Greedy Partitioned Algorithms for the Shortest-Path Problem

E. Tick and P. Adamson
University of Oregon

CIS-TR-91-01a
July 1992

Abstract

A partitioned, priority-queue algorithm for solving the single-source best-path problem is defined and evaluated. Finding single-source paths for sparse graphs is notable because of its definite *lack* of parallelism — no known algorithms are scalable. Qualitatively, we discuss the close relationships between our algorithm and previous work by Quinn, Chikayama, and others. Performance measurements of variations of the algorithm, implemented both in concurrent and imperative programming languages on a shared-memory multiprocessor, are presented. This quantitative analysis of the algorithms provides insights into the tradeoffs between complexity and overhead in graph-searching executed in high-level parallel languages with automatic task scheduling.

This article was published in the *International Journal of Parallel Programming*, vol. 20, no. 4, August 1991. This version has been extended with Appendicies containing the source programs analyzed.

Department of Computer and Information Science
University of Oregon

1 Introduction

Many graph-theoretic algorithms have practical applications for transportation, genetic, and communication networks. The speed with which programs solve large graph problems is becoming a litmus test for cost effectiveness of parallel processors. To properly demonstrate efficiency of the hardware, an efficient algorithm is required to distribute workload and facilitate communication among processors. The time it takes to develop the algorithm and efficiently map it to new hardware is a significant part of this effort. With higher-level paradigms, such as concurrent logic programming languages, machine-independent algorithms (that migrate to different host hardware without alteration) can be more easily developed. In many logic programming systems, a built-in process scheduler automatically schedules parallel tasks. This eliminates the need to determine the distribution of work among processors at compile time, and can dynamically adapt the program to changes in the search space.

In this article we review and analyze algorithms commonly used to solve the single-source shortest-path problem. Our main purpose is to characterize solutions to the problem in the context of high-level parallel languages that enable automatic scheduling of processes. Two different classes of algorithms, using depth-first and breadth-first solution strategies, were implemented in Parlog, a concurrent logic programming language [2] and C. Alternative implementations for the critical data structure in the search, a priority queue, were also examined. Empirical results for medium-sized (1000+ vertex) sparse graphs on a shared-memory multiprocessor are supplied to demonstrate the performance characteristics of the algorithms and data structures.

A novel result of this study is the development and evaluation of a partitioned priority-queue algorithm that achieves the highest performance of any parallel logic programming solution to the single-source best-path problem. This algorithm can be viewed as a variant of Quinn's multiqueue [12] approach to Moore's algorithm [9], or as a software implementation of Chikayama's prioritization of Ichiyoshi's algorithm [19]. These relationships are fully discussed in the article and help in understanding the tradeoffs between complexity and overhead in developing efficient parallel algorithms.

The article is organized as follows. In Section 2 we review the shortest path problem, its associated terminology, and concurrent logic programming. In Section 3, the fundamental algorithms in the field are summarized and compared. In Section 4, we introduce several variations of these algorithms appropriate for parallel logic programming, and qualitatively compare their strengths and weaknesses. In Section 5, execution measurements of the algorithms are analyzed. Conclusions are summarized in Section 6.

2 Terminology and Review

A graph $G = (V, E)$ of size n consists of a set V of n vertices (or nodes) and a set E of e nondirected edges. An edge is described by the tuple (u, v) where u is the source vertex and v is the destination vertex. Each edge has a given cost, $cost(u, v)$. The degree of a vertex is the number of edges connected to it. The cost of a path l is the sum of costs of all the edges along l. This article will only be concerned with non-directed graphs (i.e., edges that can be traveled in either direction) that (1) have degree less than six, and (2) contain only positive edge costs. Such graphs are representative of planar maps, for example.

For complexity analysis purposes, the number of processor elements (PEs) available is denoted p, where $p \ll n$, i.e., not unbounded parallelism. The shortest-path, or best-path, problem is a search of a graph. There are two famous shortest-path problems: single-source and all-pairs. This article is concerned with the former: to find the path of least cost from a designated source vertex, S, to every other vertex in the graph. The all-pairs shortest-path problem is to find the shortest paths between *each pair* of vertices in the graph.

Sparse graphs of degree < 6 are of interest to us because they are more difficult to search in parallel than graphs with higher connectivity. Finding the single-source best paths in such graphs falls into the class of *fine-grain parallel problems*. That is, the solution of the shortest-paths problem requires the parallel execution of many small sized, interdependent tasks. In contrast, coarse (or large-grain) parallelism is composed of larger, independent tasks that can be solved simultaneously. Within the framework of parallel logic programming, fine-grain tasks are well suited to stream AND-parallel languages, such as FCP, FGHC, and Parlog [15] with their built-in communication and synchronization features.

Concurrent logic programs just mentioned are built of guarded Horn clauses of the form: "H :- $G_1, G_2, ..., G_m$ | $B_1, B_2, ..., B_n$" where $m \geq 0$ and $n \geq 0$. H is the clause head, G_i is a guard goal, and B_i is a body goal. The commit operator '|' divides the clause into a passive part (the *guard*) and active part (the *body*). Note that if the guard is empty, no commit operator is given. The informal semantics of such programs are: to execute a procedure $p(A_1, A_2, ..., A_k)$ with arguments A_i, the arguments must match (passively unify with) the formal parameters of the head of some clause with name p. In addition, the guard goals must all succeed. If these conditions hold, the procedure invocation may "commit" to that clause (or *any* other clause so satisfying these conditions). The body of the committed clause is then executed. A program successfully executes when no goals remain to be executed.

Synchronization in committed-choice programs in enforced implicitly in the commit

rule. If no clause can commit to a procedure invocation, that invocation *suspends*. Suspension occurs because a passed parameter is not sufficiently instantiated to allow successful head matching or guard reduction. A suspended invocation is *resumed* when a later binding is made to a variable associated with the suspended invocation.

3 Literature Review: Algorithms

A comparison of several "traditional" solutions to the best-path problem revealed that adaptations of Dijkstra's [6] and Moore's [9] single-source algorithms are most suitable for implementation in logic programming. Other more distributed versions have been implemented in logic programming: for instance, Taki's monitor algorithm and Ichiyoshi's nearest-neighbor algorithm [18]. However, these algorithms have significantly increased overheads and search complexity, making them grossly inefficient. As shown in Tick [18], these latter algorithms, developed within parallel logic programming paradigms, do not compare favorably with efficient sequential algorithms, developed within imperative, sequential programming paradigms. One of the purposes of this article is to re-evaluate the notion that efficient parallel algorithms require radical restructuring of control and data.

In this section we review and critique the aforementioned algorithms.

3.1 Dijkstra's Method

Dijkstra's sequential algorithm (Figure 1) is the most well-known solution to the single-source problem. The basis of the algorithm is that each shortest path is either an edge from the source vertex or a single-edge extension of another shortest path. This constitutes a depth-first search, finding all shortest paths after n iterations.

The original algorithm [6] initially creates a sorted list of known paths, sometimes called the *boundary*. Paths in the boundary are represented by a triplet: the last vertex in the path, the path cost, and the next-to-last vertex on the path. By storing the next-to-last vertex, the path from the final vertex to the source can be reconstructed (by following the previous values to other path endpoints all the way to the source). Initially, the boundary contains only the source vertex (a path of cost zero).

Each iteration, the lowest-cost path is extracted from the boundary. This is the shortest path to this destination vertex, called the *pivot*. To determine the shortest paths extending from the pivot, new paths are created along each pivot edge to vertices without known shortest paths. The cost of each of these paths is computed as the path cost plus the pivot-edge cost. The new paths are then added to the boundary, with the pivot noted as the previous vertex in each path. If a path for this vertex is already

```
begin
    for i = 1 to n do
        distance[i] ← ∞;
        found[i] ← FALSE;
    distance[s] ← 0;
    Q.init();
    Q.insert(s);
    while (node ← Q.extract()) do
        work(node,Q);
end

proc work(u,Q)
    begin
        if (found[u] = TRUE) then return                    (1)
        found[u] ← TRUE;                                    (2)
        for (every edge {u,v}) do
            if (distance[u] + cost(u,v) < distance[v]) then
                distance[v] ← distance[u] + cost(u,v);
                Q.insert(v);
            endif
    endproc
```

Figure 1: Dijkstra's Algorithm: the boundary queue, an Abstract Data Type (ADT), is defined as an object Q, with three self-explanatory methods: init(), insert(), and extract().

in the boundary, only the path with the minimum cost is left in the boundary. The boundary is then sorted and the next iteration begins.

The advantage of this greedy method is that a new shortest path is found with each iteration. Johnson [7] showed that for sparse graphs it is best use a priority queue [17] to store boundary. He suggests implementing the queue as a heap. A heap presents a low complexity implementation of the sorted boundary ($O(log(n))$), but without special control to keep multiple paths to a vertex out of the heap the algorithm is no longer guaranteed to execute in n iterations. This does not affect correctness because the shortest path will always be extracted first. Any longer paths to a vertex that were in the heap will have be discarded upon extraction consuming an iteration. The sequential complexity of the algorithm becomes $O(nlog(n))$. Unfortunately, the inherent sequentiality of Dijkstra's technique provides little opportunity for parallelism since each iteration must wait for the completion of the boundary sort to guarantee that the shortest available path will be extracted next [5]. In fact, the operations that maintain the boundary are the only opportunity for parallelism.

The problem is not that parallel execution can't provide speedup — parallel queue

operations have been widely examined [13], but that the algorithm is not *scalable*. A scalable algorithm displays non-decreasing efficiency with increasing numbers of PEs [8], where efficiency is the speedup divided by the number of PEs used. With the single thread of execution that must synchronize each iteration on the completion of the boundary sort, the amount of parallelism is clearly dependent on the average degree (connectivity) of the graph, d. With the heap implementation, the opportunity for parallel execution occurs only when inserting vertices into the boundary at the end of each iteration. Thus, the granularity (size of parallelism) of execution in Dijkstra is governed by d because that is the average number of insertions into the boundary during each iteration. If d is small ($d \ll n$), as in the sparse graphs we analyze here, then this algorithm is not scalable. In Paige [10], complexity analyses are performed which demonstrate this limitation.

3.2 Moore's Method

Moore's algorithm [9] (Figure 2) is a breadth-first search that has been shown to be well suited for parallelism [5]. Although Moore's algorithm uses the same queue-based search method as Dijkstra's, there is one important difference: the queue is not sorted. The Q.insert() method in Moore's algorithm is simply first-in first-out (FIFO), not sorted by the distance from s. Because the boundary is no longer sorted, paths in the queue are merely considered *candidate* best paths when extracted. Therefore Moore's algorithm cannot take advantage of the optimization of lines 1–2 in work() in Figure 1. A global table, recording the current distance of each vertex from the source, is used to check candidate paths. Initially, all table values are set to infinity. When a path is extracted from the queue, the cost of the candidate path is compared to the current cost of the destination vertex. If the candidate represents a new shortest path, the global table is updated. In addition, new paths are extended and added to the queue, as in Dijkstra's algorithm.

Because the queue is FIFO, much computation is wasted extending erroneous shortest paths. The Pape-D'Espo algorithm [11] reduces this waste with a *dequeue* to store the candidates. The newly-extended paths are inserted into the dequeue with the following rule: if a path to this vertex has ever been in the dequeue, then the new path is inserted at the front of the dequeue; otherwise, it is added at the rear. The observation is that if a candidate path through a vertex has been in the dequeue, but a better path has been found, then false shortest paths probably have been generated. Placing the newly-discovered path at the head of the dequeue ensures that shorter paths will be searched before those false paths.[1]

[1] Note that a dequeue is *not* equivalent to a priority queue. See discussion in Section 4.4.

```
begin
    for i = 1 to n do
        distance[i] ← ∞;
        found[i] ← FALSE;
    distance[s] ← 0;
    Qlist.Init();
    Qlist.Insert(s);
    for j = 1 to NumPEs do
        spawn(worker(Qlist));
end

proc worker(Qlist)
    begin
        u ← Qlist.Extract(pid);
        for (every edge {u,v}) do
            if distance[u] + cost(u,v) < distance[v] then
                distance[v] ← distance[u] + cost(u,v);
                Qlist.insert(v);
            endif
    endproc

method Qlist::Insert(node)
    begin
        i ← compute-partition(node);
        Qlist[i].Insert(node);
    endmethod

method Qlist::Extract(i)
    begin
        Qlist[i].Extract();
    endmethod
```

Figure 2: Parallel Moore's Algorithm: compute-partition() maps the vertices onto the queues. The original, sequential algorithm has NumPEs = 1. Processing terminates when all Qlist objects are empty.

The worst-case sequential complexity of the FIFO implementation is $O(ne)$ where nearly every path in the graph is examined [17]. The dequeue implementation reduces that to $O(n^2)$. This is obviously less efficient than the Johnson-Dijkstra algorithm. But when considering parallel execution, Moore's algorithm is superior because there is no synchronized sorting of the boundary after each extraction. Each process thread can independently grab work from the queue, check the global table to determine if the vertex is worthy of expansion, and insert the new paths into the queue when necessary. The parallel complexity of Moore's algorithm is difficult to obtain because of the nondeterminism involved in the expansion of paths. As the number of PEs increases, competition for access to the queue becomes problematic. Queue contention increases, and the order in which paths are examined and expanded varies. In addition, PEs will be idle while waiting for queue access.

To alleviate the queue bottlenecks, Quinn and Yoo [12] proposed three data structures, in lieu of the serial queue, to hold the boundary: a linked array, multiple dequeues, and parallel semi-dequeues. Each of these structures removed the critical bottleneck caused by the single boundary queue, by using a number of multiple queues (one for each PE). Speedup is not scalable, but some of these techniques were shown to achieve 50% efficiency on 16 PEs (simulated results for 500-node graph with connectivity of five). Observing that partitioning Moore's algorithm provided the opportunity for reasonable speedup, we attempted to reduce its complexity by exploiting priority queues from the Johnson-Dijkstra algorithm, as described in Section 4.4.

3.3 Distributed-Monitor Method

Taki's monitor program [18] (illustrated in Figure 3) is an attempt to further distribute the breadth-first path expansion. Unlike Moore or Dijkstra, there isn't any boundary data structure of paths *per se*, but rather an expanding boundary of *processes* that represent paths. Starting from the source vertex, a process is spawned for each possible path expansion from the current vertex. Each process that is created checks to see if its (newly expanded) path represents a new shortest path. If so, new processes are started with all paths that can be extended from the current one, and the current process terminates. If the process's path is not a shortest path, the process terminates and no action is taken. Implemented on a concurrent logic programming system, the boundary queue is the actual running process queue, which is managed by the process scheduler. This allows an ideal distribution of the work (limited only by the ability of the scheduler).

In Taki's algorithm, the global cost table is managed by a single monitor process which serves requests generated by the search processes. To communicate with the

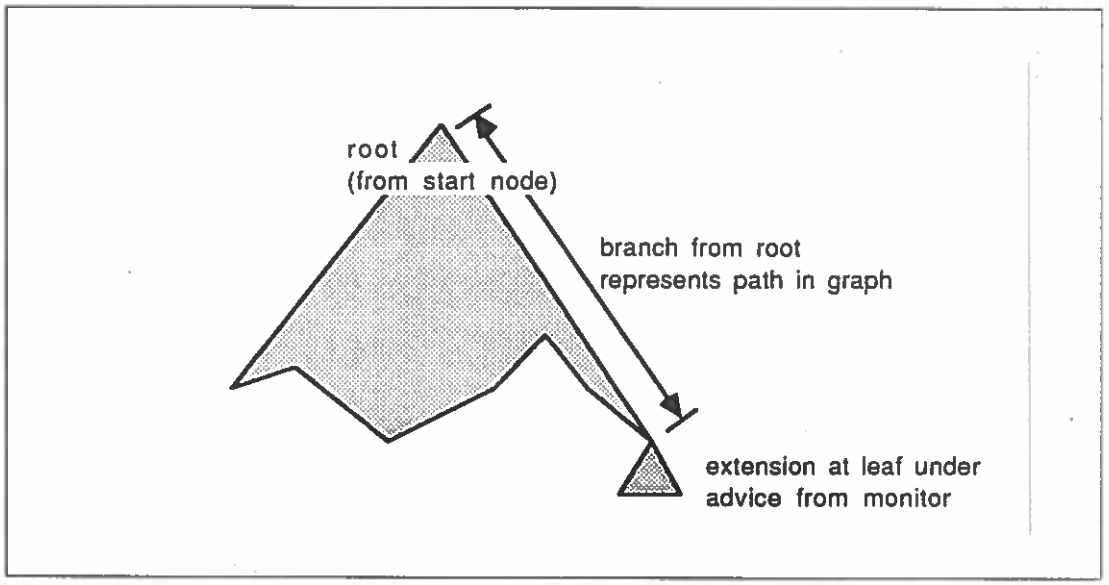

Figure 3: Sketch of Taki's Distributed-Monitor Algorithm

monitor, each process has a stream which is *merged* [16] with the streams from all the other processes into one stream that leads to the monitor. This moves the bottleneck of the traditional Moore method from the single work queue to the interaction between the single monitor and the many search processes. This critical section prevents the technique from providing scalable speedup. Taki's algorithm achieved relative efficiency of 60% on 12 PEs, dropping to 40% on 15 PEs, for the large graph analyzed here (on the Panda FGHC system [14] executing on a Sequent Symmetry). In any case, the program could not achieve real speedup with respect to Dijkstra's algorithm, on any number of PEs, because of the increased complexity of independent searches.

3.4 Nearest-Neighbor Method

At the other end of the spectrum from Dijkstra's depth-first method is Ichiyoshi's fully-distributed, nearest-neighbor algorithm [18] (illustrated in Figure 4). In this algorithm, the graph is represented as a static network of processes. Each vertex in the graph is a running process with a communication stream to each neighboring process (vertex). The stream corresponds to an edge in the graph. Execution begins with vertex processes communicating with each other, updating their neighbors with their current best path. When a vertex receives a path message that has less cost than its current best-cost path, the vertex updates its neighboring vertices with the new cost information. More costly path messages are ignored. Eventually all vertices contain best paths and communication ceases.

