ∈ V.
```

Figure 2: Dijkstra's algorithm, index as vertex

1. the set *nearest* is restricted to be a singleton. In other words, only one vertex is expanded in each iteration. As a consequence, when several vertices are qualified for *nearest*, the one with the smallest index is always chosen.

2. NEWPL refinement in statement 10 is implemented as a conditional assignment.

3. since v_i can be identified by its index i, we use the indices of vertices to represent vertices themselves.

4. NEWPL, EXPAND, and MINPL are all implemented as arrays. i.e.,

$$\text{NEWPL}(v_i) \stackrel{\text{def}}{=} \text{NEWPL[i]}$$

.

The specialized version of Dijkstra's algorithm is shown in Figure 2.

Before closing this section, we would like to rephrase the three time-consuming steps in each iteration of the specialized Dijkstra's algorithm. They are,

1. statements 9-12: update NEWPL's of a subset of unexpanded vertices,

2. statement 13: find the minimum of NEWPL's of those vertices,

3. statement 14: find an unexpanded vertex with that minimum.

5 Interpretation of C-code

The specialized version of Dijkstra's algorithm in Figure 2 has three time-consuming steps in each iteration. When NEWPL and EXPAND are implemented as arrays, statements 13 and 14 can only be done as a sequential search over these arrays, which results in $O(N)$ time. But for statements 9-12, if the graph is represented "right", then finding the particular subset of unexpanded vertices (or say, partial graph traversal) can be done efficiently. C-code uses the *adjacency list* as the graph representation, substantially reducing the total time for all graph traversal from naive $O(N \cdot E)$ [5] to $O(E)$.

The adjacency list representation considers a graph as a collection of the adjacency lists of graph vertices. Every vertex v has an adjacency list $Adj(v)$ that includes all vertices adjacent from v.

$$G \stackrel{\text{def}}{=} \{Adj(v) | v \in \mathcal{V}\}, \text{ where } Adj(v) = \{(v_i, \mathcal{W}(v, v_i) | (v, v_i \in \mathcal{E}\}$$

Figure 5 gives the adjacency list representation over the example graph Figure 4.

If we look carefully, statement 9 asks exactly those unexpanded vertices in $Adj(nearest)$. Rather than search all graph edges, the adjacency list representation organizes the graph in such a way that partial search is enough. It is this partial search which makes C-code efficient in sequential world.

The statements 9-12 of C-code are interpreted as follows:

(1) iterate over elements of $Adj(nearest)$ one by one, say,

$$p = (v_i, \mathcal{W}(nearest, v_i)) \in Adj(nearest)$$

(2) if v_i is unexpanded, then update its NEWPL if possible.

Since each edge will be examined once in the whole program, the total time spent here is $O(E)$.

Statement 13 of C-code iterates over elements of array NEWPL and keeps the current minimum in *nearest_pl* all the time. When iteration ends, the minimum of NEWPL will be in *nearest_pl* already. Statement 14 does the iteration backwardly so as to get the smallest index. Both are very straightforward implementations and take $O(N)$. Statement 16 is an element-to-element copy between two arrays. It takes $O(N)$.

Since each vertex will be expanded once, the number of iteration is $O(N)$, and the whole algorithm takes $O(E + N^2)$. As we can see, the bottleneck is statements 13 and 14, finding *nearest_pl* and *nearest*. LC-N-code reduces this bottleneck to $O(N)$, since each finding takes only constant time.

6 Interpretation of LC-N-code

The LC-N-code is almost the same as the C-code, except that statements 13, 14, and 16 are turned into Linear-C statements. The algorithm is sped up by these substitutives from $O(E + N^2)$ to $O(E + N)$. Therefore, in this section we will focus on these three statements.

[5] Use edge set representation instead. Edge set representation is discussed in Section 7.

Statement 13 intends to find the *nearest_pl* among a subset of NEWPL values. Since Linear-C has the capability of finding the minimum among a subset of array elements, i.e., /\(ACT ? B), statement 13 can be implemented directly in this style. And the time to compute this partial minimum and assign it into *nearest_pl* is $O(1)$.

Statement 14 has to be done indirectly. First, we need an auxiliary array to represent the indices i. And then we apply the same technique as statement 13 to get *nearest*. This auxiliary is called NAME in LC-N-code. It is initialized as B[i] = i in statement (*) of Figure 9. The total time for statement (*) and 14 is $O(N)$.

