Analysis of Parallel Dijkstra

Eric Andrews

December 5, 2017

Initialize

Work:

For each vertex v, the algorithm must set perform a single assignment. Thus the work of is given by:

$$v$$

Span:

No iteration requires any other; given v threads, each assignment can be done concurrently. Thus the span is given by:

$$1$$

Get Min Vertex

Work:

The algorithm must check each vertex once. Thus the work of the algorithm is given by:

$$v$$

Span:

No iteration requires any other; given v threads, getMinVertex could perform each check concurrently.

If given v threads, the algorithm must merge them together to find the global min from the local mins. This is done in a tree structure, which gives it a logarithmic runtime. Thus the span is given by:

$$1 + \lceil \log_2(v) \rceil$$

Update Distances

Work:

The algorithm must check each vertex once. Thus the work of the algorithm is given by:

$$v$$

Span:

Each check can be performed concurrently. Thus the span of the algorithm is given by:

$$1$$

Parallel Dijkstra

Work:

initialize is performed once, and has work v.

getMinVertex and updateDistances are each performed $v-1$ times, and have a combined work of 2v. Thus the work of the overall algorithm is given by:

$$\begin{aligned} & v + 2v(v-1) \\ & = v + 2v^2 - 2v \\ & = 2v^2 - v \end{aligned}$$

Span:

initialize is performed once, and has work 1.

getMinVertex and updateDistances are each performed $v-1$ times in serial, and have spans of $1 + \lceil \log_2(v) \rceil$ and 1, respectively. Thus the span of the overall algorithm is given by:

$$\begin{aligned} & 1 + (v-1)(1 + 1 + \lceil \log_2(v) \rceil) \\ & = 1 + (v-1)(2 + \lceil \log_2(v) \rceil) \end{aligned}$$

Questions

1. The ideal runtime of parallel Dijkstra's is approximately $O(v \times \log_2(v))$. Therefore if e is significantly greater than v, it is more efficient to use a parallel algorithm; otherwise thread overhead makes the adjacency list implementation more efficient. Thus in sparse graphs I would use an adjacency list, and in dense graphs I would use a parallel algorithm.

2. With only 1 processor, the runtime of parallel Dijkstra's is $O(v^2 - v)$. As the number of processors increases, this is divided by the number of processors. Taking into account thread overhead, unless the graph is dense enough that e is significantly greater than v^2 it is more efficient to use an adjacency list implementation.

3. The parallel implementation iterates on all vertices for each vertex checked. This makes it very easy to divide into independent segments for each thread to operate on. The adjacency list implementation, however, iterates by the number of edges adjacent to the vertex currently being worked on. This makes it highly efficient for serial implementations, but renders it difficult to run in parallel, as each iteration can only be divided as many times as there are adjacent edges. This is both irregular—and therefore awkward to parallelize—and not worth the thread overhead if the graph is not exceptionally dense.