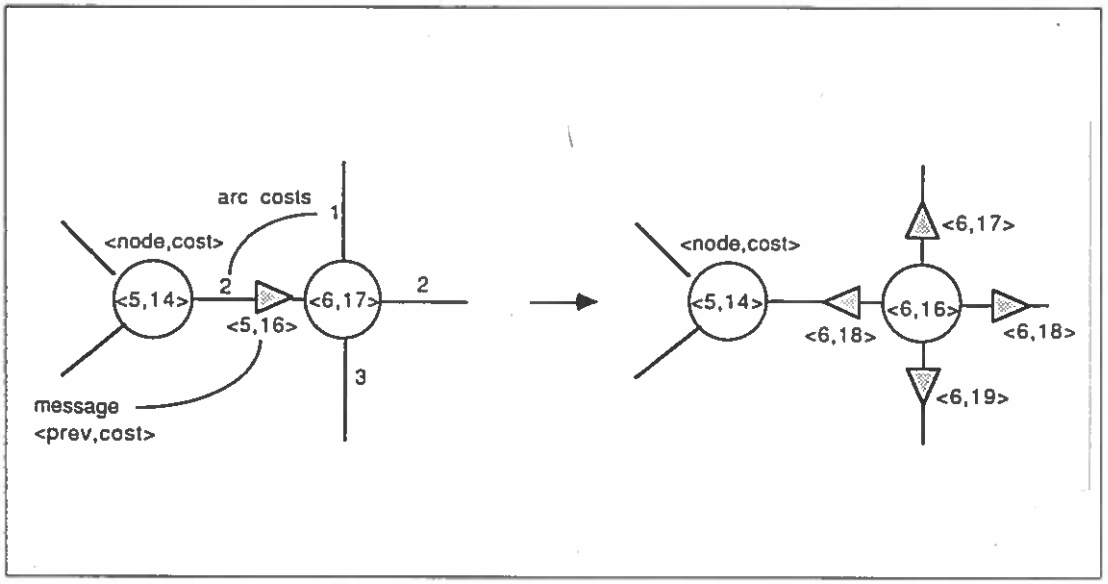

Figure 4: Sketch of Ichiyoshi's Nearest-Neighbor Algorithm

There is potential for a significant amount of wasted work due to the nondeterministic exchange of messages, especially when the number of vertices greatly exceeds the available PEs. Nodes may receive "false" best-path messages before the actual shortest path arrives. The false messages flood the network with false path extensions. Without enough PEs to process the flood of messages, true best-path information may take longer to propagate to other processes because of the backup of the erroneous data. Ichiyoshi's algorithm achieved relative efficiency of 93% on 15 PEs (again, on Panda executing on Symmetry). However, the real efficiency, with respect to Dijksta's algorithm, was only 8%.

A solution proposed by Chikayama [1] to reduce the amount of wasted work, was to add some determinism to the process communication by assigning *priority* to messages based on the value of their best-path information. Assigning a priority as the inverse of the path cost (lower path cost = higher priority) ensures that the shortest-path message will arrive before any other recently generated message. Wada and Ichiyoshi [19] measured this algorithm in the concurrent logic programming language KL1, executing on the Multi-PSI/V2 distributed-memory multiprocessor. The mesh topology of the Multi-PSI provided a particularly advantageous mapping of vertices onto PEs. In addition, Multi-PSI implements prioritization in microcode. However, even with these hardware advantages, the best achieved efficiency was 50% on 16 PEs, dropping to 31% on 64 PEs (40,000-node grid graph with connectivity of four).

4 Experimental Methodology

In this section, five implementations of the previous algorithms are discussed. Three programs are direct parallelizations of Dijkstra's algorithm and two programs are adaptations of Moore's breadth-first approach. The differences between the Dijkstra implementations are in the boundary data structures: a linear list, a passive heap, and an active heap (defined herein). Though we have already noted that Dijkstra is poorly suited for parallelization, the three implementations provide insight into the performance of different data structures as well as the fastest sequential execution time. The two partitioned versions of the breadth-first Moore method improve on Quinn's multiqueue approach (to reduce competition for queue access among processes), by using cost-sorted queues instead of FIFO queues.

When selecting algorithms to implement with logic programming, it is important to consider that not all data structures in traditional languages map efficiently to logic programming. Specifically, there are no constant-time access, destructible arrays in most logic programming implementations.[2] Destructible, or mutable, data structures must be simulated with structure copying. In order to assign a value to an element more than once, portions of the structure must be copied, which is prohibitively expensive for large arrays. For this reason, algorithms that performed reasonably well in imperative programming languages, may perform poorly in logic programming because they rely on mutable structures. From this perspective, the Dijkstra method is particularly adaptable to logic programming because there is no need for an updatable array to track each vertex's current cost. The shortest cost is simply recorded once for each vertex as it is extracted from the boundary. Moore's algorithm, on the other hand, relies heavily on the ability to obtain and change the current shortest-path cost of a vertex in the global cost table.

In the next sections we examine the performance characteristics of these algorithms: prime candidates for efficiently solving the single-source best-path problem. The methodology used is empirical measurement of Parlog and C implementations executing on a shared-memory multiprocessor, rather than asymptotic complexity bounds, for numerous reasons. First, complexity bounds for *nondeterminate* programs (which depend on worst-case assumptions) do not give us intuitions about how parallel pro-

[2] Some logic programming systems have destructive arrays, e.g., implemented by explicit programmer annotation or by multiple reference bit(s). Our use of JAM Parlog [4], with no such capability, should not be construed as a serious shortcoming of this analysis. When appropriate for comparison, we calibrate all programs to implement arrays as trees with structure copying, increasing update overhead uniformly. In these algorithms, search strategy, rather than array access, makes the major contribution to execution complexity.

grams behave, whereas empirical measurements do. Second, constant overheads in complexity bounds are significant in programs not operating at asymptotic data sizes. We wish to show the tradeoff between complexity and overhead in actual programs. Third, we want to deflate claims that parallel programming requires radical departures from "traditional" control and data structures, by showing the real performance characteristics of a variety of designs.

4.1 List

The simplicity of Dijkstra's algorithm with a sorted linear list provides a little parallelism with the least amount of overhead. The parallelism is controlled by the built-in suspension mechanism of Parlog. As noted before, each iteration must wait for the new paths to be inserted into the boundary from the previous iteration, before extracting the next shortest path. The main loop will suspend waiting for the first element (the next shortest path) in the boundary list to be determined after the insertion of new paths. As soon as this first element is bound by the insertion process, the next iteration can proceed. Thus the next iteration can begin work even if the rest of the list is undetermined; the inserter may complete the sort of the rest of the list in the background. The insertion process should be executed in such a fashion as to determine the first element in the new boundary as soon as possible.

A new path is inserted into the boundary by starting at the beginning of the boundary and scanning down until the proper location in the boundary list is found for the new path. This provides the opportunity to ensure that no shorter path to this vertex is in the boundary. But it would be inefficient to continue down the list to check for a longer path (if it existed). Thus the boundary may contain extra nodes, and this method does not guarantee that execution will halt after n iterations. The technique is still correct, as shortest paths will always be encountered before the extraneous path. However, extra iterations and processing will be consumed discarding paths for vertices that already have shortest paths.

The main drawback is that this linear, list-oriented approach is not the most efficient method for maintaining a sorted list, especially as the boundary becomes large. For this reason the more efficient heap data structure was examined in hope to reduce the complexity of boundary management from $O(b)$ to $O(log(b))$, where b is the size of the boundary.

4.2 Passive Heap

A heap is a binary tree implementation of a sorted list [17]. In logic programming a heap can be represented concisely with nested predicates:

```
node(Data, Leftnode, Rightnode)
```

where `Leftnode` and `Rightnode` are themselves `node/3` structures and `Data` is a value. Thus each node in the heap consists of data and two child nodes. This is a pure data structure, hence the term "passive," i.e., although the structure can grow dynamically, a manager procedure is required to operate on it. The heap invariant is that the value of each node is less than or equal to that of its children. Therefore the root of a heap, the outermost predicate, is the minimum value and the next to be extracted.

After the extraction of the root of the heap (a minimum path), the heap is rootless or *partial.* Before another extraction can be made on a partial heap, the heap must be corrected and checked for balance. An insertion of a new value into the heap produces this effect. Since most extractions will be followed by a series of insertions (in the normal execution of the algorithms presented here), the heap will be corrected automatically. But in the cases when new paths are not inserted and two successive extractions are required, a dummy path with cost infinity is inserted into the rootless heap between extractions. This action is called *heapification.* The propagation of this large element down the heap (called *heapifying*) will push up a node to the root and maintain the balance of the heap.

The passive heap provides an efficient sorted boundary, at some cost. First, heapification required between two extractions is pure overhead. Note that the search will complete before any of the dummy (infinite value) heapified nodes will be extracted. The most costly overhead comes from the representation of the heap as a single nested predicate. Inserting a node into the heap entails creating new links down the tree as the node finds its proper place within the tree. In a language without destructive assignment, this relinking of nodes must be executed by structure copying, an expensive action. The constant allocation and deallocation of memory involved in structure copying greatly adds to execution time in the form of memory management overhead.

The complexity of boundary insertion is reduced from $O(b)$ (in list) to $O(log(b))$ (in heap); however, a price is paid in constant overhead. Both the passive heap and the list can support pipelined accesses to the boundary, i.e., as structure copying progresses, subsequent accesses can proceed with no suspension. However, copying a single heap node structure involves more overhead than copying a list cell. As a result, the heap tends to suspend more often, stalling the software pipeline. The tight loop that manages the pipelined access to the heap cannot be effectively distributed onto multiple PEs.

In addition, the heap boundary will contain more extraneous paths than will the list boundary. With the list boundary, a path to a vertex is not added if a shorter path is already in the boundary. This optimization is easy to implement in the list because all shorter paths are compared while finding the new path's place in the list. Such an

```
mode node(?,?,^,^).
node([], _, L, R) <- L = [], R = [].
node([cost(X)|Ms], V, L, R) <- X = V, node(Ms, V, L, R).
node([extract(X)|Ms], V, L, R) <- X = V, partial(Ms, V, L, R).
node([insert(Vin)|Ms], V, L, R) <-
    Vin = a(N0,C0,P0), V = a(N1,C1,P1),
    Even is (C0+C1) mod 2,
    intoheap(Even, Vin, V, Vnew, L, R, Ls, Rs),
    node(Ms, Vnew, Ls, Rs).
node([replace(Vin)|Ms], _, L, R) <-
    L = [cost(Vl)|L1], R = [cost(Vr)|R1],
    compare(Vl, Vr, Vin, V, L1, L2, R1, R2),
    node(Ms, V, L2, R2).
```

Figure 5: Active Heap Node with Two Children (Parlog)

optimization with a heap is non-trivial, because not every node with shorter cost is encountered during an insertion. Thus the heap boundary will be larger, increasing the number of extractions of non-shortest paths, increasing the number of iterations. As shown in Section 5, this is a costly problem, because increases in complexity are not easily overcome by adding more PEs.

4.3 Active Heap

The active heap attempts to both eliminate the overhead of structure copying, and better distribute insertion/deletion management, in the passive heap. Instead of implementing the heap with a pure data structure, the heap is represented by an isomorphic binary tree of perpetual processes. Each process represents a heap node containing the data and three communication streams: an input stream from the node's parent, and two output streams to the node's children. Changes in the heap are effected by node processes swapping data values with their children over the communication streams. Each node process remains active until its input stream is closed.[3]

A portion of the Parlog code implementing the active heap is shown in Figure 5. Procedure `node(I,V,L,R)` defines a node process with an input stream `I`, a state value `V`, and two output streams (to children), `L` and `R`. When the input stream is closed, the node terminates by closing its output streams (clause 1). A `cost/1` message is answered by binding the return variable to the node's value (clause 2). If this node is the root, then it can receive an `extract/1` message, which it binds to its value (clause 3). The node then changes its state to a partial heap. To insert a new node, `Vin`, into this heap, the new cost `C0` must be compared to the current node's cost `C1` (clause 4).

[3] For proper termination, a node process closes its output streams when its input stream is closed.

Procedure `intoheap/8` performs this comparison: whichever cost is lower is installed at this position in the heap, and the larger cost node is inserted at one of the children. Either child can be chosen, so a balancing heuristic inserts on the left if `C0+C1` is even, and the right if odd.

Whereas `insert/1` forces an extra node into a full heap, a `replace/1` message inserts a new node `Vin` into a rootless heap (clause 5). The costs of each of the root's children are accessed by sending `cost/1` messages down their input streams. Procedure `compare/8` performs an exchange if `Vin` is not minimal. Otherwise `Vin` is placed as the current node, and heapification terminates.

Since the heap is a set of running processes, it can be distributed over different PEs automatically by the process scheduler. With different sections of the heap running on different PEs, balancing actions in the heap, resulting from an extraction or insertion, are better distributed. The need for heapification of a partial heap still exists, but the heap is ready for the next iteration as soon as the new root has been decided. This implementation provides more parallelism, but adds the relatively expensive overhead of maintaining a process for each node in the heap and the communication streams between them. Again, sequential execution has been inflated with the hope that distribution of the heap onto more PEs will decrease the overall execution time, when more PEs are made available.

4.4 Partitioning Dijkstra

Moore's algorithm has been shown, in Section 3.2, to be adaptable to parallel execution if the bottleneck of queue access can be eliminated. While partitioning Moore's algorithm may provide better distribution of work, a considerable amount of time is still spent examining paths that are ignored in Dijkstra's greedy algorithm. What is desired is the distribution of Moore's algorithm with the efficiency of Dijkstra's algorithm: a smarter breadth-first search. We do this by combining the flow control of Moore's technique with the cost-based boundary of Dijkstra's technique.

While the use of a dequeue reduces wasted work (prevalent when processing candidate paths in a FIFO manner), a dequeue is still not as efficient as a priority queue. This is because the placing of candidates in the front of the queue, regardless of cost, can preempt the processing of much shorter candidate paths in the queue. The priority queue ensures not only that any wasted expansion will be counteracted as quickly as possible, but also that the shortest paths are found as quickly as possible. The problem is to implement the priority queue as efficiently as a dequeue. Both the list and active heap implementations are examined here.

As illustrated in Figure 6, work on the graph is statically divided among a number

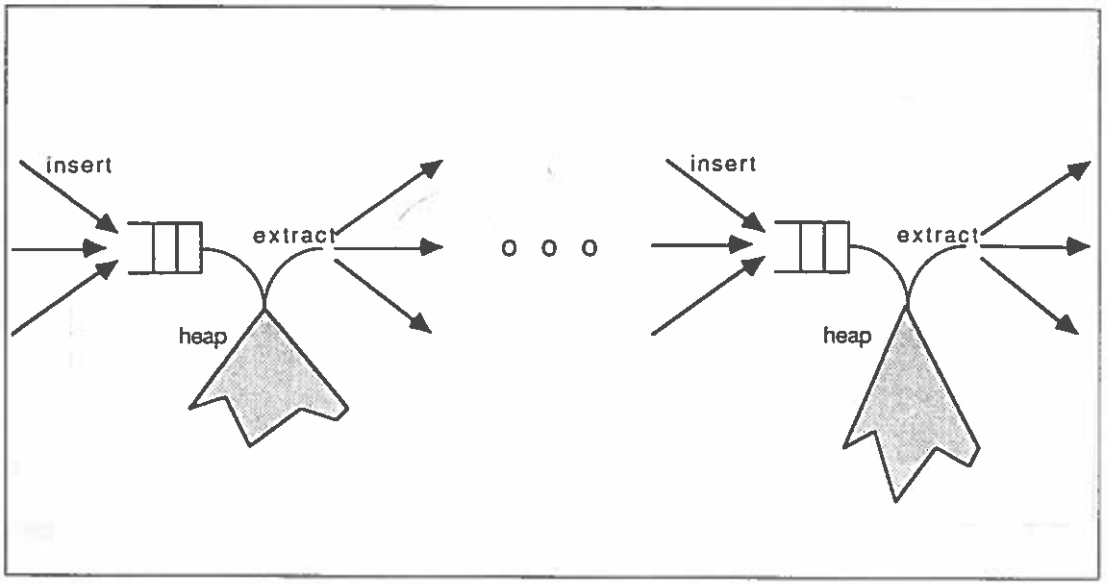

Figure 6: Sketch of Partitioned Dijkstra Algorithm

of processes called workers.[4] Each worker is assigned a set of vertices for which it will track the current shortest cost. In addition each worker has its own priority queue from which it obtains candidate shortest paths for its assigned vertices. The number of workers can equal the number of PEs, although that is not necessary as is explained later. The limiting case, where the number of workers equals the number of vertices, reduces to Chikayama's variation of Ichiyoshi's algorithm with message prioritizing.

Essentially, each worker is performing the greedy Dijkstra algorithm on its own partition. Note that, unlike Dijkstra, workers are examining only the shortest path *currently available*. Early in the search, a worker may receive false shortest paths, and other workers may later find shorter paths, as with a breadth-first search. Thus a worker's control flow resembles that in Moore's algorithm; treating boundary extractions as candidates, not as final shortest paths. To alleviate the bottleneck of a global cost table, evident in the monitor technique described in Section 3.2, the current cost of a vertex is stored locally by the worker.

The boundary partition examined here is based on the multiqueue Moore approach first documented by Quinn and Yoo [12]. Each worker can extract only from its own queue, but can insert into all workers' queues based on a static partition. Each queue has its own manager to interface between the workers and the queue. The queue manager accepts insert and extract requests to its queue via a merged stream. Because an N-way merger is unavailable in JAM Parlog, in these experiments, an N-way merge

[4]The pseudo-code of this algorithm is essentially that given in Figure 2, with a prioritized queue method.

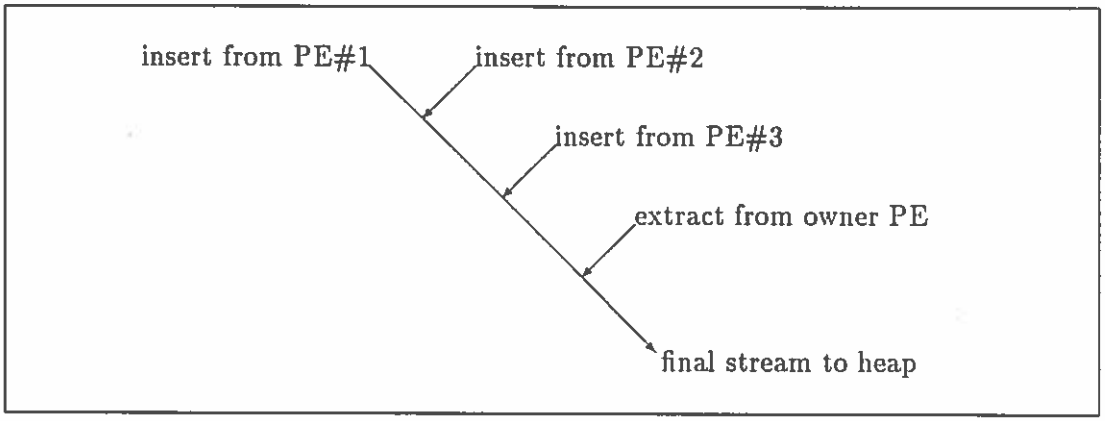

Figure 7: Graphical Representation of a Four-Way Merge Chain

was built from a chain of binary merges as shown in Figure 7. This figure points out the important characteristic of the chain: the merges are not fair. Each worker's extract request is given highest priority in the merge. By giving extracts priority, work is generated at a higher frequency. This priority keeps the queues full, thus sorting has a more potent effect of filtering away erroneous paths (i.e., moving these "false" paths towards the rear of the queue). Although this allows efficient execution of the algorithm in Parlog, it does not in C. The key difference is that Parlog's ratio of computation to communication cost is high (the instruction set is emulated) whereas C's ratio is low. Therefore in C, communication costs are proportionally higher, and by issuing messages less frequently, the number of messages is throttled and performance improves (see Section 5.3).