Statement 14 raises the general question, can we find the *index* of the array element which is the minimum among all array elements? For C-code it is easy since iteration over indices of array elements records the current index. On the other hand, Linear-C treats the whole array as a pure aggregate value, without recording the index of each element. It is this loss of indexing information which makes it hard for Linear-C to answer the above question.

Statement 16 is simply an array-to-array assignment. Linear-C provides this style of statement and it is executed in $O(1)$ time. And the total time complexity of LC-N-code is $O(E + N)$. Now the bottleneck shifts to the partial graph traversal. In the next section we will present another version of Dijkstra's algorithm to reduce it from $O(E)$ to $O(N)$ and makes the running time $O(N)$.

7 Interpretation of LC-E-code

As pointed out in the previous section, partial graph traversal in statements 9-12 becomes the most time-consuming step in LC-N-code. To make this traversal faster, we use another graph representation since the adjacency list representation imposes the sequential nature on the traversal. In other words, we abandon *list*; instead, we use *array*. The graph representation used by LC-E-code is called *edge set* representation. We will see later on that it successfully reduces the total time for graph traversal from $O(E)$ to $O(N)$. Therefore, the main theme of this section is the edge set representation and its impact on the specialized Dijkstra's algorithm.

The edge set representation considers a graph as a set of weighted edges.

$$G \stackrel{\text{def}}{=} \{Edge(e) | e \in \mathcal{E}\}, \text{ where } Edge(e) = (v_i, v_j, \mathcal{W}(v_i, v_j)).$$

Every element in this set identifies a distinguished edge. Unlike the adjacency list representation, there is no relation between edges. Therefore, it is thought to be more naive than the adjacency list representation. Figure 6 shows the edge set representation of Figure 4.

If we adopt the same guidelines described in Section 4, we have to modify the specialized Dijkstra's algorithm slightly to preserve the property that *each index identifies an edge*. In other words, instead of defining NEWPL and EXPAND over vertices, we define them over edges. That is, NEWPL(e_j) gives the upper bound of path length from v_0 to (including) e_j. The algorithm is in spirit the same: find *nearest_pl* and *nearest*, refine NEWPL(e_j) to a lower value if possible, and iterate this process until no more vertices can be expanded.

Since NEWPL is defined over edges, when the algorithm ends, all edges incoming to a vertex contain different NEWPL values. To put the solution in canonical form, we perform the following action (line 16 in Figure 3):

1 set up the graph.
2 initialize NEWPL(e_j) $\leftarrow \infty$ for all $e_j \in \mathcal{E}$.
3 initialize EXPAND(e_j) $\leftarrow$ *no* for all $e_j \in \mathcal{E}$.
5 *nearest_pl* $\leftarrow 0$.
6 *nearest* $\leftarrow v_0$.
7 while (*nearest_pl* $\neq \infty$) do
8 EXPAND(e_j) $\leftarrow$ *yes* for all $e_j = (-, nearest) \in \mathcal{E}$.
9 for $\{e_j \mid$ EXPAND(e_j) = *no* && $e_j = (nearest, -) \in \mathcal{E}\}$
10 if NEWPL(e_j) > *nearest_pl* + $\mathcal{W}(e_j)$
11 NEWPL(e_j) $\leftarrow$ *nearest_pl* + $\mathcal{W}(e_j)$.
12 end for.
13 *nearest_pl* $\leftarrow \min\{\text{NEWPL}(e_j) \mid \text{EXPAND}(e_j) = no\}$.
14 *nearest* $\leftarrow \min \left\{ v_i \;\middle|\; \begin{array}{l} \text{EXPAND}(e_j) = no \;\&\&\; \text{NEWPL}(e_j) = nearest\_pl \\ \&\&\; e_j = (v_k, v_i) \end{array} \right\}$.
15 end while.
16 MINPL(v_i) $\leftarrow \min\{\text{NEWPL}(e_j) \mid e_j = (v_k, v_i) \in \mathcal{E}\}$ for all $v_i \in \mathcal{V}$.

Figure 3: Dijkstra's algorithm, index as edge

$$\text{MINPL}(v_i) \leftarrow \min\{\text{NEWPL}(e_j) | e_j = (v_k, v_i) \in \mathcal{E}\} \text{ for all } v_i \in \mathcal{V}$$

The whole algorithm is presented in Figure 3.

