Page 4: [1] Comment [7] KU USER 3/8/15 11:55 AM

I don't know whether the run-time dynamics of this system lend themselves to the following approach, but Dylan & I applied the following to a search problem that had been using BlockingQueue, and we are applying them again this semester in CSC543 to a parallel state machine (a cellular automaton, triggered partially by my advising of Dan Rassler's senior seminar project last semester, including some of my ideas that he never got around to implementing). The idea is this: If the worker threads that are reading the queues typically find data in the queues during the work cycle, then use a non-blocking ConcurrentLinkedQueue that the threads poll() to get their next work item. If the threads have non-busy times, have them block somewhere else (we are using a CyclicBarrier in the current CSC543 sequence of projects) until the next round of work. Also, in both Dylan/my case and this semester, I have been able to get another doubling of performance by giving each worker thread its own nonblocking ConcurrentLinkedQueue from which to poll() work. This eliminates contention among the worker threads for access to a shared ConcurrentLinkedQueue. See Graph 4 in http://faculty.kutztown.edu/parson/pubs/BidSearchPDPTA2012ParsonSchwesinger.pdf . Just going to multiple nonblocking queues cuts time in half from the single nonblockingqueue approach. Also, Graph 3 compares blocking queue to a single ConcurrentLinkedQueue. The dotted "minimum blocking" in Graphs 3 & 4 is the same curve; it uses a single ConcurrentLinkedQueue that is poll()ed for work. The "blocking queue" uses LinkedBlockingQueue and blocks for work, but is otherwise the same algorithm. The "multiple queues" in Graph 4 is the same as "minimum blocking" except that its uses one ConcurrentLinkedQueue for each worker thread. It is actually engineered to minimize thread contention at each side of each ConcurrentLinkedQueue. The writes into each ConcurrentLinkedQueue happen in round-robin order, controlled by an AtomicInteger index that we increment using (+1) % numberofQueues, for two reasons: in avoid thread contention for a ConcurrentLinkedQueue in enqueuing, and to distribute the workload evenly. The reads from each ConcurrentLinkedQueue take place only by the since worker thread feeding from that queue in the "multiple queues" approach, so there is absolutely 0 contention for access to a given ConcurrentLinkedQueue object. See also the activity diagram on page 7 of http://faculty.kutztown.edu/parson/spring2015/csc543spring2015assn2.pdf . It shows busy poll()ing of a queue during busy work times (queue has a high probability of having a work item when polled). I needed to outline the algorithm without giving it away; pseudocode would have been too close to the Java solution, so I used an activity diagram. Here are my stats from that assigment for four configurations on hermione:

HANDOUT, NON-THREADED CODE:

```
 113.57user 1.56system 1:52.81elapsed 102%CPU
```

4 worker threads using a LinkedBlockingQueue

```
284.61user 32.86system 1:19.66elapsed 398%CPU
```

4 worker threads, this time polling a single nonblocking ConcurrentLinkedQueue, and blocking in a CyclicBarrier when there is no work. The students are writing this based on the non-threaded version.

```
287.15user 4.57system 1:04.62elapsed 451%CPU
```

4 worker threads, this time each is polling its own, dedicated nonblocking ConcurrentLinkedQueue

```
226.41user 4.03system 1:00.39elapsed 381%CPU
```

The differences are more pronounced on harry. Also, some of thesse scale better with more threads than others. The last solution tends to scale well, just as this approach did in the paper with Dylan. Conclusion: use poll()ing ConcurrentLinkedQueue is work comes in bursts when there is a high probability of work in a queue, waiting in some other blocker during the idle periods; and, dedicate one ConcurrentLinkedQueue to each worker queue.