Dijkstra Shortest Path Parallelization

Dijkstra's algorithm is a popular graph algorithm for calculating the shortest path between nodes. The single-source variation calculates the shortest path between a single source node and all other nodes in the graph reachable from that source node. This algorithm is used in link-state routing protocols, such as the Open Shortest Path First (OSPF) protocol used for IP routing.

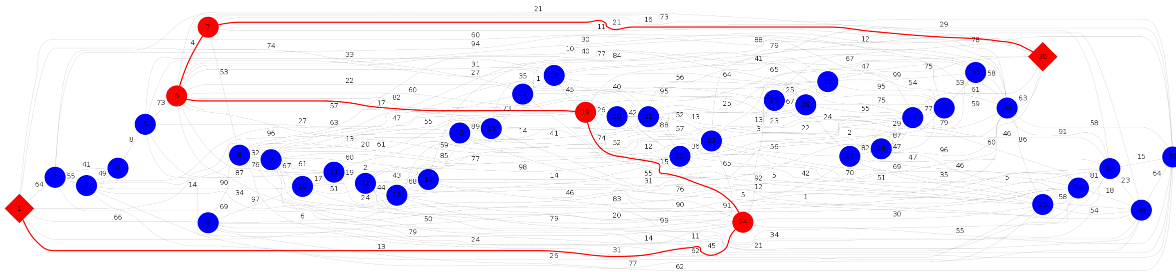


Figure 1: Example Dijkstra Shortest Path

The Dijkstra algorithm operates on a graph of nodes connected by edges with non-negative costs. The graph may have cycles. Starting from the source node, the algorithm greedily visits the closest unvisited node and updates its neighbors with a new minimum path cost. This is repeated until all reachable nodes have been visited. In pseudocode:

```
initialize all node path information

set shortest_path_node = source_node

until all reachable nodes visited {
  set shortest_path_node = visited
  for each unvisited neighbor of shortest_path_node {
    if path from shortest_path_node < node path_distance {
      update node path_distance
      set node path_predecessor = shortest_path_node
    }
  }
  for each unvisited node {
    select new shortest_path_node
  }
}
```

Listing 1: Example Dijkstra Shortest Path

The shortest path from source to any target node can be found by working backwards from the target node using the predecessor node information.

© 2011 Issue 001

Prism is used to migrate a sequential Dijkstra implementation into a properly threaded and synchronized implementation running on multiple cores. The parallelization flow follows CriticalBlue's recommended methodology:

Analysis
: Tracing and analysis of the initial code

Exploration
: Determination of a parallelization strategy before modifying code

Implementation
: Implementation of the parallelization strategy

Verification
: Verification of parallel synchronization and performance

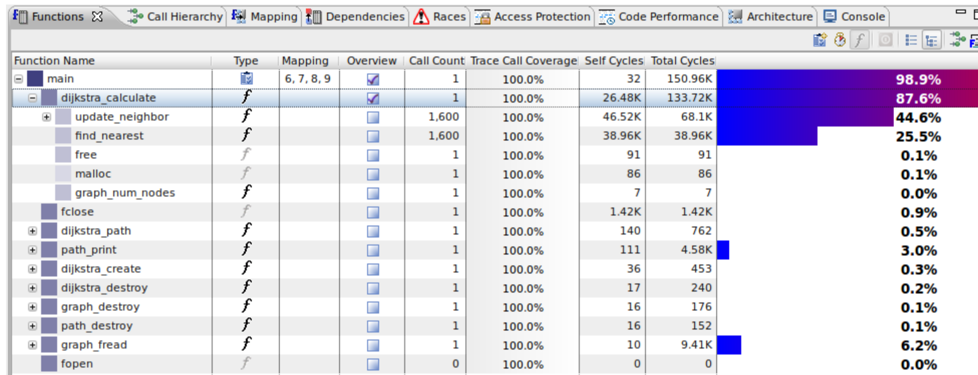
Tuning
: Additional refinements for performance and resource efficiency

One parallel variation of Dijkstra involves computing the shortest paths from *all* source nodes. Since the shortest path computation from one source is independent from any other, this is trivial to parallelize. However, protocols such as OSPF are single source, so this variation is not appropriate for this type of routing. Parallelization *within* the single-source algorithm is the focus of the following sections.