Statements 9-12 of LC-E-code essentially update NEWPL's of those unexpanded edges which are outgoing from *nearest*. Therefore, we can use Linear-C conditional assignment A = (ACT ? B) to implement these steps directly in constant time. Since there are $O(N)$ iterations, the total time for this graph traversal is $O(N)$.

Statement 16 states that, for each vertex, take the minimum NEWPL among all its incoming edges. A naive way to implement it, as shown in LC-E-code, is to iterate over all vertices and in each iteration a minimum-finding is done. Clearly, the time for this step is $O(N)$.

Note that, in statement 8, instead of marking *nearest* expanded, we mark all incoming edges to *nearest* expanded. These are two other important differences: several edges are marked simultaneously, and the final result of NEWPL is merely an upper bound since some of the edges may be marked earlier.

Finally, this implementation runs in $O(N + N)$, which is faster than the C-code and the LC-N-code. In the next section, some efficiency issues will be considered. Though not fully utilizing all data parallelism in the original Dijkstra's algorithm, LC-E-code is the best of the three.

8 Efficiency Concerns

There are several factors involved in determining the efficiency of an implementation of Dijkstra's algorithm. The first one is the graph representation, which has impact on the time for (partial) graph traversal. The second factor is the representation of unexpanded vertices. This representation determines the efficiency of finding the minimum among unexpanded vertices. The algorithm itself is also a contributing factor to the efficiency. An inherently sequential version of the algorithm makes the data-parallel language such as Linear-C no more powerful than C. In the following we will examine these issues in more detail.

To process graphs, we have to select a representation for them in the code. Two alternatives are used in C-code, LC-N-code, and LC-E-code: the edge set representation and the adjacency list representation. Both representations are space-efficient in the sense that they are the most economical for sparse graphs. In Dijkstra's algorithm, only adjacency list information is needed. It makes the adjacency list representation more appropriate intuitively. On the other hand, its list property has the inherent sequential nature, which makes data-parallelism hard to achieve.

If we define "naive" as containing least information, then the edge set representation is more naive than the adjacency list representation, since the adjacency list collects outgoing edges from a vertex together while edge set does not keep this information. As a consequence, a preprocessing time may be needed for a less naive representation such as adjacency list. In our time complexity analysis, we ignore this time and as our results therefore favor the C-code.

We can see in the following how graph representations determine part of the efficiency. If we use the algorithm in Figure 2 and adopt an edge set as graph representation, then the total cost for statements 9-12 are $O(N\ cdotE)$. If we adopt the adjacency list representation, then the cost is reduced to $O(E)$ since most of the unnecessary edge searches are avoided.

Another determining factor is the representation of unexpanded vertices. In C-code and LC-N-code, an array is used as the representation. Again, this is a naive representation since there is no relationship between vertices. And this naiveness results in $O(N)$ time for a sequential search of *neartes_pl* and *nearest*. A less naive representation is the *priority queue*. Priority queue representation collects unexpanded vertices as a binary search tree dynamically. As a result, the total time for statements 13, 14 is reduced from $O(N^2)$ to $O(N \log N)$.

There is an efficiency-oriented history along the way from C-code to LC-E-code. At the beginning, algorithm Figure 2, edge set as the graph, and arrays as the unexpanded vertices are used. The total time complexity is $O(NE + N^2)$ and the bottleneck is right at the graph traversal. The C-code utilizes an adjacency list to break the bottleneck. It successfully reduces the total time to $O(E + N^2)$, and the bottleneck becomes finding the minimum among unexpanded vertices. Now we have two choices. Choosing on priority queue for unexpanded vertices can reduce the total time to $O(E + N \log N)$. To use Linear-C to find the minimum in constant time reduces the total cost to $O(E + N)$. Linear-C requires that unexpanded vertices be implemented as arrays, and the bottleneck is now back to the graph traversal. Since adjacency list sequentializes the graph traversal, to break it and utilize Linear-C, we have to choose edge set implemented as arrays. Furthermore, the algorithm itself unnecessarily restricts the possible implementations. We have to modify the algorithm a little bit to fit our representation settings. As a result, algorithm in Figure 3 emerges,

and the whole time is reduced to $O(N + N)$.