Distribution of work is obviously dependent on an advantageous assignment of vertices to workers. "Optimal" partitioning entails reducing communication between workers (with minimal boundaries between partitions), yet allowing the search to progress in a maximally parallel manner (with maximal boundaries between partitions). This tradeoff of partition granularity, as well as the affects of edge-cost variance, makes finding an "optimal" partition a difficult problem, perhaps NP-hard. One viable family of partitioning functions, based on vertex number modulo the number of workers [12], is also explored in the next section.

For comparison, the performance measurements of two Parlog versions of the partitioned Moore algorithm are presented in the next section: a linear-list queue and an active-heap queue. For Parlog, the passive-heap version is not discussed because it demonstrated characteristics similar to the list boundary in the Dijkstra implementation. However, we implemented the passive-heap algorithm in C (where an active heap is difficult to program) for comparison with the logic programs.

search	partitioned	boundary	cost table
greedy	no	sorted list	array
			tree
		passive heap	array
		active heap	
	yes	sorted lists	trees
		active heaps	

Table 1: Summary of the Single-Source Shortest-Path Algorithms Analyzed

5 Discussion of Results

A summary of the algorithms discussed in the previous sections is given in Table 1. The implementations of the cost table and boundary refer to the specific codes measured here. The partitioned programs use a simple modulus mapping function, discussed further at the end of this section.

Statistics were compiled by running each of the implementations with a 1237-vertex graph. The graph was derived from a digitized street map of Shimooma, a suburb of Tokyo. In this context, vertices correspond to intersections and edge costs are the length of streets between two intersections. Execution times were measured on a 24 PE Sequent Symmetry with the JAM Parlog emulator, version 1.4 [4]. All execution times reported are the average of a series of consecutive executions. The standard deviation of the data sets reported for speedup was always within 4%.

The JAM scheduler [3] is based on local ready goal queues, one per PE. The head of a queue has the oldest goal ready for execution, whereas the tail of the queue has the latest goal. The scheduler normally selects from the tail, executing depth-first. However, the head of the queue is accessed for both time slice interrupts and when an idle PE steals work. Idle PEs search the PE queues in some order relative to themselves, thus spreading demand. The scheduler was optimized for a shared-memory multiprocessor, and would need modification to scale well on a distributed system. The key point, however, is that goal priority, granularity, etc. are not used to schedule goals — only queue position is used.

5.1 Characteristics of Boundary Data Structures

Figure 8 compares the execution times of the three non-partitioned Dijkstra implementations. These trends support an earlier conjecture (Sections 3.1 and 4.1) about the limited parallelism in Dijkstra's method. Note that each algorithm was fastest on four PEs, the average degree of the graph. The list boundary is fastest because it provides

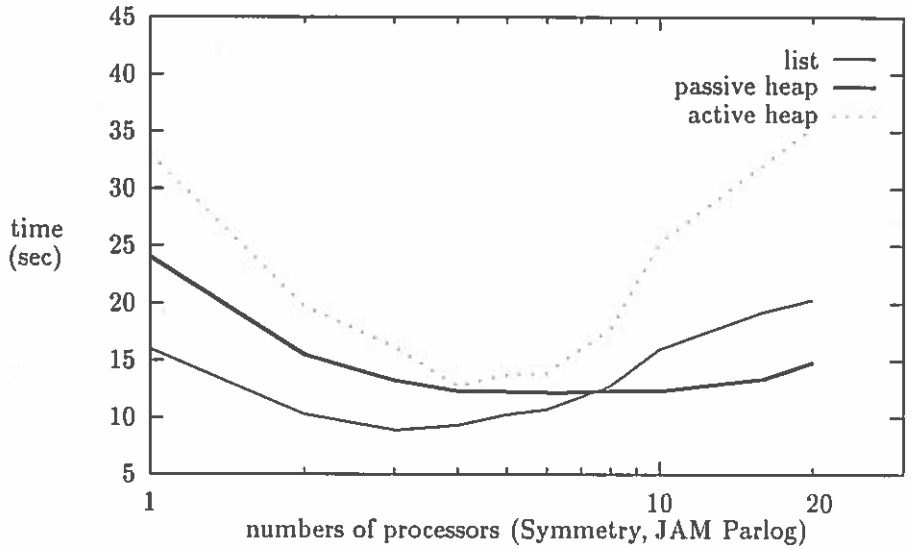

Figure 8: Execution of the Non-Partitioned Dijkstra Implementations (with Cost Arrays)

limited parallelism with the least amount of operational overhead. The fastest times for the list-boundary version demonstrate that the importance of simplicity cannot be overstated. As the number of PEs increased, however, the performance of the list boundary degrades because processes contend for the limited amount of available work.

The passive-heap boundary never achieves the speeds of the list version. Its performance takes longer to degrade because more work is available due to its overhead to maintain the heap ("false parallelism"). Even though a passive-heap boundary will sort in fewer operations, which shortens the length of each iteration, it cannot compete with a list because more iterations occur. This increase in iterations is caused by the presence of extraneous paths in the boundary that the list version filters out (Section 4.2).

An important issue is the cost of heapifications when two successive extracts are made from the heap. Tests showed that after completing execution, 660 entries were left in the heap, 590 of which were dummy paths resulting from heapification. This is indicative of the number iterations resulting in no new paths being inserted into the heap. This is disconcerting considering that the graph contained only 1237 nodes. Developing sophisticated heap implementations, with little or no heapification overheads, is a worthy research topic. In logic programming, however, the overheads incurred by added sophistication will likely not break even with the simple heapification overheads measured here.

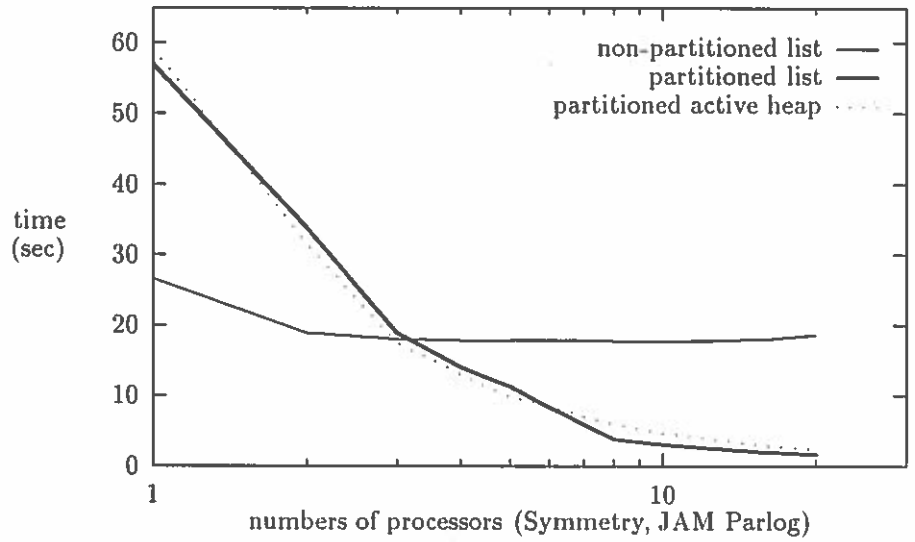

Figure 9: Comparison of Partitioned and Non-Partitioned Algorithms (with Cost Trees)

The active-heap boundary shares the complexity handicap of the passive heap, but another important result, about the distribution of work, can also be seen in Figure 8 (as well as in Figure 12, to be introduced later). The advantage of distributing the heap among different PEs is limited. As the active heap grows in size, the running processes that compose the heap will be distributed over more PEs. Such distribution is desirable, but only up to a point. The distribution needs to be limited because messages traveling between active-heap nodes on different PEs will be much more costly than those that travel between nodes running on the same PE. Therefore at some point the benefits of distribution are overtaken by the high cost of synchronization between different PEs. For the 1237-vertex graph this point is reached between six and eight PEs (see Figure 8). For larger graphs, this point will move to the right.

5.2 Characteristics of Partitioning

Figure 9 compares the partitioned implementations to the fastest non-partitioned algorithm (the list-boundary version). Binary trees are used to implement the cost tables in these algorithms, c.f., array cost tables used in Figure 8. This results in the "non-partitioned list" curve (Figure 9) being higher than the "list" curve (Figure 8), although the algorithms are the same. Binary trees are used because partitioned algorithms require multiple cost-table updates, which cannot be efficiently implemented in JAM Parlog.

Executing partitioned Dijkstra on one PE reduces the algorithm to a single queue,

sorted by cost — the same as the non-partitioned algorithm. Combined with the fact that all three programs use the same type of table to store costs, the execution times for one PE in Figure 9 are revealing. Relative to the tight control loop in the non-partitioned algorithm, the expense of partition management doubles the execution time. However, the implementation achieves improving speedup. Although PEs must be added to absorb the extra work of the queue manager, efficiency (with respect to the non-partitioned algorithm) remains steady and even improves for this data set (as shown in Figure 12). Note that the choice of boundary data structure does not affect partitioned Dijkstra significantly.

Figures 10 and 11 plot the number of reductions and suspensions in the partitioned Dijkstra (with list) algorithm *vs.* the number of partitions, ranging from 1 to $p-1$, for $p = \{10, 16, 20\}$. In a logic programming language, a reduction is equivalent to a procedure invocation. A suspension is equivalent to a fine-grain task switch, incurred because of data synchronization. Analysis of reduction and suspension behavior is more useful than that of raw execution times when evaluating scheduler effectiveness. Non-partitioned program performance is included for reference in the figures: reductions and suspensions are constant since there is no partitioning. The partitioned Dijkstra reduction curves exhibit a concave shape. The reductions executed increase at both ends of the partition spectrum, for the following two reasons.

As mentioned before, when the number of partitions approaches one, the execution complexity reduces to that of the depth-first non-partitioned Dijkstra algorithm (which has a lower curve because of lower overheads). Above a sufficient number of PEs, the number of reductions remains constant for a given number of partitions. For few partitions, reductions decrease when partitions increase because management of smaller priority queues is cheaper. Thus we observe rapidly decreasing reductions, leveling off when average priority queue size levels off.

At the other end of the spectrum, there is no advantage of increasing the number of partitions beyond the number of PEs. When the partitions increase past this point, nondeterminism dominates the algorithm's execution complexity. In the limit, each partition contains one vertex, and no prioritization is exploited. Even for only two partitions per PE, complexity increases sharply. For JAM Parlog searching the graph analyzed here, when the number of PEs was greater or equal to the number of partitions, execution times fluctuated wildly: a standard deviation of over 40% was observed. We also observed a sharp increase in suspensions as the number partitions and PEs approach. The greater number of partitions give the task scheduler too many tasks to choose from, resulting in poor scheduling decisions. The lesson learned was to minimize reductions with approximately two PEs per partition. We shall see in the next section

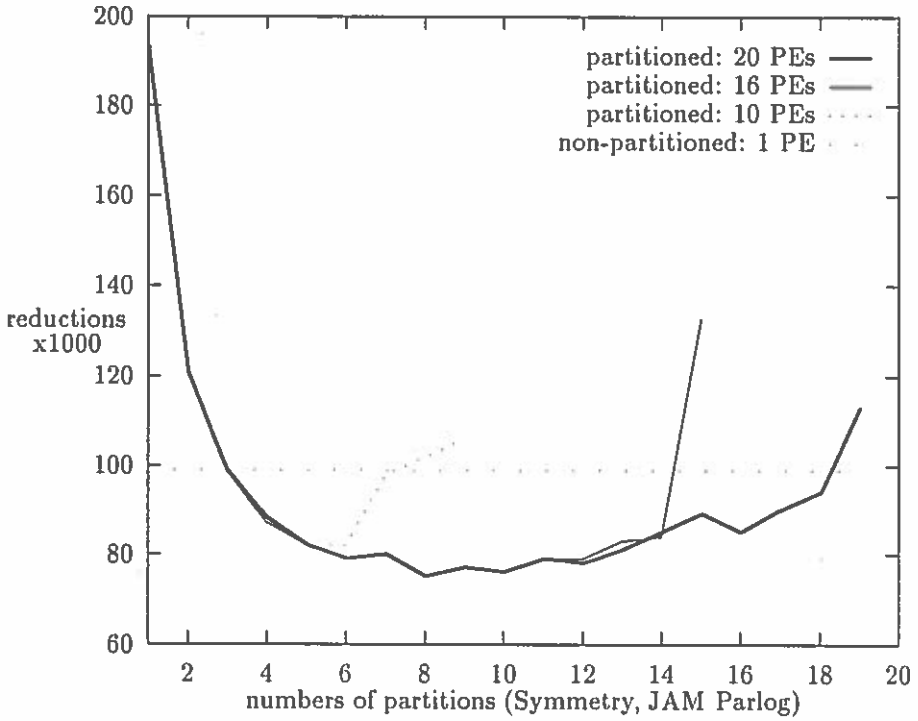

Figure 10: Number of Reductions in List Algorithms

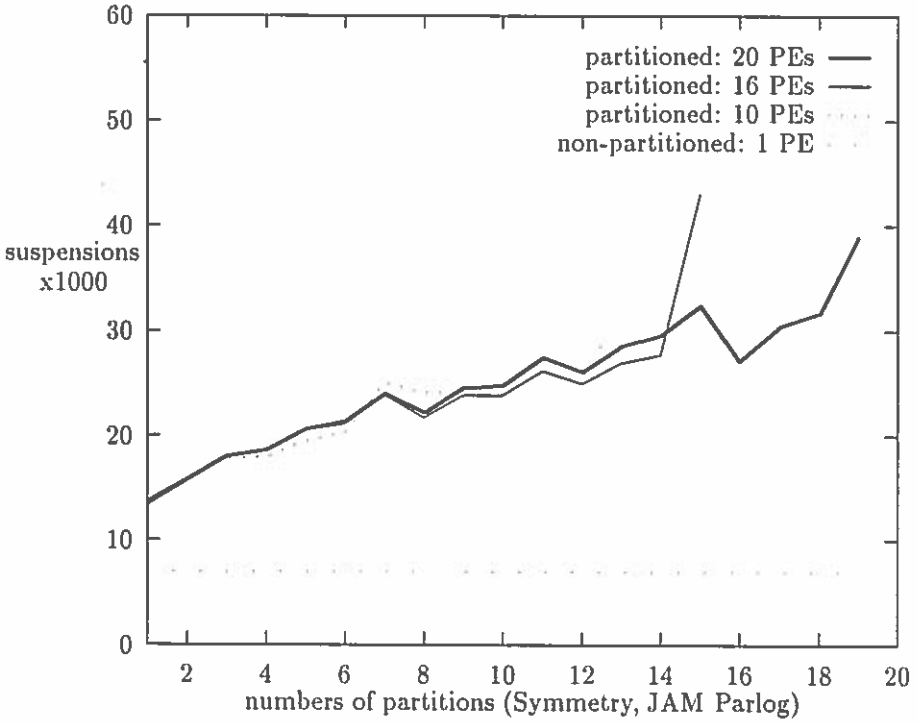

Figure 11: Number of Suspensions in List Algorithms

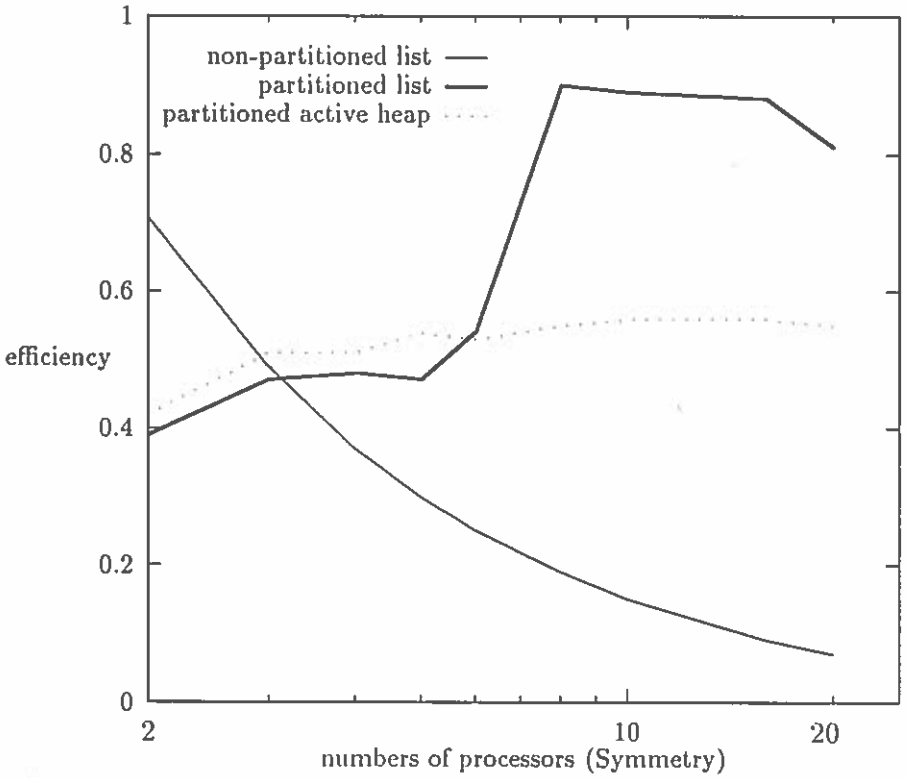

Figure 12: Efficiency of 1237-Node Map (w.r.t. Non-Partitioned List)

that by statically allocating partitions to PEs, this problem is avoided.

Figure 12 displays the efficiency of the partitioned Dijkstra programs *vs.* the number of PEs, for the 1237-node graph. For example, the partitioned-list efficiency curve is calculated as:

$$\text{efficiency} = \frac{1}{\#\ \text{PEs}} \times \frac{T_{\text{non-part list}}}{T_{\text{part list}}}$$

In these plots, the number of partitions giving the minimum average execution times were measured.

The most important characteristic revealed by the efficiency plots is the sudden increase in efficiency from 6–8 PEs. This exposes another nondeterministic aspect of the algorithm, related to the mapping of vertices to partitions. The increase in efficiency is not due to a decrease in the complexity in the algorithm, but rather to a change in the order in which the paths are examined. The distribution of vertices among partitions plays an important role in this order of expansion. It is feasible that groups of expanded paths occur within one partition's set of vertices, momentarily starving the other partitions and causing them to generate wasted work as they empty

their queues.