Initial Analysis

The starting point for parallelization is a working sequential implementation of the Dijkstra algorithm. Initial analysis focuses on understanding the sequential flow and performance of the application. Prism is used to trace the application run on a typical sample set to get an idea of the execution profile and the function call tree of the application. Figure 2 shows profile results using **`prismtrace`** to trace Dijkstra for the 40 node input graph shown in Figure 1:

© 2011 Issue 001



Function Name	Type	Mapping	Overview	Call Count	Trace Call Coverage	Self Cycles	Total Cycles	
main		6, 7, 8, 9		1	100.0%	32	150.96K	98.9%
dijkstra_calculate	f			1	100.0%	26.48K	133.72K	87.6%
update_neighbor	f			1,600	100.0%	46.52K	68.1K	44.6%
find_nearest	f			1,600	100.0%	38.96K	38.96K	25.5%
free	f			1	100.0%	91	91	0.1%
malloc	f			1	100.0%	86	86	0.1%
graph_num_nodes	f			1	100.0%	7	7	0.0%
fclose	f			1	100.0%	1.42K	1.42K	0.9%
dijkstra_path	f			1	100.0%	140	762	0.5%
path_print	f			1	100.0%	111	4.58K	3.0%
dijkstra_create	f			1	100.0%	36	453	0.3%
dijkstra_destroy	f			1	100.0%	17	240	0.2%
graph_destroy	f			1	100.0%	16	176	0.1%
path_destroy	f			1	100.0%	16	152	0.1%
graph_fread	f			1	100.0%	10	9.41K	6.2%
fopen	f			0	100.0%	0	0	0.0%

Figure 2: Execution Profile

Examining the profile shows that the **`dijkstra_calculate`** function has two functions **`update_neighbor`** and **`find_nearest`** which are called 1600 times and take a majority of the execution time. The relevant code within **`dijkstra_calculate`** is shown in Listing 2:

```
 93
 94  int dijkstra_calculate(dijkstra_t *dijkstra) {
 95    // ...
 96
 97    // initialize source
 98    nearest_node = src;
 99    nearest_dist = 0;
100
101    while (nearest_node != NULL_NODE) {
102      // mark nearest node visited
103      visited[nearest_node] = 1;
104
105      // update neighbor minimum distances
106      for (u = 0; u < num_nodes; ++u) {
107        update_neighbor(u, &calc);
108      }
109
110      // find nearest unvisited node
111      nearest_node = NULL_NODE;
112      nearest_dist = INF_DIST;
113      for (u = 0; u < num_nodes; ++u) {
114        find_nearest(u, &calc);
115      }
116    }
```

Listing 2: Dominant **`dijkstra_calculate`** loop

Structurally, the code consists of one outer loop with two inner loops. The number of nodes is 40. Since the two inner loop functions are called 1600 times, this suggests that the

© 2011 Issue 001

outer loop executes for 40 times, and the inner loop functions have an N^2^ dependence on the number of nodes in the graph. At each outer loop iteration, a new nearest node is found and used in the next iteration.

At each outer loop iteration, this implementation greedily finds the nearest unvisited node which represents the shortest partial path from the source node so far. The N^2^ dependence reflects that simple lists are used to hold the path distances found to each node. This is a reasonable choice for dense graphs where the number of edges between nodes is greater than the number of nodes.

This initial analysis highlights two important considerations. First, each outer loop iteration depends on the `nearest_node` computed in the previous iteration, so there is a fundamental loop dependency which means there will be no benefit in trying to parallelize the outer loop directly.

Secondly, for any reasonable graph size, the number of inner loop executions will be much larger than the number of cores, and it will grow as the number of graph nodes increases. Running each inner loop function in parallel is not practical because the synchronization costs and memory usage will overwhelm the application.