As an aside, Linear-C provides more than one way to implement statement (*) of LC-N-code and statement 16 of LC-E-code. For statement (*), a partial sum operation is enough and can be done in constant time. For statement 16, segment edges according to their TO fields and then find the minimum of this segmented array yields the same result as MINPL. This also can be done in constant time. Both are improvements over the original $O(N)$ time.

Finally, as a conclusion, the "best" algorithm and representation in sequential world may not be the best choice in data-parallel world. In a data-parallel world, numerous simple data parallelism will make the algorithm the most efficient. Nonetheless, data-parallelism can make even a sequential algorithm run substantially faster.

9 Summary

In this report, we illustrate how Linear-C is used to implement (data-parallel) Dijkstra's algorithm for solving the shortest-paths problem. Three versions of the implementation are provided. Each expresses a different degree of data parallelism and has a different time complexity. These programs are explained, compared, and discussed. The inter-relationship between the algorithm, the data representation, and the efficiency is also identified.

Intuitively, the more data-parallelism is exploited, the more efficient the implementation. But the data representation, or even the algorithm itself, will be affected and modified so as to exploit the data parallelism clearly. As a result, the "best" algorithm and data representation in sequential world may not be the best choice in data-parallel world. Nonetheless, a small amount of data-parallelism will make the sequential algorithm run substantially faster.

Data parallelism can be easily expressed in Linear-C. In this report several distinct features of Linear-C are illustrated. Among them are the aggregate value, reduction operations, for-all/for-each semantics, and activity control. An aggregate value behaves like an ordinary scalar, it can be "summarized" (e.g., minimum-finding), and it can be activity-controlled. All these features are easy to learn, to use, and help to represent data parallelism easily. Furthermore, all Linear-C primitives are directly supported by the CAM2000 hardware, which makes Linear-C programs run as expected in practice.

References

[1] Hall, J.S. (1994). *Associative Processing: Architectures, Algorithms, and Applications*, Ph.D. Dissertation, Computer Science Department, Rutgers University.

[2] Hsu, C.H., Smith, D.E., and Levy, S. (1996). *Linear-C: A Data-Parallel Extension to C*, LCSR-TR-273, Computer Science Department, Rutgers University.

[3] Smith, D.E., Hall, J.S., and Miyake, K. (1993). *Rutgers' CAM 2000 Chip Architecture*, LCSR-TR-196, Computer Science Department, Rutgers University.

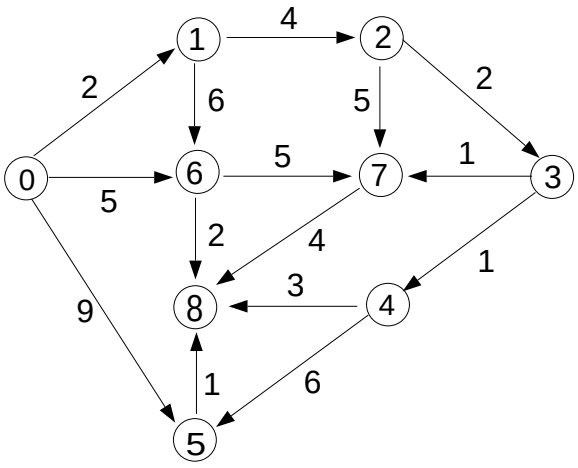

Figure 4: An example graph

ADJ[0] →(1,2) →(5,9) →(6,5)
ADJ[1] →(2,4) →(6,6)
ADJ[2] →(3,2) →(7,5)
ADJ[3] →(4,1) →(7,1)
ADJ[4] →(5,6) →(8,3)
ADJ[5] →(8,1)
ADJ[6] →(7,5) →(8,2)
ADJ[7] →(8,4)
ADJ[8] →∅

Figure 5: Adjacency list representation of the graph

graph G	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}	e_{11}	e_{12}	e_{13}	e_{14}
FROM	0	0	0	1	1	2	2	3	3	4	4	5	6	6	7
TO	1	5	6	2	6	3	7	4	7	5	8	8	7	8	8
W	2	9	5	4	6	2	5	1	1	6	3	1	5	2	4

Figure 6: Edge set representation of the graph

Stmt	C-code	LC-N-code	LC-E-code
(*)		$O(N)$	
2-3	$O(N)$	$O(1)$	$O(1)$
4-6	$O(1)$	$O(1)$	$O(1)$
7-8	$O(N)$	$O(N)$	$O(N)$
9-12	$O(E)$	$O(E)$	$O(N)$
13	$O(N^2)$	$O(N)$	$O(N)$
14-15	$O(N^2)$	$O(N)$	$O(N)$
16	$O(N)$	$O(1)$	$O(N)$
total	$O(N^2)$	$O(E)$	$O(N)$

Figure 7: Time complexity analysis of three implementations

Stmt	C-code
	...