With the present labeling of our 1237-vertex graph coupled with simple modulus mapping, execution times dramatically improve when the partitions increase from three to four. This implies that vertices that are critically important to the search (called "hot spots") are better distributed (split apart) when the partitions increase to four. Given another partitioning scheme, even with four partitions, the hot spots might remain grouped, causing poor efficiency. In the next section, where the underlying system characteristics are different, an alternative modulus partitioning function achieves higher performance.

5.3 Imperative Implementation of the Algorithm

Most applications are written in imperative languages, so it is of considerable interest to determine how the algorithms discussed in this paper perform when implemented in such languages. For this purpose, we implemented the partitioned static-heap algorithm in C on the Symmetry. It should be first be noted that a translation of this algorithm from Parlog into C is not straightforward:

- JAM Parlog has an automatic process scheduler so that the graph partitioning is independent of the number of PEs. The C implementation fixes the number of partitions and PEs to be equal.
- Parlog facilitates the nondeterminate merge of message streams, implemented as lists. This is emulated in C with circular message buffers, implemented with arrays. In Parlog, extract and insert messages are nondeterministically merged. In C, the frequency of extract messages is explicitly determined by an input control parameter.
- JAM Parlog has a builtin garbage collector, which is emulated in C by an explicit stop-and-copy algorithm invoked if a heap overflows.
- Parlog and C differ in natural and practical data structures. C allows the implementation of the cost table as an array, whereas in the partitioned Parlog algorithms, binary trees must be used. Parlog allows the implementation of active heaps, whereas this is difficult to program in C.
- Parlog facilitates termination with the short-circuit technique applied to logical variables. Termination of the C program requires an explicit monitor checking when all heaps have emptied.

The most striking characteristic of the C implementation is its absolute speed compared to JAM Parlog. To accurately measure the parallel performance, a 21,029-node graph

was used (17 end-to-end copies of the Tokyo map). A single Symmetry PE solved this problem in 15.6 seconds, about 17 times faster than the non-partitioned list Parlog version.[5] The single-PE C version is 29 times faster than the partitioned-list Parlog version (used as the baseline for efficiency measures in the previous section). Destructive arrays and optimized C compilation contribute the most to this speed increase.

The C implementation was given two input parameters to control the search: the *eagerness* or frequency of extract requests made to a given queue, and the *granularity* of the partitioning. These parameters are needed to fine-tune the implementation, which is sufficiently faster than the Parlog version to introduce problems of over-eager message spawning and too-fine partitioning. The C program can perform an iteration of the basic algorithm so rapidly that excessive extract requests flood the system with messages from an over-eager search (the Parlog program has the exact *opposite* problem). The eagerness parameter, indicating the number of iterations required between issuing each extraction request, throttles flooding.

The simple modulus partition function, which works well for Parlog, fails for C because of the differing computation to communication cost ratios. Parlog, with a high ratio, is relatively immune to severe hot spot problems. C, with a low ratio, is more susceptible to larger percentages of performance degradation. To clarify, consider the following mapping function:[6]

$$\text{partition} = (\text{node}/\text{granularity}) \; \textit{mod} \; \text{PEs}$$

The effect is to compose partitions out of larger groups of local nodes. This simple function is effective because the graph is labeled so that local nodes have consecutive numbering. The algorithms presented act locally within the worker's partition, but must communicate via messages for path extensions to other partitions. Thus partition granularity is inversely proportional to communication. Very large granules have the disadvantage, however, that information propagation is delayed because of lower worker efficiency. Without a coarse-enough mapping, the algorithms incur too much communication, which in the case of C leads to performance degradations proportionally greater than Parlog's.

Figure 13 shows a family of speedup curves with different search parameters (eagerness, granularity). We show only a representative sample of the parameter space. Speedup is calculated with respect to the single-PE C version, which is equivalent to Dijkstra's algorithm with the added overhead of emptying the heap for termination. The results illustrate the nondeterminate interaction between the number of PEs and

[5] assuming that the speed scales linearly with the size of the graph.

[6] This is a generalization, in some sense, of the two-dimensional grid partitioning function presented by Ichiyoshi and Wada [19].

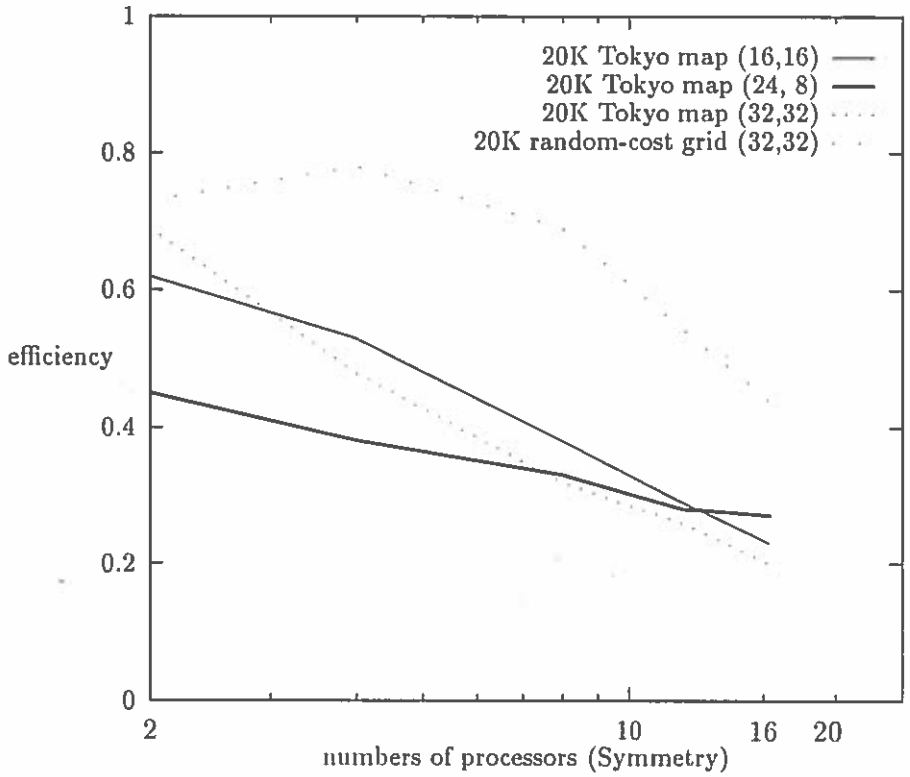

Figure 13: Efficiency of Partitioned Static-Heap Algorithm (C Version)

the search parameters. A maximum efficiency of 27% was achieved on 16 PEs for the search parameters (24,8).

C speedup was lower than Parlog speedup because the efficiency of the C implementation was greater. However, the algorithm produced improving speedups with increasing numbers of PEs. The 20K-node Tokyo graph has lower speedup for two reasons. First, the graph is constructed by smaller end-to-end graphs connected by single arcs (for ease of construction). This constrains the search. Second, the partition function does not form perfect granules because the node labeling is not perfect in this regard. For comparison, we tested a 20K-node random-cost (1–99) grid that had neither of these problems. On 16 PEs, search parameters (32,32) achieved the best efficiency of 44%. This efficiency result is compatible with Ichiyoshi and Wada's measurements of Chikayama's algorithm [19].

6 Conclusions and Future Work

In summary, the contributions of this article include: (1) a complete survey of parallel algorithms solving the single-source shortest-path problem; (2) exposition of the relationships between the algorithms in terms of their fundamental component: the boundary; (3) introduction of a new variant of these algorithms, using partitioned priority queues; and (4) empirical performance measurements and analysis of a realistic sparse data set, executing on both concurrent logic programming and imperative language systems.

We have successfully implemented a parallel algorithm for the single-source best-path problem, in both concurrent logic programming and imperative languages, that achieves shared-memory multiprocessor efficiency comparable to the most efficient algorithms proposed in the literature. The algorithm was implemented in the JAM Parlog system, allowing automatic distribution of fine-grained parallel tasks. Distribution was adequately handled by the task scheduler built into the runtime environment and did not require manual processor assignment for peak distribution. The exploitation of dynamic workload distribution for these graph-searching algorithms is a major step forward. Traditional parallel paradigms require programmers to assign work distribution statically, preventing running programs from adapting to data irregularities.

The algorithm was also implemented in C, requiring explicit control of two critical control parameters: search greediness and partition granularity. We argued that because C is more efficient than Parlog, its ratio of computation to communication costs is lower, and therefore greediness and too-fine partitioning degrade performance in greater proportion than they do in Parlog.

Other than the obvious need for an efficient object-code compiler for concurrent logic programming languages, there is still a great deal of experimentation needed in the area of data structures. The lack of, or inefficient implementation of, constant-access destructible arrays hampers many logic programming implementations. Future research should analyze the characteristics of parallel graph-searching algorithms given the use of efficient destructive arrays.

Additional experimentation with partitioning functions is needed. The development and characterization of *dynamic* vertex-to-partition mapping schemes that are not susceptible to hot spots is interesting, as well as more sophisticated static mappings. We are examining a static scheme based on characterizing each node by its potential "hotspottedness," for instance computed by the variance of its edge costs. Nodes would be partitioned, in granules, allocating a fair share of total "hotspottedness" to each partition.

Acknowledgements

The authors thank the referees for suggestions that led to the strengthening of the paper. E. Tick was supported by an NSF Presidential Young Investigator award, with funding from Sequent Computer Systems Inc. Computer resources were supplied by Sequent, OACIS, and Argonne MCS.

References

[1] T. Chikayama. personal communication, 1989.

[2] K. L. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 84–139. MIT Press, Cambridge MA, 1987.

[3] J. A. Crammond. Scheduling and Variable Assignment in the Parallel Parlog Implementation. In *North American Conference on Logic Programming*, pages 642–657. Austin, MIT Press, October 1990.

[4] J. A. Crammond *et al.* The Parallel Parlog User Manual. Technical report, Dept. of Computing, Imperial College, June 1990.

[5] N. Deo, C. Pang, and R. Lord. Two Parallel Algorithms for the Shortest Path Problems. In *International Conference on Parallel Processing*, pages 244–253, Penn State, August 1980.

[6] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerical Mathematics*, 1:269–271, 1959.

[7] D. B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM*, 24:1–13, 1977.

[8] V. Kumar and V. Singh. Scalability of Parallel Algorithms for the All Pairs Shortest Path Problem: A Summary of Results. In *International Conference on Parallel Processing*, volume 3, pages 136–140, Penn State, August 1990.

[9] E. F. Moore. The Shortest Path Through a Maze. In *Proceedings of the International Symposium on Theory of Switching*, pages 285–292, Cambridge MA, April 1957. Harvard University Press.

[10] R. Paige and C. Kruskal. Parallel Algorithms for Shortest Path Problems. In *International Conference on Parallel Processing*, pages 14–19, Penn State, August 1985.

[11] U. Pape. Implementation and Efficiency of Moore-Algorithms for the Shortest Route Problems. *Math. Programming*, 7:212–222, 1974.

[12] M. Quinn and Y. Yoo. Data Structures for the Efficient Solution of Graph Theoretic Problems on Tightly Coupled MIMD Computers. In *International Conference on Parallel Processing*, pages 431–438, Penn State, August 1984.

[13] V. N. Rao and V. Kumar. Concurrent Insertions and Deletions in a Priority Queue. In *International Conference on Parallel Processing*, Penn State, August 1988.

[14] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *IFIP Working Conference on Parallel Processing*, pages 305–318. Pisa, North Holland, May 1988.

[15] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.

[16] E. Y. Shapiro and C. Mierowsky. Fair, Biased, and Self-Balancing Merge Operators: Their Specification and Implementation in Concurrent Prolog. In *International Symposium on Logic Programming*, pages 83–92, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.

[17] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia PA, 1983.

[18] E. Tick. Parallel Logic Programming Algorithms for the Bestpath Problem. Technical Report TRIE-89-4, University of Tokyo, 1989.

[19] K. Wada and H. Ichiyoshi. A Study of Mapping Locally Message Exchanging Algorithms on a Loosely-Coupled Multiprocessor. Technical Report 587, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, August 1990.

A Algorithms: Source Code

The graphs themselves are not listed in this report because their size is too large. In the following Parlog programs, utility procedure definitions are given *once*, in the program where they first occur. For example, `time/1`, `checksum/3`, `myfunctor/2`, `myarg/3`, etc., are all defined in the first program.

A.1 Non-Partitioned List Boundary and Cost Array

```
% Program: Non-Partitioned List Boundary and Cost Array
%          (quite similar to Dijkstra's original algorithm)
% Authors: P. Adamson and E. Tick
% Date:    February 1 1991
% Queries: ?- go(+#Nodes,+StartNode,-Time).
%          ?- go_check(+#Nodes,+StartNode,-CheckSum,-Time).
%          ?- go_stats(+#Nodes,+StartNode,-Statistics).

mode go(?,?,^).
go(N,Start,T) <- Start < N :
    time(_)                    &
    work(N,Start,A)            &
    time(T).

mode go_check(?,?,^,^).
go_check(N,Start,S,T) <- Start < N :
    time(_)                    &
    work(N,Start,A),           &
    time(T),                   &
    checksum(N,A,S).

mode go_stats(?,?,^).
go_stats(N, Start, Stats) <- Start < N :
    statistics(suspensions,S1) &
    statistics(calls,C1)       &
    go(N,Start,_)              &
    statistics(calls,C2)       &
    statistics(suspensions,S2) &
    Cf is C2 - C1,
    Sf is S2 - S1,
    Stats = [Cf, Sf].

mode work(?,?,^).
work(N,Start,A) <-
    myfunctor(A,N) &
    find([a(Start,0,none)],A).

mode find(?,?).
find([a(Node,Cost,Prev)|Boundary],Array) <-
     myarg(Node,Array,V,Synch),
     find1(Synch,V,Node,Prev,Cost,Boundary,Array).
find([],_).

mode find1(?,?,?,?,?,?,?).
```

```
find1([],V,Node,Prev,Cost,Boundary,Array) <- var(V) :
    V = Cost-Prev,
    myedge(Node,Edges),
    insertall(Edges,Cost,Node,Boundary,Array).
find1([],V,_,_,_,Boundary,Array) <- nonvar(V) :
    find(Boundary,Array).

mode insertall(?,?,?,?,?).
insertall([Edge*Delta|Es],Cost,Node,Boundary,Array) <-
    myarg(Edge,Array,V,Synch),
    insertall1(Synch,V,Delta,Edge,Es,Cost,Node,Boundary,Array).
insertall([],_,_,Boundary,Array) <-
    find(Boundary,Array).

mode insertall1(?,?,?,?,?,?,?,?,?).
insertall1([],V,Delta,Edge,Es,Cost,Node,Boundary,Array) <- var(V) :
    NewCost is Cost + Delta,
    list_in(a(Edge,NewCost,Node),Boundary,NewBoundary),
    insertall(Es,Cost,Node,NewBoundary,Array).
insertall1([],V,_,_,Es,Cost,Node,Boundary,Array) <- nonvar(V) :
    insertall(Es,Cost,Node,Boundary,Array).

mode list_in(?,?,^).
list_in(a(E1,C1,P1),[a(E2,C2,P2)|Rest],A)<- E1 =\= E2, C1 > C2 :
    A = [a(E2,C2,P2)|As],
    list_in(a(E1,C1,P1),Rest,As).
list_in(a(E1,C1,P1),[a(E2,C2,P2)|Rest],A)<- C1 =< C2 :
    A = [a(E1,C1,P1),a(E2,C2,P2)|Rest].
list_in(a(E,C1,_),[a(E,C2,P2)|Rest],A)<- C1 >= C2 :
    A = [a(E,C2,P2)|Rest].
list_in(Item,[],A)<- A = [Item].

% big arrays...
mode myfunctor(^,?).
myfunctor(V,Arity) <-
    H is Arity // 255,
    M is Arity mod 255,
    H1 is H+1,
    functor(V,t,H1),
    loop(H,V)&
    mysub(M,H1,V).

mode mysub(?,?,?).
mysub(M,H1,V) <- M>0 : functor(G,t,M), arg(H1,V,G).
mysub(0,_,_).

mode loop(?,?).
loop(K,V) <- K > 0 :
    K1 is K-1,
    functor(G,t,255),
    arg(K,V,G),
    loop(K1,V).
loop(0,_).

mode myarg(?,?,^,^).
myarg(I,A,E,Synch) <-
```

```
    H is ((I-1)// 255)+1,
    M is ((I-1) mod 255)+1,
    arg(H,A,G),
    arg(M,G,E) & Synch = [].

% checksum is quick way to verify if solution is correct...
mode checksum(?,?,^).
checksum(N,L,Sum) <- checksum(N,L,0,Sum).

mode checksum(?,?,?,^).
checksum(K,L,P,Sum) <- K>0 :
    K1 is K-1,
    myarg(K,L,Item,_),
    cont2(Item,P,K1,L,Sum).
checksum(0,_,Sum,Final) <- Final = Sum.

mode cont2(?,?,?,^).
cont2(C-_,P,K1,L,Sum) <-
    P1 is P+C,
    checksum(K1,L,P1,Sum).

% this hack is to avoid a compiler bug wherein edge/2
% cannot be compiled in one module (too many facts)
mode myedge(?,^).
myedge(X,Y) <-          X < 200 : g0edge(X,Y).
myedge(X,Y) <- 199 < X, X < 400 : g1edge(X,Y).
myedge(X,Y) <- 399 < X, X < 600 : g2edge(X,Y).
myedge(X,Y) <- 599 < X, X < 800 : g3edge(X,Y).
myedge(X,Y) <- 799 < X, X <1000 : g4edge(X,Y).
myedge(X,Y) <- 999 < X, X <1238 : g5edge(X,Y).

init <- load(g0) & load(g1) & load(g2) &
        load(g3) & load(g4) & load(g5).

time(T) <- statistics(runtime,[_|T]).
```