One solution is to divide the graph into regions, where the number of regions is of the same order as the number of cores. As the number of nodes in a graph increases, the number of regions remains constant. This simple refactoring makes the parallel explorations, performed in the next section, independent of the graph size. The `update_neighbor` and `find_nearest` functions are therefore rewritten to iterate over nodes contained in regions as shown for `find_nearest` in Listing 3:

© 2011 Issue 001

```
 87
 88 static void find_nearest_in_region(region_info_t *region) {
 89   // ...
 90
 91   uint_t u_begin = region->begin_node;
 92   uint_t u_end = region->end_node;
 93   for (u = u_begin; u < u_end; ++u) {
 94     if (!visited[u]) {
 95       dist = path_dist[u];
 96       if (dist != INF_DIST && dist < *nearest_dist) {
 97         *nearest_node = u;
 98         *nearest_dist = dist;
 99       }
100     }
101   }
102 }
```

Listing 3: Refactored **`find_nearest_in_region`** inner loop

The refactored outer loop is shown in listing 4:

```
109
110 int dijkstra_calculate(dijkstra_t *dijkstra) {
111   // ...
112
113   nearest_node = src;
114   nearest_dist = 0;
115
116   while (nearest_node != NULL_NODE) {
117     // mark nearest node visited
118     visited[nearest_node] = 1;
119
120     // update neighbor minimum distances
121     for (r = 0; r < num_regions; ++r) {
122       update_neighbor_in_region(&region[r]);
123     }
124
125     // find nearest unvisited node
126     nearest_node = NULL_NODE;
127     nearest_dist = INF_DIST;
128     for (r = 0; r < num_regions; ++r) {
129       find_nearest_in_region(&region[r]);
130     }
131   }
132
133   // ...
```

Listing 4: Refactored **`dijkstra_calculate`** loop

© 2011 Issue 001

What-If Exploration

The exploration phase is used to determine a good strategy for introducing parallelism into the Dijkstra application.

The initial target multicore architecture is set to four cores, so the graph is split into four regions to match that number of cores. The two refactored functions, **`update_neighbor_in_region`** and **`find_nearest_in_region`** take up almost all the execution time, so it makes sense to explore running them as independent tasks.

Within Prism, forcing the **`update_neighbor_in_region`** and applying the change generates the *what-if* schedule shown in figure 3:

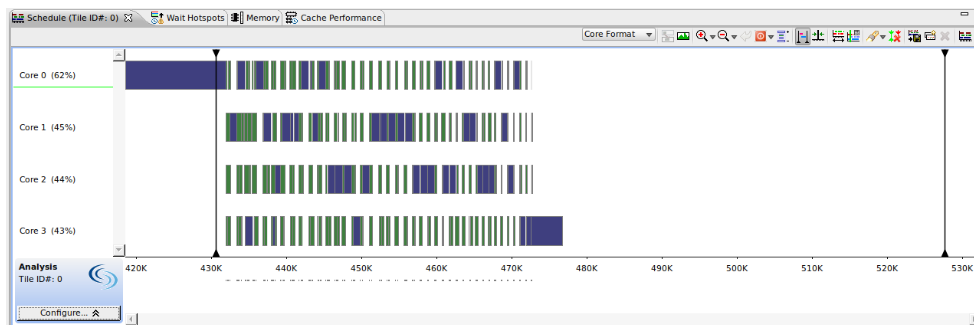


Figure 3: **`update_neighbor_in_region`** Schedule

In the figure, the black measurement markers span the original **`dijkstra_calculate`** execution. On an Intel Core i5 processor with four cores, the execution time is approximately 97k cycles. In the parallel schedule running on four cores, the same function executes in approximately 42k cycles or roughly a 2.2x speed up.

Forcing the second function, **`find_nearest_in_region`**, and applying, yields the new schedule in figure 4:

© 2011 Issue 001

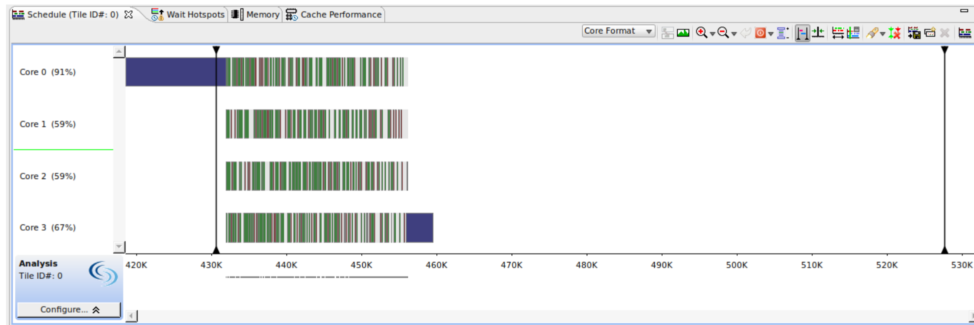


Figure 4: Both `update_neighbor_in_region` and `find_nearest_in_region` Schedule

With both functions forced, execution on the quad Core i5 is approximately 25k cycles, a 3.8x speed up. This is coming close to a linear speed up, but it should be understood that this schedule is initially computed with no estimated cost of synchronization. For the sample graph, each task occurs 160 times, so even a synchronization cost of 50 cycles modeled in Prism would cause a reduction in speed up to closer to 3×. This sensitivity will be considered further during the tuning phase.

Figure 5 shows the same schedule with tasks on the vertical axis:

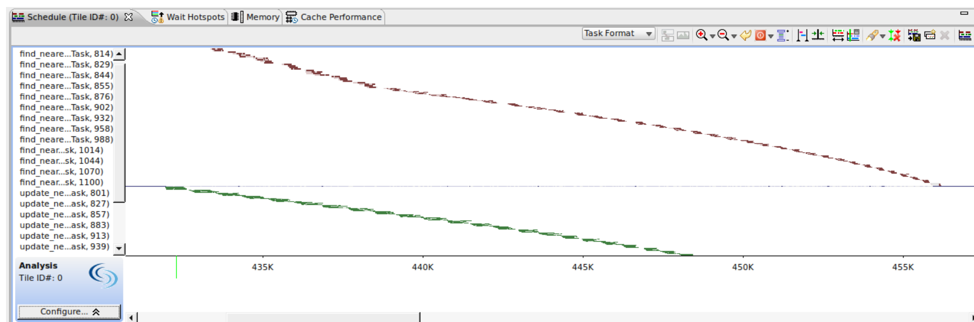


Figure 5: By-Tasks view of Previous Schedule

This schedule shows the 160 executions of both region functions. Zooming in on a group of `find_nearest_in_region` functions shows dependencies between them as shown in Figure 6:

© 2011 Issue 001

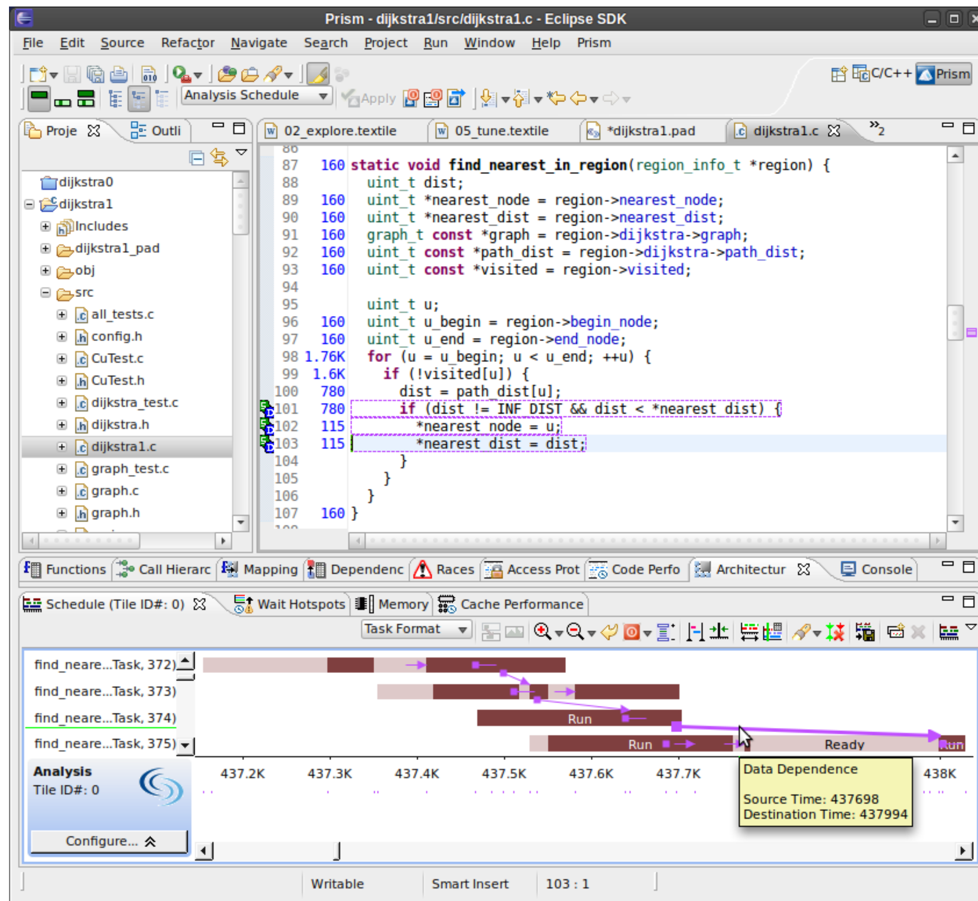


Figure 6: Inter-Task Dependencies

Each **find_nearest_in_region** group represents the calls for one iteration of the outer loop. Selecting one of the dependency arrows and annotating the dependency is also shown in Figure 6. The source-destination pairs show that the dependency concerns reading and writing the **nearest_node** values across all the regions. This will need to be accounted for in the implementation.

Parallel Implementation

The parallel implementation uses the **pthread** library to implement threads and synchronization.

The implementation strategy involves threading both inner loop functions, **update_neighbor_in_region** and **find_nearest_in_region**. Within **find_nearest_in_region**, the inter-region **nearest_node** dependency is handled using a

© 2011 Issue 001

mutex to lock the critical regions.

The threaded inner loops are shown in Listing 5:

```
122
123 int dijkstra_calculate(dijkstra_t *dijkstra) {
124   // ..
125
126   nearest_node = src;
127   nearest_dist = 0;
128
129   while (nearest_node != NULL_NODE) {
130     // mark nearest node visited
131     visited[nearest_node] = 1;
132
133     for (r = 0; r < num_regions; ++r) {
134       pthread_create(&update_thread[r], NULL,
135           update_neighbor_in_region, &region[r]);
136     }
137     for (r = 0; r < num_regions; ++r) {
138       pthread_join(update_thread[r], NULL);
139     }
140
141     // find nearest unvisited node
142     nearest_node = NULL_NODE;
143     nearest_dist = INF_DIST;
144     for (r = 0; r < num_regions; ++r) {
145       pthread_create(&find_thread[r], NULL,
146           find_nearest_in_region, &region[r]);
147     }
148     for (r = 0; r < num_regions; ++r) {
149       pthread_join(find_thread[r], NULL);
150     }
151   }
152
153   // ...
154 }
```

Listing 5: Threaded `dijkstra_calculate` loop

All unit tests from the original implementation pass with the new implementation pass cleanly, and rerunning the application several times appear to give consistent and correct results.

Parallel Verification

Verification is used within Prism to check for synchronization errors and performance issues. As with the sequential implementation, the parallel implementation is traced

© 2011 Issue 001

within Prism. Though the results appear correct, Prism identifies many data races as shown in Figure 7:

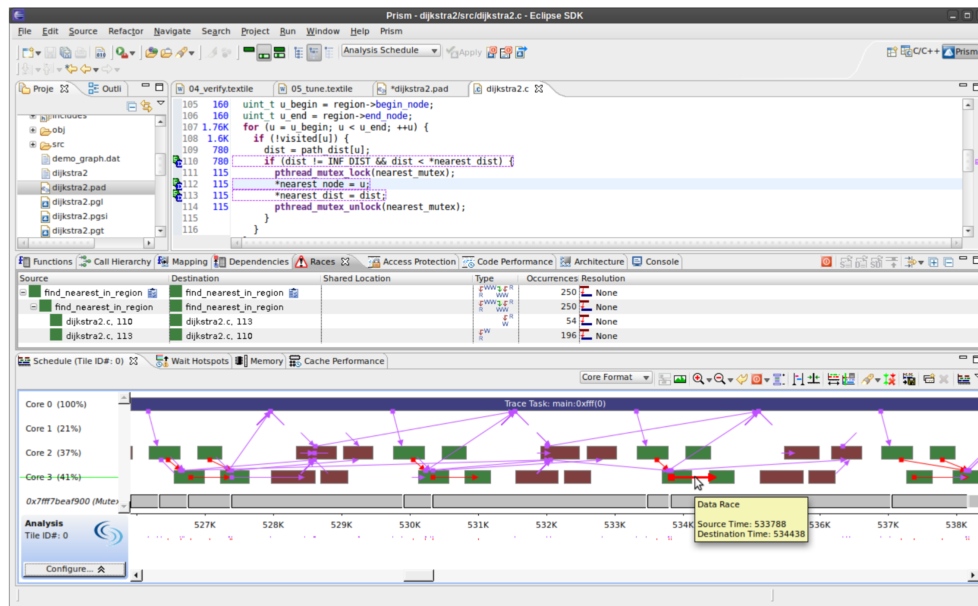


Figure 7: Data Races in Implementation

The source and destination of these races, also shown in Figure 7, points to the critical section within the `find_nearest_in_region`. At first glance, the `nearest_dist` and `nearest_node` writes seem properly protected. However, as Prism indicates, there is also a read of `nearest_dist` in the `if` expression immediately above the mutex lock. It is possible that two regions can read the `nearest_dist` value in the `if` clause and both decide to update that value in an arbitrary order, possibly incorrectly overwriting the lowest `nearest_dist` with a larger value.

The fix is straightforward. The `nearest_mutex` lock and unlock should surround the entire `if` clause as shown in Listing 6:

© 2011 Issue 001

```
122
123 static void *find_nearest_in_region(void *regionv) {
124   // ...
125
126   uint_t u_begin = region->begin_node;
127   uint_t u_end = region->end_node;
128   for (u = u_begin; u < u_end; ++u) {
129     if (!visited[u]) {
130       dist = path_dist[u];
131       pthread_mutex_lock(nearest_mutex);
132       if (dist != INF_DIST && dist < *nearest_dist) {
133         *nearest_node = u;
134         *nearest_dist = dist;
135       }
136       pthread_mutex_unlock(nearest_mutex);
137     }
138   }
139
140   return NULL;
141 }
```

Listing 6: Enlarged Critical Region Lock

Retracing the corrected implementation in Prism shows no races detected.

Tuning

Figure 8 shows the trace results for a quad Core i5:

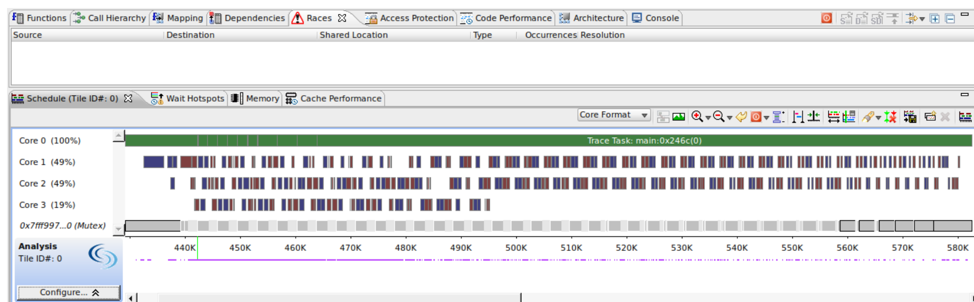


Figure 8: Race-Free Schedule

The traced `dijkstra_calculation` function takes approximately 150k cycles. The trace is slower than the original sequential implementation, and these extra cycles are due to inner loop synchronization requirements. There are two primary causes. The first occurs entirely within the `find_nearest_region` execution. As shown in Listing 6, the `nearest_mutex` is locked and unlocked every iteration.

© 2011 Issue 001

The `find_nearest_in_region` loop performs a minimum search function, and because the minimum function is associative, the mutex can be removed using a familiar parallel reduction technique. Instead of locking each iteration, a region can calculate its own local minimum using no lock and pass that value up to the main thread, which, in turn, finds the minimum of all regions. The local lock-free implementation is shown in Listing 7:

```
132
133 static void *find_nearest_in_region(void *regionv) {
134   // ...
135
136   uint_t nearest_region_node = NULL_NODE;
137   uint_t nearest_region_dist = INF_DIST;
138
139   uint_t u_begin = region->begin_node;
140   uint_t u_end = region->end_node;
141   for (u = u_begin; u < u_end; ++u) {
142     if (!visited[u]) {
143       dist = path_dist[u];
144       if (dist != INF_DIST && dist < nearest_region_dist) {
145         nearest_region_node = u;
146         nearest_region_dist = dist;
147       }
148     }
149   }
```

Listing 7: Local Lock-Free Implementation

For a graph with `N` nodes, `N(N-1)/2` mutex lock/unlocks can be removed.