	main ()
	{
	struct entry {int to; int w; struct entry *next} ADJ[N];
	int NEWPL[N], EXPAND[N];
	int MINPL[N];
	int nearest, nearest_pl;
1	init(ADJ);
2	{ int i;
	for (i = 0; i < N; i++)
	NEWPL[i] = Inf; }
3	{ int i;
	for (i = 0; i < N; i++)
	EXPAND[i] = no; }
4	NEWPL[0] = 0;
5	nearest_pl = 0;
6	nearest = 0;
7	while (nearest_pl != Inf) {
8	EXPAND[nearest] = yes;
9	{ struct entry *p;
	for (p = ADJ[nearest]; p != NULL; p = p->next) {
	if (EXPAND[p->to] == no)
10	if (NEWPL[p->to] > nearest_pl + p->w)
11	NEWPL[p->to] = nearest_pl + p->w;
12	} /* end for */ }
13	{ nearest_pl = Inf;
	int i;
	for (i = 0; i < N; i++)
	if (EXPAND[i] == no && NEWPL[i] < nearest_pl)
	nearest_pl = NEWPL[i]; }
14	{ int i;
	for (i = N-1; i >= 0; i--)
	if (EXPAND[i] == no && NEWPL[i] == nearest_pl)
	nearest = i; }
15	} /* end while */
16	{ int i;
	for (i = 0; i < N; i++)
	MINPL[i] = NEWPL[i]; }
	} /* end main */

Figure 8: the C-code

Stmt	LC-N-code
	...
	main ()
	{
	struct entry {int to; int w; struct entry *next} ADJ[N];
	int NEWPL[N], EXPAND[N];
	int MINPL[N];
	int nearest, nearest_pl;
	int ACT[N], NAME[N];
(*)	{ int i;
	for (i = 0; i < N; i++)
	NAME[i] = i; }
1	init(ADJ);
2	NEWPL = Inf;
3	EXPAND = no;
4	NEWPL[0] = 0;
5	nearest_pl = 0;
6	nearest = 0;
7	while (nearest_pl != Inf) {
8	EXPAND[nearest] = yes;
9	{ struct entry *p;
	for (p = ADJ[nearest]; p != NULL; p = p->next) {
	if (EXPAND[p->to] == no)
10	if (NEWPL[p->to] > nearest_pl + p->w)
11	NEWPL[p->to] = nearest_pl + p->w;
12	} /* end for */ }
13	{ nearest_pl = Inf;
	ACT = (EXPAND == no);
	nearest_pl = /\(ACT ? NEWPL); }
14	{ ACT = (EXPAND == no && NEWPL == nearest_pl);
	nearest = /\(ACT ? NAME); }
15	} /* end while */
16	MINPL = NEWPL;
	} /* end main */

Figure 9: The LC-N-code

Stmt	LC-E-code
	...
	main ()
	{
	int FROM[E], TO[E], W[E];
	int NEWPL[E], EXPAND[E];
	int MINPL[N];
	int nearest, nearest_pl;
	int ACT[E];
1	init(FROM,TO,W);
2	NEWPL = Inf;
3	EXPAND = no;
5	nearest_pl = 0;
6	nearest = 0;
7	while (nearest_pl != Inf) {
8	ACT = (TO == nearest);
	EXPAND = ACT ? yes;
9	ACT = (EXPAND == no && FROM == nearest);
10	ACT = ACT && (NEWPL > nearest_pl + W);
11	NEWPL = ACT ? nearest_pl + W;
12	
13	{ nearest_pl = Inf;
	ACT = (EXPAND == no);
	nearest_pl = /\(ACT ? NEWPL); }
14	{ ACT = (EXPAND == no && NEWPL == nearest_pl);
	nearest = /\(ACT ? TO); }
15	} /* end while */
16	{ MINPL[0] = 0;
	int i;
	for (i = 1; i < N; i++) {
	ACT = (TO == i);
	MINPL[i] = /\(ACT ? NEWPL);
	} }
	} /* end main */

Figure 10: The LC-E-code