A.2 Non-Partitioned List Boundary and Cost Tree

```
% Program: Non-Partitioned List Boundary and Cost Tree
%          (quite similar to Dijkstra's original algorithm)
% Author:  P. Adamson and E. Tick
% Date:    February 1 1991

mode work(?,?,^).
work(N,Start,Table) <-
    init(N,Table),
    find([a(Start,0,0)],Table).

mode find(?,?).
find([a(Node,Cost,Prev)|Boundary],Table) <-
    lookup(Node,Table,V,Synch,NewTable),
    find1(Synch,V,Node,Prev,Cost,Boundary,NewTable).
find([],_).

mode find1(?,?,?,?,?,?,?).
```

```
find1([],V,Node,Prev,Cost,Boundary,Table) <- var(V) :
    V = Cost-Prev,
    myedge(Node,Edges),
    insertall(Edges,Cost,Node,Boundary,Table).
find1([],V,_,_,_,Boundary,Table) <- nonvar(V) :
    find(Boundary,Table).

mode insertall(?,?,?,?,?).
insertall([Edge*Delta|Es],Cost,Node,Boundary,Table) <-
    lookup(Edge,Table,V,Synch,NewTable),
    insertall1(Synch,V,Delta,Edge,Es,Cost,Node,Boundary,NewTable).
insertall([],_,_,Boundary,Table) <-
    find(Boundary,Table).

mode insertall1(?,?,?,?,?,?,?,?,?).
insertall1([],V,Delta,Edge,Es,Cost,Node,Boundary,Table) <- var(V) :
    NewCost is Cost + Delta,
    list_in(a(Edge,NewCost,Node),Boundary,NewBoundary),
    insertall(Es,Cost,Node,NewBoundary,Table).
insertall1([],V,_,_,Es,Cost,Node,Boundary,Table) <- nonvar(V) :
    insertall(Es,Cost,Node,Boundary,Table).

mode list_in(?,?,^).
list_in(a(E1,C1,P1),[a(E2,C2,P2)|Rest],A) <- E1 =\= E2, C1 > C2 :
    A = [a(E2,C2,P2)|As],
    list_in(a(E1,C1,P1),Rest,As).
list_in(a(E1,C1,P1),[a(E2,C2,P2)|Rest],A) <- C1 =< C2 :
    A = [a(E1,C1,P1),a(E2,C2,P2)|Rest].
list_in(a(E,C1,_),[a(E,C2,P2)|Rest],A) <- C1 >= C2 :
    A = [a(E,C2,P2)|Rest].
list_in(Item,[],A) <- A = [Item].

% cost table is a binary tree...
mode init(?,^).
init(N,Tree) <- N1 is N+1, make(1,N1,Tree).

mode make(?,?,^).
make(E1,E2,Tree) <- E1 =\= E2 :
    R  is E2-E1,
    H  is R // 2,
    M  is H+E1,
    M1 is M+1,
    Tree = t(M,_,Left,Right),
    make(E1,M,Left),
    make(M1,E2,Right).
make(E,E,_).

mode lookup(?,?,^,^,^).
lookup(Key,t(Key,Value,Left,Right),V,S,T) <-
    V = Value & S = [],        % '&' is very important!
    T = t(Key,Value,Left,Right).
lookup(Key,t(Key1,Old,Left,Right),V,S,T) <- Key < Key1 :
    T = t(Key1,Old,NewLeft,Right),
    lookup(Key,Left,V,S,NewLeft).
lookup(Key,t(Key1,Old,Left,Right),V,S,T) <- Key > Key1 :
    T = t(Key1,Old,Left,NewRight),
```

```
    lookup(Key,Right,V,S,NewRight).
```

A.3 Non-Partitioned Static-Heap Boundary and Cost Array

```
% Program: Non-Partitioned Static-Heap Boundary and Cost Array
%          (quite similar to Johnson-Dijkstra algorithm)
% Authors: P. Adamson and E. Tick
% Date:    February 1 1991

mode work(?,?,^).
work(N,Start,A) <-
    myfunctor(A,N),
    heap_in(complete(void),a(Start,0,none),Heap),
    find(N,Heap,A).

mode find(?,?,?).
find(M,Heap,Array) <- M > 0 :
    heap_out(Heap,Root,NewHeap),
    cont(Root, Array, M, NewHeap).
find(0,_,_).

mode cont(?,?,?,^).
cont(a(Node,Cost,Prev), Array, M, NewHeap) <-
    myarg(Node,Array,V,Synch),
    find1(Synch,V,Node,Prev,Cost,M,NewHeap,Array).

mode find1(?,?,?,?,?,?,?,?).
find1([],V,Node,Prev,Cost,M,NewHeap,Array) <- var(V) :
    M1 is M-1,
    V = Prev-Cost,
    myedge(Node,Edges),
    insertall(Edges,Cost,Node,NewHeap,Array,M1).
find1([],V,_,_,_,M,NewHeap,Array) <- nonvar(V) :
    find(M,NewHeap,Array).

mode insertall(?,?,?,?,?,?).
insertall([Edge*Delta|Es],Cost,Node,Heap,Array,M) <-
    myarg(Edge,Array,V,Synch),
    insertall1(Synch,V,Delta,Edge,Es,Cost,Node,Heap,Array,M).
insertall([],_,_,Heap,Array,M) <-
    find(M,Heap,Array).

mode insertall1(?,?,?,?,?,?,?,?,?,?).
insertall1([],V,Delta,Edge,Es,Cost,Node,Heap,Array,M) <- var(V) :
    NewCost is Cost + Delta,
    heap_in(Heap,a(Edge,NewCost,Node),NewHeap),
    insertall(Es,Cost,Node,NewHeap,Array,M).
insertall1([],V,_,_,Es,Cost,Node,Heap,Array,M) <- nonvar(V) :
    insertall(   Es,Cost,Node,Heap,Array,M).

%-----------------------------------------------------------------
% heap manager interface:
%       fast_heap_in(+Heap, +Data, -NewHeap)
%           insert Data into Heap creating NewHeap
%
```

```
%       fast_heap_out(+Heap, -Data, -NewHeap)
%           remove root Data from Heap creating NewHeap
%-----------------------------------------------------------------
mode heap_in(?,?,^).
heap_in(partial(L,R),Data,FinalHeap) <-
    FinalHeap = complete(NewHeap),
    heapify(node(L,Data,R),NewHeap).
heap_in(complete(Heap),Data,FinalHeap) <-
    FinalHeap = complete(NewHeap),
    h_in(Heap,Data,NewHeap).

mode switch(?,?,?,?,^,^).
switch(C0,C,A,A0,Top,Bot) <- C0 =< C : Top = A0, Bot = A.
switch(C0,C,A,A0,Top,Bot) <- C0  > C : Top = A,  Bot = A0.

mode h_in(?,?,^).
h_in(void,Data,Out) <- Out = node(void,Data,void).
h_in(node(Left,a(N0,C0,P0),void),a(N,C,P),Out) <-
    Out = node(Left,Top,node(void,Bot,void)),
    switch(C0,C,a(N,C,P),a(N0,C0,P0),Top,Bot).
h_in(node(void,a(N0,C0,P0),Right),a(N,C,P),Out) <-
    Out = node(node(void,Bot,void),Top,Right),
    switch(C0,C,a(N,C,P),a(N0,C0,P0),Top,Bot);
h_in(node(Left,a(N0,C0,P0),Right),a(N,C,P),Out) <-
    Even is (C0+C) mod 2,
    switch(C,C0,a(N0,C0,P0),a(N,C,P),Top,Bot),
    heap_in1(Even,Left,Right,Top,Bot,Out).

mode heap_in1(?,?,?,?,?,^).
heap_in1(0,Left,Right,Top,Bot,Out) <-
    Out = node(Left,Top,NewRight),
    h_in(Right,Bot,NewRight).
heap_in1(1,Left,Right,Top,Bot,Out) <-
    Out = node(NewLeft,Top,Right),
    h_in(Left,Bot,NewLeft).

mode heap_out(?,^,^).
heap_out(complete(node(L,Data,R)),Item,New) <-
    Item = Data, New = partial(L,R).
heap_out(partial(L,R),Data,New) <-
    heapify(node(L,a(leaf,999999,leaf),R),Out),
    h_out(Out,Data,New).

mode h_out(?,^,^).
h_out(node(L1,Data,R1),Item,New) <-
    Item = Data, New = partial(L1,R1).

mode exchange(?,?,?,?,^,?,^).
exchange(C0,C1,Heap,_,_,_,   New) <- C0 =< C1 : New = Heap.
exchange(C0,C1,_,Root,Sub,In,New) <-  C0 > C1 :
    New = Root,
    heapify(In,Sub).

mode heapify(?,^).
heapify(node(node(L1,a(N1,C1,P1),R1),a(N0,C0,P0),void),New) <-
    exchange(C0,C1,node(node(L1,a(N1,C1,P1),R1),a(N0,C0,P0),void),
```

```
        node(Left,a(N1,C1,P1),void),Left,node(L1,a(N0,C0,P0),R1),New).

heapify(node(void,a(N0,C0,P0),node(L2,a(N2,C2,P2),R2)),New) <-
    exchange(C0,C2,node(void,a(N0,C0,P0),node(L2,a(N2,C2,P2),R2)),
        node(void,a(N2,C2,P2),Right),Right,node(L2,a(N0,C0,P0),R2),New).

heapify(node(node(L1,a(N1,C1,P1),R1),
             a(N0,C0,P0),
             node(L2,a(N2,C2,P2),R2)),New) <-
    C0 > C1, C0 =< C2 :
    New = node(NewLeft,a(N1,C1,P1),node(L2,a(N2,C2,P2),R2)),
    heapify(node(L1,a(N0,C0,P0),R1),NewLeft).

heapify(node(node(L1,a(N1,C1,P1),R1),
             a(N0,C0,P0),
             node(L2,a(N2,C2,P2),R2)),New) <-
    C0 > C1, C1 =< C2 :
    New = node(NewLeft,a(N1,C1,P1),node(L2,a(N2,C2,P2),R2)),
    heapify(node(L1,a(N0,C0,P0),R1),NewLeft).

heapify(node(node(L1,a(N1,C1,P1),R1),
             a(N0,C0,P0),
             node(L2,a(N2,C2,P2),R2)),New) <-
    C0 > C2, C0 =< C1 :
    New = node(node(L1,a(N1,C1,P1),R1),a(N2,C2,P2),NewRight),
    heapify(node(L2,a(N0,C0,P0),R2),NewRight).

heapify(node(node(L1,a(N1,C1,P1),R1),
             a(N0,C0,P0),
             node(L2,a(N2,C2,P2),R2)),New) <-
    C0 > C2, C2 < C1 :
    New = node(node(L1,a(N1,C1,P1),R1),a(N2,C2,P2),NewRight),
    heapify(node(L2,a(N0,C0,P0),R2),NewRight);

heapify(In,New) <- New = In.
```

A.4 Non-Partitioned Active-Heap Boundary and Cost Array

```
% Program: Non-Partitioned Active-Heap Boundary and Cost Array
%          (quite similar to Johnson-Dijkstra algorithm)
% Authors: P. Adamson and E. Tick
% Date:    February 1 1991

mode work(?,?,^).
work(N,Start,Table) <-
    myfunctor(Table, N),
    node(Ms, a(Start,0,none)),
    find(N, Ms, Table).

mode find(?,?,?).
find(M, HeapStream, Table) <- M > 0 :
    HeapStream = [h_out(E)|Hs],
    find2(Hs, E, M, Table).
find(0, HeapStream, _) <- HeapStream = [].
```

```
mode find2(?,?,?,?).
find2(Hs, a(Node,Cost,Prev), M, Table) <-
    myarg(Node,Table,V,Synch),
    find1(Synch,V, Node,Cost,Prev, M,Hs,Table).

mode find1(?,?, ?,?,?, ?,?,?).
find1([],V, _,_,_, M,Hs,Table) <- nonvar(V) :
    find(M,Hs,Table).
find1([],V, Node,Cost, Prev,M,Hs,Table) <- var(V) :
    M1 is M - 1,
    V = Cost-Prev,
    myedge(Node,Edges),
    insertall(Edges,Cost,Node,Hs,Table,M1).

mode insertall(?,?,?,?,?,?).
insertall([],_,_,Hs,Table,M) <-
    find(M,Hs,Table).
insertall([Edge*Delta|Es],Cost,Node,Hs,Table,M) <-
    myarg(Edge,Table,V,Synch),
    insertall1(Synch,V,Delta,Edge,Es,Cost,Node,Hs,Table,M).

mode insertall1(?,?,?,?,?,?,?,^,?,?).
insertall1([],V,_,_,Es,Cost,Node,Hs,Table,M) <- nonvar(V) :
    insertall(Es,Cost,Node,Hs,Table,M).
insertall1([],V,Delta,Edge,Es,Cost,Node,Hs,Table,M) <- var(V) :
    NewCost is Cost + Delta,
    Hs = [h_in(a(Edge,NewCost,Node))|Hs1],
    insertall(Es,Cost,Node,Hs1,Table,M).

% node/2, partial/1 have NO children!
mode partial(?).
partial([]).
partial([h_in(Vin)|Ms]) <- node(Ms,Vin).

mode node(?,?).
node([],                _).
node([cost(X)     |Ms],V) <- X = V, node(Ms,V).
node([h_out(X)    |Ms],V) <- X = V, partial(Ms).
node([replace(Vin)|Ms],_) <- node(Ms,Vin).
node([h_in(Vin)   |Ms],V) <-
    switch(Vin, V, Vpar, Vchild),
    node(Ms, Vpar, Ls),
    node(Ls, Vchild).

% node/3, partial/2 have ONE child...
mode partial(?,^).
partial([],           Ls) <- Ls = [].
partial([h_out(V)|Ms],L0) <-
    L0 = [cost(Vc)|L1],
    compare(Vc, a(leaf,999999,leaf), V, L1, L2),
    partial(Ms, L2).
partial([h_in(Vin)|Ms],L0) <-
    L0 = [cost(Vc)|L1],
    compare(Vc, Vin, V, L1, L2),
    node(Ms, V, L2).
```

```
mode node(?,?,^).
node([],                _,Ls) <- Ls = [].
node([cost(X)     |Ms],V,Ls) <- X = V, node(Ms,V,Ls).
node([h_out(X)    |Ms],V,Ls) <- X = V, partial(Ms,Ls).
node([replace(Vin)|Ms],_,L0) <-
    L0 = [cost(Vc)|L1],
    compare(Vc, Vin, V, L1, L2),
    node(Ms, V, L2).
node([h_in(Vin)|Ms], V, Ls) <-
    switch(Vin, V, Vpar, Vchild),
    node(Ms, Vpar, Ls, Rs),
    node(Rs, Vchild).

% node/4, partial/3 have TWO children:
mode partial(?,^,^).
partial([],           L,R) <- L=[], R=[].
partial([h_out(V)|Ms],L,R) <-
    L = [cost(Vl)|L1], R = [cost(Vr)|R1],
    compare(Vl, Vr, a(leaf,999999,leaf), V, L1, L2, R1, R2),
    partial(Ms, L2, R2).
partial([h_in(Vin)|Ms],L,R) <-
    L = [cost(Vl)|L1], R = [cost(Vr)|R1],
    compare(Vl, Vr, Vin, V, L1, L2, R1, R2),
    node(Ms, V, L2, R2).

mode node(?,?,^,^).
node([],                _,L,R) <- L = [], R=[].
node([cost(X)     |Ms],V,L,R) <- X = V, node(Ms,V,L,R).
node([h_out(X)    |Ms],V,L,R) <- X = V, partial(Ms,L,R).
node([h_in(Vin)   |Ms],V,L,R) <- continue(Vin,V,Ms,L,R).
node([replace(Vin)|Ms],_,L,R) <-
    L = [cost(Vl)|L1], R = [cost(Vr)|R1],
    compare(Vl, Vr, Vin, V, L1, L2, R1, R2),
    node(Ms, V, L2, R2).

mode continue(?,?,?,?,?).
continue(a(N0,C0,P0),a(N1,C1,P1),Ms,L,R) <-
    Even is (C0+C1) mod 2,
    intoheap(Even, a(N0,C0,P0), a(N1,C1,P1), V, L, R, Ls, Rs),
    node(Ms, V, Ls, Rs).

mode intoheap(?,?,?,?,?,?,^,?).
intoheap(0, Ain, Aold, Anew, L, R, Ls, Rs) <-
    Ls = L,
    R = [h_in(Anext)|Rs],
    switch(Ain, Aold, Anew, Anext).
intoheap(1, Ain, Aold, Anew, L, R, Ls, Rs) <-
    Rs = R,
    L = [h_in(Anext)|Ls],
    switch(Ain, Aold, Anew, Anext).

% compare(+,+,-,+,-): for a parent with a single child...
mode compare(?,?,^,?,^).
compare(a(_,Cr,_), a(Lin,Cin,Rin), Aout, R, Rs) <- Cin =< Cr :
    Aout = a(Lin,Cin,Rin), Rs = R.
compare(a(Lr,Cr,Rr), a(Lin,Cin,Rin), Aout, R, Rs) <- Cin > Cr :
```

```
    Aout = a(Lr,Cr,Rr), R = [replace(a(Lin,Cin,Rin))|Rs].

% compare(+,+,+,-,+,-,+,-): for a parent with two children...
% parent cost is less than both child costs: stop heapification
mode compare(?,?,?,^,?,^,?,^).
compare(a(_,Cl,_), a(_,Cr,_), a(Lin,Cin,Rin), Aout, L, Ls, R, Rs) <-
    Cin =< Cl, Cin =< Cr :
    Aout = a(Lin,Cin,Rin), Rs = R, Ls = L.
% parent cost exchanged with left child: heapify down left branch
compare(a(Ll,Cl,Rl),a(_,Cr,_), a(Lin,Cin,Rin), Aout, L, Ls, R, Rs) <-
    Cl =< Cr, Cl < Cin :
    Aout = a(Ll,Cl,Rl), L = [replace(a(Lin,Cin,Rin))|Ls], Rs = R.
% parent cost exchanged with right child: heapify down right branch
compare(a(_,Cl,_), a(Lr,Cr,Rr), a(Lin,Cin,Rin), Aout, L, Ls, R, Rs) <-
    Cl > Cr, Cl < Cin :
    Aout = a(Lr,Cr,Rr), R = [replace(a(Lin,Cin,Rin))|Rs], Ls = L.
% parent cost exchanged with right child: heapify down right branch
compare(a(_,Cl,_), a(Lr,Cr,Rr), a(Lin,Cin,Rin), Aout, L, Ls, R, Rs) <-
    Cr =< Cl, Cr < Cin :
    Aout = a(Lr,Cr,Rr), R = [replace(a(Lin,Cin,Rin))|Rs], Ls = L.
% parent cost exchanged with left child: heapify down left branch
compare(a(Ll,Cl,Rl), a(_,Cr,_), a(Lin,Cin,Rin), Aout, L, Ls, R, Rs) <-
    Cr > Cl, Cr < Cin :
    Aout = a(Ll,Cl,Rl), L = [replace(a(Lin,Cin,Rin))|Ls], Rs = R.

mode switch(?,?,^,^).
switch(a(L0,C0,R0),a(L1,C1,R1),Top,Bot) <- C0 =< C1 :
    Top = a(L0,C0,R0), Bot = a(L1,C1,R1).
switch(a(L0,C0,R0),a(L1,C1,R1),Top,Bot) <- C0 > C1 :
    Top = a(L1,C1,R1), Bot = a(L0,C0,R0).
```