The second cause for concern is that the lifetime of `update_neighborhood_in_region` and `find_nearest_in_region` parallel tasks is only one iteration of the outer loop. Each task involves a thread create/join pair- an expensive operation. With this parallel implementation, there are `2N` threads created every run.

One way to reduce the number of threads is to create one thread per region, and have the threads synchronize a few times each outer loop iteration to ensure proper operation. This implementation style typically uses barrier synchronization. Each barrier uses a mutex-condition variable pair to ensure synchronization. Listing 8 shows a portion of the barrier implementation:

© 2011 Issue 001

```
160
161 void *dijkstra_calculate_loop(void *regionv) {
162   // ...
163
164   // wait until everyone is here
165   barrier_wait(loop_barrier);
166
167   while (nearest_node[id] != NULL_NODE) {
168
169     // update neighbor minimum distances
170     update_neighbor_region(region);
171
172     // wait until everyone is here
173     barrier_wait(loop_barrier);
174
175     // find nearest unvisited node
176     find_nearest_region(region);
177
178     // wait until everyone is here
179     barrier_wait(loop_barrier);
180
181     // find global nearest node and mark visited
182     if (id == MASTER_REGION) {
183       find_nearest_and_visit(&region[id]);
184     }
185
186     // wait until everyone is here
187     barrier_wait(loop_barrier);
188   }
189 }
```

Listing 8: `dijkstra_calculation` Outer Loop using Barriers

Some pthreads implementations provide a non-portable `pthread_barrier_t` type. Prism does not directly support this, so the Dijkstra example uses a `barrier_t` implemented from `pthread_mutex_t` and `pthread_cond_t` primitives so that barrier synchronization is considered during verification.

Barrier synchronization occurs 3 times per outer loop iteration, so `3N+1` barrier synchronizations are traded for `2RN-3` fewer thread create/joins (where `N` = # of graph nodes and `R` = # of regions).

Tracing the new implementation inside Prism shows the final schedule in Figure 9:

© 2011 Issue 001

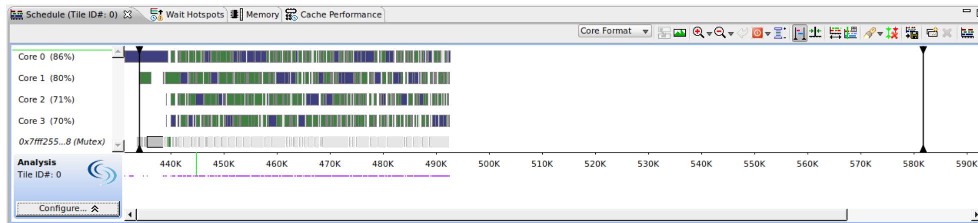


Figure 9: Schedule using Reduction and Barrier Techniques

By removing the inner loop mutex and using the barrier implementation style, the optimized implementation is 2.5x faster than the previous oversynchronized version.

Summary

Prism was used to migrate a sequential Dijkstra implementation into a properly threaded and synchronized implementation running on multiple cores. The approach demonstrates CriticalBlue's multicore programming flow:

Analysis
: Tracing and analysis of initial code

Exploration
: Determination of parallelization strategy before modifying code

Implementation
: Implementation of parallelization strategy

Verification
: Verification of parallel synchronization and performance

Tuning
: Additional refinements for performance and resource efficiency

Within the Dijkstra shortest-path algorithm, dependencies between outer loop iterations required a good understanding of the implementation tradeoffs. Within this case study, initial analysis and exploration identified the key functions to parallelize and suggested a simple refactoring to desensitize the implementation to the problem size. During verification, a typical data race bug was identified and corrected. The resulting implementation was correct but costly. During tuning, reduction and barrier techniques were used to optimize the final parallel performance. Prism provided intuitive visualization and verification to productively arrive at this solution.

© 2011 Issue 001