A.5 Partitioned List Boundary and Cost Tree

```
% Program: Partitioned List Boundary and Cost Tree
% Author:  P. Adamson and E. Tick
% Date:    February 1 1991
% Queries: ?- go(+#Partitions,+#Nodes,+StartNode,-Time).
%          ?- go_check(+#Partitions,+#Nodes,+StartNode,-CheckSum,-Time).
%          ?- go_stats(+#Partitions,+#Nodes,+StartNode,-Statistics).

mode go(?,?,?,^).
go(NumParts,N,Start,T) <- Start < N :
    time(_)                    &
    work(NumParts,N,Start,_)   &
    time(T).

mode go_check(?,?,?,^).
go_check(NumParts,N,Start,S,T) <- Start < N :
    time(_)                    &
    work(NumParts,N,Start,Table) &
    time(T)                    &
    checksum(N,Table,S).

mode go_stats(?,?,?,^).
go_stats(NumParts,N,Start,Stats) <- Start < N :
```

```
    statistics(suspensions,S1)   &
    statistics(calls,C1)         &
    work(NumParts,N,Start,_)     &
    statistics(calls,C2)         &
    statistics(suspensions,S2)   &
    Cf is C2 - C1,
    Sf is S2 - S1,
    Stats = [Cf, Sf].

mode work(?,?,?,^).
work(NumParts, NumNodes, Start, Table) <-
    make_tables(NumParts, NumNodes, Tables),
    build_streams(NumParts, Extracts, Inserts, MergedStreams),
    spawn(0,Start,NumParts,Tables,Extracts,Inserts,
                                MergedStreams,_,FTables-[]),
    join_tables(NumNodes,FTables,Table).

mode spawn(?,?,?,?,?,?,?,?,?).
spawn(K,_,K,_,_,_,_,_,_,F0-F1) <- F0 = F1.
spawn(K,Start,NumPs,[Table|Rest],[Ex|Es],[Ins|Is],
                            [ToMerge|Ms],D,F0-F2) <-
    K < NumPs :
    K1 is  K+1,
    compute_partition(Start,NumPs,Partition),
    getQueue(Partition,K,Start,Merged,D-done),
    merger(ToMerge,Merged),
    worker(Ex,Table,NumPs,Ins,D,F0-F1),
    spawn(K1,Start,NumPs,Rest,Es,Is,Ms,D,F1-F2).

mode compute_partition(?,?,^).
compute_partition(Id,NumParts,Part) <- Part is Id mod NumParts.

mode worker(?,?,?,?,?,^).
worker(ExStream,Table,NumPs,Outs,D,F) <- var(D) :
    ExStream = [extract(V)|More],
    check1(V,More,Table,NumPs,Outs,D,F).
worker(Ex,Table,_,Outs,done,F0-F1) <-
    F0=[Table|F1], Ex = [],
    closeStreams(Outs).

mode check1(?,?,?,?,?,?,^).
check1(a(Node,Cost,Prev,D),Id,T,N,A,Dvar,F) <- Cost \== 99999 :
    lookup(Node,Cost-Prev,T,NewT,Flag),
    check2(Flag,Node,Cost,Prev,D,Id,NewT,N,A,Dvar,F).
check1(a(_,99999,_,_),Id,T,N,A,Dvar,F) <-
    worker(Id,T,N,A,Dvar,F).

mode check2(?,?,?,?,?,?,?,?,?,?,?).
check2(no,_,_,_,D0-D1,Id,T,N,A,D,F) <-
    D0=D1,
    worker(Id,T,N,A,D,F).
check2(yes,Node,Cost,Prev,D,Id,T,N,A,Dvar,F) <-
    myedge(Node,Edges),
    dispatch(Edges,Node,Cost,Prev,D,Id,T,N,A,Dvar,F).

mode dispatch(?,?,?,?,?,?,?,?,?,?, ?).
```

```
dispatch([],_,_,_,D0-D1,Id,Table,N,A,D,F) <- D0=D1,
    worker(Id,Table,N,A,D,F).
dispatch([Prev*_|Rest],Node,Cost,Prev,D,Id,T,N,A,Dvar,F) <-
    dispatch(Rest,Node,Cost,Prev,D,Id,T,N,A,Dvar,F).
dispatch([Node*Delta|Rest],LastNode,Cost,Prev,D0-D2,Id,
                                    T,NumParts,A,D,F) <-
    Node =\= Prev :
    NewCost is Cost + Delta,
    compute_partition(Node, NumParts, Partition),
    mysend(Partition,insert(a(Node,NewCost,LastNode,D0-D1)),A,NewNet),
    dispatch(Rest,LastNode,Cost,Prev,D1-D2,Id,T,NumParts,NewNet,D,F).

mode mysend(?,?,?,^).
mysend(0,Z,[A              |Ss],S) <- A=[Z|Zs], S=[             Zs|Ss].
mysend(1,Z,[A,B            |Ss],S) <- B=[Z|Zs], S=[A,           Zs|Ss].
mysend(2,Z,[A,B,C          |Ss],S) <- C=[Z|Zs], S=[A,B,         Zs|Ss].
mysend(3,Z,[A,B,C,D        |Ss],S) <- D=[Z|Zs], S=[A,B,C,       Zs|Ss].
mysend(4,Z,[A,B,C,D,E      |Ss],S) <- E=[Z|Zs], S=[A,B,C,D,     Zs|Ss].
mysend(5,Z,[A,B,C,D,E,F    |Ss],S) <- F=[Z|Zs], S=[A,B,C,D,E,   Zs|Ss].
mysend(6,Z,[A,B,C,D,E,F,G  |Ss],S) <- G=[Z|Zs], S=[A,B,C,D,E,F, Zs|Ss].
mysend(7,Z,[A,B,C,D,E,F,G,H|Ss],S) <- H=[Z|Zs], S=[A,B,C,D,E,F,G,Zs|Ss].
mysend(8,Z,[A,B,C,D,E,F,G,H,I |Ss],S) <- I=[Z|Zs],
        S=[A,B,C,D,E,F,G,H,Zs|Ss].
mysend(9,Z,[A,B,C,D,E,F,G,H,I,J |Ss],S) <- J=[Z|Zs],
        S=[A,B,C,D,E,F,G,H,I,Zs|Ss].
mysend(10,Z,[A,B,C,D,E,F,G,H,I,J,K |Ss],S) <- K=[Z|Zs],
         S=[A,B,C,D,E,F,G,H,I,J,Zs|Ss].
mysend(11,Z,[A,B,C,D,E,F,G,H,I,J,K,L |Ss],S) <- L=[Z|Zs],
         S=[A,B,C,D,E,F,G,H,I,J,K,Zs|Ss].
mysend(12,Z,[A,B,C,D,E,F,G,H,I,J,K,L,M |Ss],S) <- M=[Z|Zs],
         S=[A,B,C,D,E,F,G,H,I,J,K,L,Zs|Ss].
mysend(13,Z,[A,B,C,D,E,F,G,H,I,J,K,L,M,N |Ss],S) <- N=[Z|Zs],
         S=[A,B,C,D,E,F,G,H,I,J,K,L,M,Zs|Ss].
mysend(14,Z,[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O |Ss],S) <- O=[Z|Zs],
         S=[A,B,C,D,E,F,G,H,I,J,K,L,M,N,Zs|Ss].
mysend(15,Z,[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P |Ss],S) <- P=[Z|Zs],
         S=[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,Zs|Ss].
mysend(16,Z,[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q |Ss],S) <- Q=[Z|Zs],
         S=[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Zs|Ss].
mysend(17,Z,[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R |Ss],S) <- R=[Z|Zs],
         S=[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,Zs|Ss].
mysend(18,Z,[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,Sp|Ss],S) <- Sp=[Z|Zs],
         S=[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,Zs|Ss].
mysend(19,Z,[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,Sp,T |Ss],S) <-
         T=[Z|Zs],
         S=[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,Sp,Zs|Ss].
mysend(19,Z,[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,Sp,T,U |Ss],S) <-
         U=[Z|Zs],
         S=[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,Sp,T,Zs|Ss].
mysend(20,Z,[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,Sp,T,U,V |Ss],S) <-
         V=[Z|Zs],
         S=[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,Sp,T,U,Zs|Ss].
mysend(21,Z,[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,Sp,T,U,V,W |Ss],S) <-
         W=[Z|Zs],
         S=[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,Sp,T,U,V,Zs|Ss].
```

```
mode build_streams(?,^,^,^).
build_streams(N, Quicks, AllStreams, Merges) <-
    make_slots(N, Slist),
    make_matrix(N, Slist, AllStreams, Merges, Quicks).

mode make_matrix(?,^,^,^).
make_matrix(0, S, A, M, Q) <-
    A = S, M = [], Q = [].
make_matrix(N, Form, A, M, Q) <- N > 0:
    N1 is N - 1,
    fill_slots(Form, A1, M1, Q1),
    M = [M1|Ms], Q = [Q1|Qs],
    make_matrix(N1, A1, A, Ms, Qs).

mode fill_slots(?,^,^,^).
fill_slots([], A, M, Q) <-
    A = [],
    M = [merge(Hnew)],
    Q = Hnew.
fill_slots([H|HRest], [A|ARest], M, Q) <-
    append(H,[HNew],A),
    M = [merge(HNew)|MRest],
    fill_slots(HRest, ARest, MRest,Q).

mode make_slots(?,^).
make_slots(0,[]).
make_slots(N,A) <- N > 0:
    N1 is N - 1,
    A = [[]|As],
    make_slots(N1,As).

mode append(?,?,^).
append([],X,Y) <- Y = X.
append([H|T],Y,Z) <- Z = [H|Zs], append(T,Y,Zs).

mode getQueue(?,?,?,?,?).
getQueue(Part,Part,Start,In,D) <- node(In,[a(Start,0,0,D)],[]).
getQueue(P,N,_,In,_) <- P =\= N : node(In,[],              []).

mode closeStreams(?).
closeStreams([]).
closeStreams([S|Ss]) <- S=[kill], closeStreams(Ss).

mode node(?,?,?).
node([],_,[]).
node([kill|Ks],D,[]) <- node(Ks,D,[]).
node([kill|Ks],L,[Q|Qs]) <-
    Q = a(leaf,99999,leaf,_),
    node(Ks,L,Qs).
node([extract(X)|Ms],[],Q) <- node(Ms,[],[X|Q]).
node([extract(X)|Ms],[H|T],Q) <- X = H, node(Ms,T,Q).
node([insert(A)|Ms],T,[]) <-
    list_in(A,T,New),
    node(Ms,New,[]).
node([insert(A)|Ms],T,[Q|Qs]) <-
    Q = A,
```

```
    node(Ms,T,Qs).

mode list_in(?, ?,^).
list_in(a(E1,C1,P1,D1),[a(E2,C2,P2,D2)|Rest],A) <- E1 \== E2, C1 > C2 :
    A = [a(E2,C2,P2,D2)|As],
    list_in(a(E1,C1,P1,D1),Rest,As).
list_in(a(E1,C1,P1,D1),[a(E2,C2,P2,D2)|Rest],A) <- C1 =< C2 :
    A = [a(E1,C1,P1,D1),a(E2,C2,P2,D2)|Rest].
list_in(a(E,C1,_,D0-D1),[a(E,C2,P2,D2)|Rest],A) <- C1 >= C2 :
    D0 = D1,
    A = [a(E,C2,P2,D2)|Rest].
list_in(X,[],A) <- A = [X].

mode make_tables(?,?,^).
make_tables(NumParts, NumNodes, Tables) <-
    TreeSize is (NumNodes // NumParts) + 2 :
    make_trees(0,TreeSize,NumParts,0,Tables).

mode make_trees(?,?,?,?,^).
make_trees(_,_,NumParts,NumParts,Trees) <- Trees=[].
make_trees(Start,Treesize,NumParts,Offset,Trees) <- NumParts > Offset :
    Next is Offset + 1,
    make_tree(Start,Treesize,NumParts,Offset,Tree),
    Trees = [Tree|Rest],
    make_trees(Start,Treesize,NumParts,Next,Rest).

mode make_tree(?,?,?,?,^).
make_tree(E,E,NumParts,Offset,X) <-
    N is (E*NumParts)+Offset,
    X = t(N,10000-_,end,end).
make_tree(S,E,_,_,X) <- S > E : X=end.
make_tree(Start,End,NumParts,Offset,Tree) <- Start < End :
    Size is End - Start,
    Split is Size // 2,
    Key is Start + Split,
    M1 is Key - 1,
    M2 is Key + 1,
    Kval is (Key*NumParts) + Offset,
    Tree = t(Kval,10000-_,Left,Right),
    make_tree(Start,M1,NumParts,Offset,Left),
    make_tree(M2,End,NumParts,Offset,Right).

mode lookup(?,?,?,^,^).
lookup(Key,New,t(Key,Old,Left,Right),T,Flag) <-
    T = t(Key,Value,Left,Right),
    insert(New,Old,Value,Flag).
lookup(Key,New,t(Key1,Old,Left,Right),T,Flag) <- Key < Key1 :
    T = t(Key1,Old,NewLeft,Right),
    lookup(Key,New,Left,NewLeft,Flag).
lookup(Key,New,t(Key1,Old,Left,Right),T,Flag) <- Key > Key1 :
    T = t(Key1,Old,Left,NewRight),
    lookup(Key,New,Right,NewRight,Flag).

mode insert(?,?,^,^).
insert(NewCost-NewPrev,OldCost-_,Rvalue,Flag) <- NewCost < OldCost :
    Flag = yes,
```

```
    Rvalue = NewCost-NewPrev.
insert(NewCost-_,OldCost-OldPath,Rvalue,Flag) <- NewCost >= OldCost :
    Flag = no,
    Rvalue = OldCost-OldPath.

mode join_table(?,?,^).
join_tables(Size,Tables,Table) <-
    myfunctor(Table,Size),
    get_data(Size,Table,Tables).

mode get_data(?,?,?).
get_data(_,_,[]).
get_data(S,V,[H|T]) <-
    read_tree(S,H,V),
    get_data(S,V,T).

mode read_tree(?,?,?).
read_tree(_,end,_).
read_tree(Size,t(Node,Data,L,R),Table) <- Node > 0, Node =< Size :
    myarg(Node,Table,V,_),
    V = Data,
    read_tree(Size,L,Table),
    read_tree(Size,R,Table).
read_tree(Size,t(Node,_,L,R),Table) <- Node > Size :
    read_tree(Size,L,Table),
    read_tree(Size,R,Table).
read_tree(Size,t(0,_,L,R),Table) <-
    read_tree(Size,L,Table),
    read_tree(Size,R,Table).

mode merge(?,?,^).
merge([X|Xs], [Y|Ys], Z) <-          Z=[X,Y|Zs], merge(Xs,Ys,Zs).
merge([X|Xs],     Ys, Z) <- var(Ys) : Z=[X  |Zs], merge(Xs,Ys,Zs).
merge(Xs,     [Y|Ys], Z) <- var(Xs) : Z=[Y  |Zs], merge(Xs,Ys,Zs).
merge([],          Y, Z) <-          Z=Y.
merge(X,          [], Z) <-          Z=X.

mode merger(?,^).
merger(Ms,Out) <- merger1(Ms,Ts), cont(Ts,Out).

mode cont(?,^).
cont([merge(S)],Out) <- Out=S ;
cont(Ms,        Out) <- merger(Ms,Out).

mode merger1(?,^).
merger1([],Ts) <- Ts = [].
merger1([merge(S)],T) <- T = [merge(S)].
merger1([merge(S0),merge(S1)|Ms],Out) <-
    Out = [merge(M)|Ts],
    merge(S0,S1,M),
    merger1(Ms,Ts).
```

A.6 Partitioned Active-Heap Boundary and Cost Tree

```
% Program: Partitioned Active-Heap Boundary and Cost Tree
```

```
% Author:  P. Adamson and E. Tick
% Date:    February 1 1991
% Queries: ?- go(+#Partitions,+#Nodes,+StartNode,-Time).
%          ?- go_check(+#Partitions,+#Nodes,+StartNode,-CheckSum,-Time).
%          ?- go_stats(+#Partitions,+#Nodes,+StartNode,-Statistics).
% Notes:   All other code shared with Partitioned List Boundary Program.

mode getQueue(?,?,?,?,?).
getQueue(Part,Part,Start,In,D) <-
    node(In,a(Start,0,0,D)).
getQueue(P,N,_,In,_) <- P =\= N :
    node(In,[]).

% node/2, spartial/1 have NO children!
mode spartial(?,?).
spartial([],                  []).
spartial([extract(Vapor)|Ms], Q) <- spartial(Ms,[Vapor|Q]).
spartial([insert(X)|Ms],      []) <- node(Ms,X).
spartial([insert(X)|Ms], [Q|Qs]) <- Q=X, spartial(Ms,Qs).
spartial([kill|Ks],           []) <- spartial(Ks,[]).
spartial([kill|Ks],       [Q|Qs]) <-
    Q=a(leaf,99999,leaf,_),
    spartial(Ks,Qs).

mode node(?,?).
node([],                 _).
node([kill|T],           V) <- node(T,V).
node([cost(X)      |Ms],V) <- X=V, node(Ms,V).
node([replace(Vin) |Ms],_) <-      node(Ms,Vin).
node([extract(X)   |Ms],V) <- X=V, spartial(Ms,[]).
node([insert(Vin)  |Ms],V) <-
    switch(Vin,V,Vpar,Vchild),
    node(Ms,Vpar,Ls),
    node(Ls,Vchild).

% node/3, partial/2 have ONE child...
mode partial(?,^).
partial([],              Ls) <- Ls = [].
partial([kill       |Ms],Ls) <- partial(Ms,Ls).
partial([extract(V) |Ms],L0) <-
    L0 = [cost(Vc)|L1],
    compare(Vc, a(leaf,999999,leaf,_), V, L1, L2),
    partial(Ms, L2).
partial([insert(Vin)|Ms],L0) <-
    L0 = [cost(Vc)|L1],
    compare(Vc, Vin, V, L1, L2),
    node(Ms, V, L2).

mode node(?,?,^).
node([],                _,Ls) <- Ls=[].
node([kill        |Ms],V,Ls) <- node(Ms,V,Ls).
node([cost(X)     |Ms],V,Ls) <- X=V, node(Ms,V,Ls).
node([extract(X)  |Ms],V,Ls) <- X=V, partial(Ms,Ls).
node([replace(Vin)|Ms],_,L0) <-
    L0 = [cost(Vc)|L1],
    compare(Vc, Vin, V, L1, L2),
```

```
    node(Ms, V, L2).
node([insert(Vin) |Ms],V,Ls) <-
    switch(Vin, V, Vpar, Vchild),
    node(Ms, Vpar, Ls, Rs),
    node(Rs, Vchild).

% node/4, partial/3 have TWO children:
mode partial(?,^,^).
partial([],              L,R) <- L = [], R = [].
partial([kill       |Ms],L,R) <- partial(Ms,L,R).
partial([extract(V) |Ms],L,R) <-
    L = [cost(Vl)|L1],
    R = [cost(Vr)|R1],
    compare(Vl, Vr, a(leaf,999999,leaf,_), V, L1, L2, R1, R2),
    partial(Ms, L2, R2).
partial([insert(Vin)|Ms],L,R) <-
    L = [cost(Vl)|L1],
    R = [cost(Vr)|R1],
    compare(Vl, Vr, Vin, V, L1, L2, R1, R2),
    node(Ms, V, L2, R2).

mode node(?,?,^,^).
node([],                _,L,R) <- L=[], R=[].
node([kill        |Ms],V,L,R) <- node(Ms,V,L,R).
node([cost(X)     |Ms],V,L,R) <- X = V, node(Ms,V,L,R).
node([extract(X)  |Ms],V,L,R) <- X = V, partial(Ms,L,R).
node([insert(Vin) |Ms],V,L,R) <- continue(Vin,V,Ms,L,R).
node([replace(Vin)|Ms],_,L,R) <-
    L = [cost(Vl)|L1], R = [cost(Vr)|R1],
    compare(Vl, Vr, Vin, V, L1, L2, R1, R2),
    node(Ms, V, L2, R2).

mode continue(?,?,?,?,?).
continue(a(N0,C0,P0,D0),a(N1,C1,P1,D1),Ms,L,R) <-
    Even is (C0+C1) mod 2,
    intoheap(Even, a(N0,C0,P0,D0), a(N1,C1,P1,D1), V, L, R, Ls, Rs),
    node(Ms, V, Ls, Rs).

mode intoheap(?,?,?,?,?,?,^,?).
intoheap(0, Ain, Aold, Anew, L, R, Ls, Rs) <-
    Ls = L,
    R = [insert(Anext)|Rs],
    switch(Ain, Aold, Anew, Anext).
intoheap(1, Ain, Aold, Anew, L, R, Ls, Rs) <-
    Rs = R,
    L = [insert(Anext)|Ls],
    switch(Ain, Aold, Anew, Anext).

% for a parent with a single child...
mode compare(?,?,^,^,?).
compare(a(_,Cr,_,_), a(Lin,Cin,Rin,Din), Aout, R, Rs) <- Cin =< Cr :
    Aout = a(Lin,Cin,Rin,Din), R = Rs.
compare(a(Lr,Cr,Rr,Dr), a(Lin,Cin,Rin,Din), Aout, R, Rs) <- Cin >  Cr :
    Aout = a(Lr,Cr,Rr,Dr), R = [replace(a(Lin,Cin,Rin,Din))|Rs].

% for a parent with two children...
```

```
mode compare(?,?,?,^,^,?,^,?).
% parent cost is less than both child costs: stop heapification
compare(a(_,Cl,_,_), a(_,Cr,_,_), a(Lin,Cin,Rin,Din),
                                  Aout, L, Ls, R, Rs) <-
    Cin =< Cl, Cin =< Cr :
    Aout = a(Lin,Cin,Rin,Din), R = Rs, L = Ls.
% parent cost exchanged with left child: heapify down left branch
compare(a(Ll,Cl,Rl,Dl),a(_,Cr,_,_), a(Lin,Cin,Rin,Din),
                                    Aout, L, Ls, R, Rs) <-
    Cl =< Cr, Cl < Cin :
    Aout = a(Ll,Cl,Rl,Dl),
    L = [replace(a(Lin,Cin,Rin,Din))|Ls], R = Rs.
% parent cost exchanged with right child: heapify down right branch
compare(a(_,Cl,_,_),a(Lr,Cr,Rr,Dr), a(Lin,Cin,Rin,Din),
                                    Aout, L, Ls, R, Rs) <-
    Cl > Cr, Cl < Cin :
    Aout = a(Lr,Cr,Rr,Dr),
    R = [replace(a(Lin,Cin,Rin,Din))|Rs], L = Ls.
% parent cost exchanged with right child: heapify down right branch
compare(a(_,Cl,_,_), a(Lr,Cr,Rr,Dr), a(Lin,Cin,Rin,Din),
                                      Aout, L,Ls, R, Rs) <-
    Cr =< Cl, Cr < Cin :
    Aout = a(Lr,Cr,Rr,Dr),
    R = [replace(a(Lin,Cin,Rin,Din))|Rs], L = Ls.
% parent cost exchanged with left child: heapify down left branch
compare(a(Ll,Cl,Rl,Dl), a(_,Cr,_,_), a(Lin,Cin,Rin,Din),
                                      Aout, L, Ls, R, Rs) <-
    Cr > Cl, Cr < Cin :
    Aout = a(Ll,Cl,Rl,Dl),
    L = [replace(a(Lin,Cin,Rin,Din))|Ls], R = Rs.

mode switch(?,?,^,^).
switch(a(L0,C0,R0,D0),a(L1,C1,R1,D1),Top,Bot) <- C0 =< C1 :
    Top = a(L0,C0,R0,D0),
    Bot = a(L1,C1,R1,D1).
switch(a(L0,C0,R0,D0),a(L1,C1,R1,D1),Top,Bot) <- C0 > C1 :
    Top = a(L1,C1,R1,D1),
    Bot = a(L0,C0,R0,D0).
```

A.7 Partitioned Static-Heap and Cost Array (in C)

```
/*
This is parallel version of Dijkstra's algorithm, using a heap for
the priority queue.  It corresponds to the JAM code in Appendix A.3.

To compile:

      cc -O par.c -lseq -lpps -lm

For single run:

      a.out -k8 -w4 -p8 -g32 -r0

executes 9 copies of 1237-node graph (glued together at end points)
on 4 PEs, with eagerness level 8, partition granularity 32, and unit cost
```

```
arcs.  The final parameter can be -r0 (Tokyo costs), -r1 (unit costs), or
-r2 (random costs).  For multiple runs:

      a.out -k8 -r1 -t

executes a bunch of parameterized runs with 9x1237 node unit-cost grid.

The heap code (found on the network) is naive and does not use a
"partial heap" to save heapifications.  In addition, no macros are
used, so function calls are frequent.
*/

#include <stdio.h>
#include <string.h>
#include <math.h>                /* sqrt and rand                        */
#include <usclkc.h>              /* microsecond clock                    */
#include <parallel/microtask.h>  /* fork                                 */
#include <parallel/parallel.h>   /* locks                                */

#define MAXPE      19
#define GRAPHSIZE  1237          /* Tokyo Shimooma graph size            */
#define MAXNODES   30000         /* max heap size per worker             */
#define MAXBUF     10000         /* max # messages outstanding per PE */
#define MAXEDGE    5
#define BIG        99999999
#define Mem_Alloc  shmalloc
#define TRUE       1
#define root       1
#define ascending  1
#define descending -1
#define down       {s_lock(&busy_lock); busy--; s_unlock(&busy_lock);}
#define up         {s_lock(&busy_lock); busy++; s_unlock(&busy_lock);}

typedef int Key;

typedef struct heaptype {
    int node;
    int prev;
} UserVal;

typedef struct messtype {
    int  node;
    int  prev;
    int  cost;
} MessVal;

typedef struct buffertype {
    slock_t lock;
    int     top;
    int     bot;
    MessVal mess[MAXBUF];
} BuffVal;

typedef struct node {
    int neighbor;
    int cost;
```

```
} MyNodeType;

typedef MyNodeType myvector[MAXEDGE];

shared int          PEs;                        /* number of workers     */
shared int          PERIOD;                     /* search eagerness      */
shared int          ISLAND;                     /* partition granularity */
shared int          MODULUS;                    /* random cost range     */
shared int          MAXDIM;                     /* total grid dimension  */
shared int          GDIM;                       /* granule dimension     */
shared int          CDIM;                       /* cluster dimension     */
shared int          t[9][16];                   /* mapping function      */
shared int          kill_yourself,myabort;      /* termination flags     */
shared int          busy;                       /* termination semaphore */
shared slock_t      busy_lock;                  /* semaphore lock        */
shared int          iteration[MAXPE];           /* iterations per PE     */
shared int          tally[MAXPE];               /* buffer use tally      */
shared int          maxtally[MAXPE];            /* buffer high-water mark*/
shared int          htally[MAXPE];              /* heap use tally        */
shared int          maxhtally[MAXPE];           /* heap high-water mark  */
shared int          inserts[MAXPE];             /* heap insertions       */
shared int          xpivots[MAXPE];             /* new bestpath pivots   */
shared BuffVal      *buffer;                    /* shmalloc'ed buffers   */
shared myvector     *graph;                     /* shmalloc'ed graph     */
shared MyNodeType   *bestpaths;                 /* shmalloc'ed answers   */

/*=====================================================================*/

/* HEAP CODE FOLLOWS... */
/* The following function depends on the type of Key  */

int Compare_Keys(Key1, Key2)
Key Key1, Key2;
{
   if (Key1 == Key2)
       return 0;
   else if (Key1 < Key2)
       return -1;
   else
       return 1;
 }

/* The following function depends on the type of Key     */
/* If Key is not a scalar, simple assignment won't work  */
/* ANSI C has structure assignment                       */

void assign_key_value(From, To)
Key From, *To;
{
   *To = From;
}

/* The following function depends on the type of UserValue     */
void assign_user_value(From, To)
UserVal From, *To;
{
```

```
   To->node = From.node;
   To->prev = From.prev;
}

/* Heap Header is a special structure used for controlling the Heap: */
struct Heap_Node {
    Key     Key_Value;
    UserVal User_Value;
};

struct Heap_Head {
   unsigned int Next_Insert;        /* Index of next slot for insert */
   unsigned int Max_Index;          /* Maximum number of nodes       */
            int Heap_Type;          /* Descending or  Ascending      */
   struct Heap_Node Heap[MAXNODES]; /* The collection of nodes       */
};

/* These macros define how to get around in the Heap: */
#define Parent(k)      (k>>1)
#define Left_Child(k)  (k<<1)
#define Right_Child(k) ((k<<1)+1)
#define Is_Left(k)     ((k&0x01)==0)
#define Is_Right(k)    ((k&0x01)!=0)
#define Is_Root(k)     (k==1)

/* Routine Initialize_Heap creates a new Heap and initializes it for
insertion. Note that node 0 is not used to make the parent-child
relationship simpler.  */

struct Heap_Head *Initialize_Heap (Max_Nodes, Type)
unsigned int Max_Nodes;
int Type;
{
        auto   struct Heap_Head *Heap_Ptr;
        extern char *Mem_Alloc();

        Heap_Ptr = (struct Heap_Head *)
                   Mem_Alloc (sizeof (struct Heap_Head));

        if (Heap_Ptr == NULL) {
           perror("malloc");
           exit(0);
           }

        Heap_Ptr->Next_Insert = 1;
        Heap_Ptr->Max_Index = Max_Nodes + 1;
        Heap_Ptr->Heap_Type = (Type >= 0) ? ascending : descending;
        return (Heap_Ptr);
}

/* Routine Insert_Heap_Entry places a new entry into the heap. It
returns the number of nodes in the heap or zero if the new entry will
not fit. Node zero is used to store the data for convenience while the
process is underway.  */

int Insert_Heap_Entry (Heap_Pointer, Key_Val, User_Val)
```

```
struct Heap_Head *Heap_Pointer;
Key      Key_Val;
UserVal  User_Val;
{
        auto   struct Heap_Head *Heap_Ptr;
        auto   unsigned int Index_1, Index_2;
        extern int Compare_Heap_Nodes();

        Heap_Ptr = Heap_Pointer;
        if (Heap_Ptr->Next_Insert >= Heap_Ptr->Max_Index)
           return (0);

        assign_key_value(Key_Val,   &(Heap_Ptr->Heap[0].Key_Value));
        assign_user_value(User_Val, &(Heap_Ptr->Heap[0].User_Value));

        Index_1 = Heap_Ptr->Next_Insert++;
        while (!Is_Root(Index_1)) {
           Index_2 = Parent(Index_1);
           if (Compare_Heap_Nodes (Heap_Ptr, 0, Index_2) < 0) {
               Heap_Ptr->Heap[Index_1] = Heap_Ptr->Heap[Index_2];
               Index_1 = Index_2;
               } else break;
        }
        Heap_Ptr->Heap[Index_1] = Heap_Ptr->Heap[0];
        return (Heap_Ptr->Next_Insert-1);
}

/* Routine Extract_Heap_Entry extracts the top entry from the Heap and
Re-Heapifies the Heap. The function returns a zero value when there
are no more entries in the heap.  */

int Extract_Heap_Entry (Heap_Pointer, Key_Value_Ptr, User_Value_Ptr)
struct Heap_Head *Heap_Pointer;
Key      *Key_Value_Ptr;
UserVal  *User_Value_Ptr;
{
        auto   unsigned int Index;
        auto   struct Heap_Head *Heap_Ptr;

        Heap_Ptr = Heap_Pointer;
        if (Is_Root(Heap_Ptr->Next_Insert))
                return (0);

        assign_key_value( Heap_Ptr->Heap[1].Key_Value,  Key_Value_Ptr);
        assign_user_value(Heap_Ptr->Heap[1].User_Value, User_Value_Ptr );

        Index = --Heap_Ptr->Next_Insert;
        Heap_Ptr->Heap[1] = Heap_Ptr->Heap[Index];
        Adjust_Heap (Heap_Ptr, 1);
        return (1);
}

/* Routine Adjust_Heap re-heapifies a heap with only its root node
out-of-order: */

Adjust_Heap (Heap_Pointer, Root_Index)
```

```
   struct Heap_Head *Heap_Pointer;
   unsigned int Root_Index;
{  auto   struct Heap_Head *Heap_Ptr;
   auto   unsigned int Index_1, Index_2;
   extern int Compare_Heap_Nodes();

   Heap_Ptr = Heap_Pointer;
   Index_1 = Root_Index;
   Heap_Ptr->Heap[0] = Heap_Ptr->Heap[Index_1];
   while ((Index_2 = Left_Child(Index_1)) < Heap_Ptr->Next_Insert) {
      if (Index_2+1 < Heap_Ptr->Next_Insert &&
          Compare_Heap_Nodes (Heap_Ptr, Index_2, Index_2+1) > 0)
          Index_2++;      /* Right Child is smaller */
      if (Compare_Heap_Nodes (Heap_Ptr, 0, Index_2) > 0) {
          Heap_Ptr->Heap[Index_1] = Heap_Ptr->Heap[Index_2];
          Index_1 = Index_2;
          } else break;
   }
   Heap_Ptr->Heap[Index_1] = Heap_Ptr->Heap[0];
}

/* Routine Compare_Heap_Nodes compares two heap nodes and returns -1
if the first node belongs on top of the second node; 0 if the nodes
are equal and +1 if the first node belongs below the second node: */

int Compare_Heap_Nodes (Heap_Pointer, Index_1, Index_2)
   struct Heap_Head *Heap_Pointer;
   unsigned int Index_1, Index_2;
{  auto   struct Heap_Head *Heap_Ptr;
   auto   struct Heap_Node *Heap_Node_Ptr_1, *Heap_Node_Ptr_2;
   auto   int H_Type, Cond;
   extern int Compare_Keys();

        Heap_Ptr = Heap_Pointer;
        H_Type = Heap_Ptr->Heap_Type;
        Heap_Node_Ptr_1 = &Heap_Ptr->Heap[Index_1];
        Heap_Node_Ptr_2 = &Heap_Ptr->Heap[Index_2];
        if (Compare_Keys(Heap_Node_Ptr_1->Key_Value,
                         Heap_Node_Ptr_2->Key_Value) == 0)
          Cond = 0;
        else if ((H_Type > 0 &&
                  Compare_Keys(Heap_Node_Ptr_1->Key_Value,
                               Heap_Node_Ptr_2->Key_Value) < 0) ||
                 (H_Type < 0 &&
                  Compare_Keys(Heap_Node_Ptr_1->Key_Value,
                               Heap_Node_Ptr_2->Key_Value) > 0))
          Cond = -1;
        else
          Cond = 1;
        return (Cond);
}
```

```
void worker()
{    long int         iter=0;
     UserVal          pivot,newentry;
     int              type=1,not_empty=0,newpe;
     int              pe,m,pivot_cost,new_cost;
     struct Heap_Head *boundary;
     extern int       map();

pe = m_get_myid();
boundary = Initialize_Heap(MAXNODES,ascending);
if (pe == 0) {                /* first worker gets start node... */
   pivot.node = 1;
   pivot.prev = 0;
   pivot_cost = 0;
   if (Insert_Heap_Entry(boundary, pivot_cost, pivot)==0)
      heap_error();
   not_empty  = 1;
   }
m_sync();        /* prevent algorithm from starting until now... */

while (type && (kill_yourself != TRUE)) {
   iter++;
   /* extract once per period: this throttles eagerness */
   if (not_empty && (iter%PERIOD == 0)) send_extract(pe);
   type = receive_message(pe,&newentry,&new_cost,not_empty);
   if (type > 0) {                              /* insert request */
      inserts[pe]++;
      htally[pe]++;
      if (htally[pe] > maxhtally[pe]) maxhtally[pe] = htally[pe];
      My_Insert_Heap_Entry(boundary,new_cost,newentry,pe);
      not_empty = 1;
      } else {
      if (type < 0) {                           /* extract request */
         not_empty = Extract_Heap_Entry(boundary,&pivot_cost,&pivot);
         if (not_empty) {
            htally[pe]--;
            if (bestpaths[pivot.node].cost > pivot_cost) {
               xpivots[pe]++;
               bestpaths[pivot.node].cost     = pivot_cost;
               bestpaths[pivot.node].neighbor = pivot.prev;
               for (m=0; m<MAXEDGE; m++) {
                  newentry.node = graph[pivot.node][m].neighbor;
                  if (newentry.node == 0) break;
                  if (newentry.node != pivot.prev) { /* avoid echo */
                     newentry.prev = pivot.node;
                     new_cost = pivot_cost + graph[pivot.node][m].cost;
                /* avoid spawning messages for hopeless extensions */
                     if (bestpaths[newentry.node].cost > new_cost) {
                         newpe = map(newentry.node);
                         send_message(newpe,newentry,new_cost);
}}}} }}}}
iteration[pe] = iter;  /* return number of iterations needed */
shfree(boundary);      /* release space for boundary         */
}
```

```
/*
receive_message returns status of buffer read:
    -1 means the message is a heap extract request
     0 means that the worker has terminated
    +1 means the message is a heap insert request
*/
int receive_message(pe,entry,cost,not_empty)
   int     pe,*cost,not_empty;
   UserVal *entry;
{  int     top,t,decr=0;

/* in case message buffer is empty... */
if (buffer[pe].top == (buffer[pe].bot-1+MAXBUF)%MAXBUF) {
   if (not_empty) return(-1);  /* auto extract request */
   down;
   decr = TRUE;
   while (buffer[pe].top == (buffer[pe].bot-1+MAXBUF)%MAXBUF) {
      if (kill_yourself) return(0);
      }
   }

/* at this point, buffer has message(s) */
if (decr) up;

s_lock(&buffer[pe].lock);

   tally[pe]--;

   top = buffer[pe].top = (buffer[pe].top+1)%MAXBUF;
   entry->node = buffer[pe].mess[top].node;
   entry->prev = buffer[pe].mess[top].prev;
   t = *cost   = buffer[pe].mess[top].cost;

s_unlock(&buffer[pe].lock);

if (t == -1)
   return(-1);
   else
   return(1);
}

/*====================================================================*/
send_extract(pe)
   int pe;
{  UserVal blankentry;
   int bot;

/* if buffer is empty, or tail of buffer is not extract   */
/* then send extract request -- no locking is needed for  */
/* tests, since if another worker writes buffer, it is ok */

bot = (buffer[pe].bot-1+MAXBUF)%MAXBUF;
if ((buffer[pe].top == (buffer[pe].bot-1+MAXBUF)%MAXBUF) ||
    (buffer[pe].mess[bot].cost != -1)) {
   send_message(pe,blankentry,-1);
} }
```

```
int clean_heap(boundary,pe)
     int              pe;
     struct Heap_Head *boundary;
{    struct Heap_Head *newboundary;
     int              not_empty,pivot_cost,hit=0,count=0;
     UserVal          pivot;

s_lock(&buffer[pe].lock);
newboundary = Initialize_Heap(MAXNODES,ascending);

while (Extract_Heap_Entry(boundary,&pivot_cost,&pivot)) {
   count++;
   if (bestpaths[pivot.node].cost > pivot_cost) {
      hit++;
      Insert_Heap_Entry(newboundary,pivot_cost,pivot);
      }
   }
shfree(boundary);
boundary = newboundary;  /* replace old with new */
s_unlock(&buffer[pe].lock);
return(count > hit);
}

My_Insert_Heap_Entry(boundary,cost,entry,pe)
     struct Heap_Head *boundary;
     int              cost,pe;
     UserVal          entry;
{
if (Insert_Heap_Entry(boundary,cost,entry) == 0) {
   if (!clean_heap(boundary,pe)) {
      kill_yourself = TRUE;
      myabort = TRUE;
      } else
      Insert_Heap_Entry(boundary,cost,entry);
}}

/*=====================================================================*/

/* simple model: if receiving buffer full, exit program */
send_message(pe,entry,cost)
   int     pe,cost;
   UserVal entry;
{  int     bot;

s_lock(&buffer[pe].lock);
   if (buffer[pe].bot == (buffer[pe].top-1+MAXBUF)%MAXBUF) {
      kill_yourself = TRUE;
      myabort = TRUE;
      s_unlock(&buffer[pe].lock);
      return;
      }

   tally[pe]++;
   if (tally[pe] > maxtally[pe]) maxtally[pe] = tally[pe];
```

```
   bot                         = buffer[pe].bot;
   buffer[pe].mess[bot].node = entry.node;
   buffer[pe].mess[bot].prev = entry.prev;
   buffer[pe].mess[bot].cost = cost;
   buffer[pe].bot            = (bot+1)%MAXBUF;
s_unlock(&buffer[pe].lock);
}

slave()
{  int pe;
pe = m_get_myid();
if (pe < PEs) {
   worker();
   } else {
   m_sync();
   while (TRUE) {
      while (busy && (!myabort));
      spin(10000);
      kill_yourself = ((busy==0) || myabort);
      if (kill_yourself) break;
}  }  }

/*=============================================================*/
main(argc,argv)
   int     argc;
   char    **argv;
{  usclk_t TIME1,TIME2;
   int     j=1,spawn,dup=0,total_nodes,AUTO=0;
   int     w[8],g[8],p[8],wi,gi,pi,mi;
   float   time[8][8][8],mintime;

   extern int read_graph();
   extern float mytotal_sec();

srand(1237); /* initialized randoms */

PEs    = 1;  /* default is one worker */
PERIOD = 1;  /* default is maximal eagerness */
ISLAND = 1;  /* default is minimal partitioning granularity */
MODULUS= 0;  /* default is unit cost */

while ((j < argc) && (argv[j][0]=='-')) {
   switch(argv[j][1]) {
      case 't': AUTO   = TRUE;            break;
      case 'r': MODULUS= atoi(argv[j]+2); break;
      case 'w': PEs    = atoi(argv[j]+2); break;
      case 'k': dup    = atoi(argv[j]+2); break;
      case 'p': PERIOD = atoi(argv[j]+2); break;
      case 'g': ISLAND = atoi(argv[j]+2); break;
      }
  j++;
  }

if (MAXPE < PEs) {
   printf("fatal error: PE number must be less than %d\n",MAXPE+1);
   exit(0);
```

```
   }

if (AUTO) PEs=16;  /* max buffer alloc. */

if (MODULUS) {
   /* following restrictions are to facilitate sub-block allocation */
   GDIM = sqrt(ISLAND*1.0);
   CDIM = GDIM*4;

   t[2][ 0] = 0;   t[2][ 1] = 1;   t[2][ 2] = 0;   t[2][ 3] = 1;
   t[2][ 4] = 1;   t[2][ 5] = 0;   t[2][ 6] = 1;   t[2][ 7] = 0;
   t[2][ 8] = 0;   t[2][ 9] = 1;   t[2][10] = 0;   t[2][11] = 1;
   t[2][12] = 1;   t[2][13] = 0;   t[2][14] = 1;   t[2][15] = 0;

   t[4][ 0] = 0;   t[4][ 1] = 1;   t[4][ 2] = 0;   t[4][ 3] = 1;
   t[4][ 4] = 2;   t[4][ 5] = 3;   t[4][ 6] = 2;   t[4][ 7] = 3;
   t[4][ 8] = 0;   t[4][ 9] = 1;   t[4][10] = 0;   t[4][11] = 1;
   t[4][12] = 2;   t[4][13] = 3;   t[4][14] = 2;   t[4][15] = 3;

   t[8][ 0] = 0;   t[8][ 1] = 1;   t[8][ 2] = 2;   t[8][ 3] = 3;
   t[8][ 4] = 4;   t[8][ 5] = 5;   t[8][ 6] = 6;   t[8][ 7] = 7;
   t[8][ 8] = 0;   t[8][ 9] = 1;   t[8][10] = 2;   t[8][11] = 3;
   t[8][12] = 4;   t[8][13] = 5;   t[8][14] = 6;   t[8][15] = 7;
   }

alloc_memory(dup);
total_nodes = (MODULUS) ? gen_graph(dup) : read_graph(dup);
usclk_init();

if (AUTO) {
   if (MODULUS) {
      if (CDIM > MAXDIM) {
         printf("partition granule is greater than graph size!\n");
         exit(0);
         }
      if ((PEs != 1) && (PEs != 2) && (PEs != 4) &&
                        (PEs != 8) && (PEs != 16)) {
         printf("number of PEs must be power of two for grid\n");
         exit(0);
         }
      p[0] =  4;  w[0] =  1;  g[0] =  4;
      p[1] =  8;  w[1] =  4;  g[1] = 16;
      p[2] = 16;  w[2] =  8;  g[2] = 36;
      p[3] = 32;  w[3] = 16;  g[3] = 64;
      p[4] = 64;  w[4] =  0;  g[4] = 81;
      p[5] =  0;              g[5] =100;
                              g[6] =  0;
      } else {
      p[0] =  1;  w[0] =  1;  g[0] =  1;
      p[1] =  8;  w[1] =  2;  g[1] =  4;
      p[2] = 16;  w[2] =  4;  g[2] =  8;
      p[3] = 24;  w[3] =  8;  g[3] = 16;
      p[4] = 32;  w[4] = 12;  g[4] = 32;
      p[5] =  0;  w[5] = 16;  g[5] = 64;
                  w[6] =  0;  g[6] =  0;
      }
```

```
   mintime = 10000.0;
   for (wi=0; w[wi]>0; wi++)
      for (pi=0; p[pi]>0; pi++) {

         printf("\nPEs  per  gran  node  cost  checksum   sec    iter");
         printf("  bmax  hmax   SU   util mess\n");

         for (gi=0; g[gi]>0; gi++) {
            PEs    = w[wi];
            PERIOD = p[pi];
            ISLAND = g[gi];

            init_search(total_nodes);
            spawn = PEs+1;
            m_set_procs(spawn);
            TIME1 = getusclk();
               m_fork(slave);
            TIME2 = getusclk();
            m_kill_procs();

            time[wi][pi][gi] = mytotal_sec(TIME1,TIME2);
            if ((PEs==1) && (time[wi][pi][gi] < mintime))
               mintime = time[wi][pi][gi];
            if (myabort)
               printf(" %2d   %2d   %3d %5d   %3d   buffer overflow\n",
                  PEs,PERIOD,ISLAND,total_nodes,MODULUS);
               else
               dump_line(total_nodes,time[wi][pi][gi],mintime);
            }
       }
   } else {
   init_search(total_nodes);
   spawn = PEs+1;
   m_set_procs(spawn);
   TIME1 = getusclk();
      m_fork(slave);
   TIME2 = getusclk();
   m_kill_procs();

   mintime = mytotal_sec(TIME1,TIME2);
   printf("\nPEs  per  gran  node  cost  checksum   sec    iter");
   printf("  bmax  hmax   SU   util mess\n");
   dump_line(total_nodes,mintime,mintime);
   }
}

dump_line(total_nodes,time,mintime)
    int   total_nodes;
    float time,mintime;
{   int   hmax,bmax,iter_sum,m,insert_sum;
insert_sum = iter_sum = 0;
hmax = bmax = 0;
for (m=0; m<PEs; m++) {
   iter_sum += iteration[m];
   insert_sum += inserts[m];
```

```
   if (maxhtally[m]>hmax) hmax = maxhtally[m];
   if (maxtally[m] >bmax) bmax = maxtally[m];
   }
printf(" %2d   %2d   %3d %5d   %3d  %8d %5.2f  ",
   PEs,PERIOD,ISLAND,total_nodes,MODULUS,
   checksum(bestpaths,total_nodes),time);
printf("%6d %5d %5d %5.2f %5.2f %4.1f\n",
   iter_sum,bmax,hmax,mintime/time,mintime/(PEs*time),
   (insert_sum*1.0)/total_nodes);
}

/*=================== auxillary functions ======================*/

/* all about sub-block mapping for grids:

This function is given a node and returns the partition (i.e., worker
or PE) associated with that node.  The function is modeled after
Ichiyoshi and Wada.  The square grid is composed of square clusters,
each holding 16 square regions.  Each region is allocated to a single
PE.  The cluster is tiled so that PEs are uniformly distributed over
the cluster.  One requirement to facilitate exact tiling is that the
number of PEs be a power of two.  The user specifies only the number
of nodes per region (this is called the partition granularity).  For
example, with a region size of 2x2 nodes, and 8 PEs, a single cluster
looks like:

    0 0 1 1 2 2 3 3
    0 0 1 1 2 2 3 3
    4 4 5 5 6 6 7 7
    4 4 5 5 6 6 7 7
    0 0 1 1 2 2 3 3
    0 0 1 1 2 2 3 3
    4 4 5 5 6 6 7 7
    4 4 5 5 6 6 7 7

where each position is a node within the grid, labeled by its
associated paritition.  For a 20x20 graph:

    0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3 0 0 1 1
    0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3 0 0 1 1
    4 4 5 5 6 6 7 7 4 4 5 5 6 6 7 7 4 4 5 5
    4 4 5 5 6 6 7 7 4 4 5 5 6 6 7 7 4 4 5 5
    0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3 0 0 1 1
    0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3 0 0 1 1
    4 4 5 5 6 6 7 7 4 4 5 5 6 6 7 7 4 4 5 5
    4 4 5 5 6 6 7 7 4 4 5 5 6 6 7 7 4 4 5 5
    0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3 0 0 1 1
    0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3 0 0 1 1
    4 4 5 5 6 6 7 7 4 4 5 5 6 6 7 7 4 4 5 5
    4 4 5 5 6 6 7 7 4 4 5 5 6 6 7 7 4 4 5 5
    0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3 0 0 1 1
    0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3 0 0 1 1
    4 4 5 5 6 6 7 7 4 4 5 5 6 6 7 7 4 4 5 5
    4 4 5 5 6 6 7 7 4 4 5 5 6 6 7 7 4 4 5 5
    0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3 0 0 1 1
    0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3 0 0 1 1
```

```
    4 4 5 5 6 6 7 7 4 4 5 5 6 6 7 7 4 4 5 5
    4 4 5 5 6 6 7 7 4 4 5 5 6 6 7 7 4 4 5 5

Note that in this example, full number of clusters don't cover the
entire graph, so that boundaries are covered by partial clusters.
This means that there is some inequality of nodes per PE (in this
example, PEs 0,1,4,5 get extra nodes, whereas PEs 2,3,6,7 get fewer
nodes).  Hopefully this effect will be small.
*/

int map(node)
   int node;
{  int n0,pe,a,b,c,d,n1,n2,q,r,x,y;

   if (PEs == 1)
      return(0);

   if (MODULUS) { /* grids... */
      n0 = node - 1;  /* must start at zero */
      q = n0 / MAXDIM;
      r = n0 % MAXDIM;
      b = r / CDIM;
      a = q / CDIM;
      n1 = q*MAXDIM + b*CDIM;
      n2 = a*CDIM*MAXDIM + r;
      c = (n0-n2)/MAXDIM;
      d = n0-n1;
      x = c/GDIM;
      y = d/GDIM;
      pe = x*4 + y;   /* cluster is always 4x4 regions */

      if (PEs == 16)
         return(pe);

      pe = t[PEs][pe];
      return(pe);

      } else {    /* Tokyo map... */
      pe = (node / ISLAND) % PEs;
      return(pe);
      }
}

/*=================================================================*/
alloc_memory(dup)
    int dup;
{   int MAXGRAPH;

MAXGRAPH = (dup+1)*(GRAPHSIZE+10);  /* extra fudge */

buffer    = (BuffVal    *)shmalloc(     PEs*sizeof(BuffVal)   );
graph     = (myvector   *)shmalloc(MAXGRAPH*sizeof(myvector)  );
bestpaths = (MyNodeType *)shmalloc(MAXGRAPH*sizeof(MyNodeType));

if ((buffer == NULL) || (graph == NULL) || (bestpaths == NULL)) {
   printf("fatal error: memory overflow in shmalloc.\n");
```

```
   exit(0);
   }
}

/*====================================================================*/
int gen_graph(dup)
   int dup;
{  int n,m,lim,i,j,k,c,g,MAXGRAPH,ok;

MAXGRAPH = (dup+1)*GRAPHSIZE;
MAXDIM   = (int)(sqrt(MAXGRAPH*1.0));
lim      = MAXDIM*MAXDIM;

for (k=0; k<MAXDIM; k++) {
   for (j=1; j<=MAXDIM; j++) {
      g = j + k*MAXDIM;
      c = 0;
      if ((j != MAXDIM) && (g+1 > 0) && (g+1 <= lim))
         graph[g][c++].neighbor = g+1;
      if ((j != 1) && (g-1 > 0) && (g-1 <= lim))
         graph[g][c++].neighbor = g-1;
      if ((g+MAXDIM > 0) && (g+MAXDIM <= lim))
         graph[g][c++].neighbor = g+MAXDIM;
      if ((g-MAXDIM > 0) && (g-MAXDIM <= lim))
         graph[g][c++].neighbor = g-MAXDIM;

      for (i=0; i<c; i++) {
         if (MODULUS > 1) {
            n = graph[g][i].neighbor;
            for (m=0; m<4; m++)
               if (graph[n][m].neighbor == g)
                  graph[g][i].cost = graph[n][m].cost;
            if (graph[g][i].cost == 0)
               graph[g][i].cost = (rand()%MODULUS)+1;
            } else
            graph[g][i].cost = 1;
         }
    }
 }
return(lim);
}

/*====================================================================*/
int read_graph(dup)
   int  dup;
{  char s;
   FILE *infile;
   int  i,j,k,m;
   int  nodes,index,cost,neighbor,link,base;

infile = fopen("big.data","r");
nodes = 0;
while (TRUE) {
   if (fscanf(infile,"%d ",&index) == EOF) break;
   nodes++;
   for (i=0; i<MAXEDGE; i++) {
```

```
      fscanf(infile,"%d*%d%1c",&neighbor,&cost,&s);
      graph[index][i].neighbor = neighbor;
      graph[index][i].cost = cost;
      if (s == '\n') break;
      }
   }

link = i+1;  /* new edge in last node */
base = nodes;
printf("%d nodes in graph\n",nodes);

/* duplicate the graph dup times */
for (j=0; j<dup; j++) {
   graph[base][link].neighbor = base+1;
   graph[base][link].cost = 10;
   for (m=1; m<=nodes; m++) {
      for (k=0; k<MAXEDGE; k++)
         if (graph[m][k].cost > 0) {
            graph[base+m][k].neighbor = graph[m][k].neighbor + base;
            graph[base+m][k].cost     = graph[m][k].cost;
            }
      }
   base += nodes;
   }
nodes = base;
printf("%d nodes in duplicated graph\n",base);
return(base);
}

/*=====================================================================*/
init_search(total_nodes)
   int  total_nodes;
{  int  j;

for (j=0;j<PEs;j++) {
   s_init_lock(&buffer[j].lock);
   buffer[j].top = 0;
   buffer[j].bot = 1;
   tally[j]      = 0;
   htally[j]     = 0;
   maxtally[j]   = 0;
   maxhtally[j]  = 0;
   inserts[j]    = 0;
   xpivots[j]    = 0;
   }

for (j=0; j<=total_nodes; j++) {
   bestpaths[j].neighbor = 0;
   bestpaths[j].cost     = BIG;
   }

kill_yourself = 0;        /* don't kill yourself (yet) */
myabort = 0;              /* buffer overflow indicator */
busy = PEs;               /* termination semaphore     */
s_init_lock(&busy_lock);  /* lock on termination sem.  */
}
```

```
int checksum(bestpaths,nodes)
     MyNodeType bestpaths[];
     int nodes;
{    int i;
     long int cost;
cost = 0;
for (i=1; i<=nodes; i++) {
    cost = (cost+bestpaths[i].cost)%BIG;
    }
return(cost);
}

float mytotal_sec(TIME1,TIME2)
     usclk_t TIME1,TIME2;
{    int total_time;
total_time = ((int) TIME2) - ((int) TIME1);
return (total_time/1000000.0);
}

spin(k)
   int k;
{  int i;
   float j=0.0;
for (i=0; i<k; i++) j++;
}

heap_error()
{
printf("fatal error: heap is full.\n");
exit(0);
}
```