

CSC 402 - Data Structures II, Fall, 2010
Overview of some multithreaded data structures, Dr. Dale E. Parson

~parson/multip/multip/initial_qsort/thread_qsort.cxx

```
1  /*      thread_qsort.cxx -- A test driver of multithreaded quick sort.
2  */
96  struct tms before, after ;
97      double rtime, utime, stime ;
98      int pre = times(&before);
99      quicksort(intarray, 0, numberOfIntegers-1);
100     int post = times(&after);
101     rtime = ((double)(post-pre))/((double)CLK_TCK ;
102     utime = ((double)(after.tms_utime-before.tms_utime))/((double)CLK_TCK;
103     stime = ((double)(after.tms_stime-before.tms_stime))/((double)CLK_TCK;
104     cout << "RUS," << setprecision(7) << showpoint
105          << rtime << " " << utime << " " << stime << endl ;

110 // If there are at least 3 elements, pick a value that is >= one
111 // of them and <= the other as the pivot. This decreases the probability
112 // of hitting a degenerate case where all or almost all elements partition
113 // to one side of the pivot.
114 // The return value is the pivot, also swapped into iarray[left].
115 static int pickpivot(int iarray[], int left, int right) {
116     int result ;
117     int count = right - left + 1 ;
118     int mid = (left + right) / 2 ;
119     if (count < 3) {
120         result = iarray[left]; // [1] element or 50/50 odds for [2]
121     } else if ((iarray[mid] <= iarray[left] && iarray[left] <= iarray[right])
122              || (iarray[right] <= iarray[left] && iarray[left] <= iarray[mid])){
123         result = iarray[left];
124     } else if ((iarray[left] <= iarray[mid] && iarray[mid] <= iarray[right])
125              || (iarray[right] <= iarray[mid] && iarray[mid] <= iarray[left])){
126         result = iarray[mid] ;
127         iarray[mid] = iarray[left] ;
128         iarray[left] = result ;
129     } else if ((iarray[mid] <= iarray[right] && iarray[right] <= iarray[left])
130              || (iarray[left] <= iarray[right] && iarray[right] <= iarray[mid])){
131         result = iarray[right] ;
132         iarray[right] = iarray[left] ;
133         iarray[left] = result ;
134     }
135     return result ;
136 }
```

```

138 // partition is a helper function for quicksort().
139 // partition partitions an array into a left "half" and a right "half,"
140 // where all values in the left "half" or <= some "pivot" value, and
141 // all values in the right "half" are > that "pivot" value.
142 // partition may not divide the array exactly in half; left and right
143 // "part" would be a better term.
144 // RETURN index of the element around which the array is partitioned.
145 static int partition(int iarray[], int left, int right) {
146     int pivot = pickpivot(iarray, left, right);
147     int low = left + 1 ;           // scan up looking for a too-high value
148     int high = right ;           // scan down looking for a too-low value
149
150     while (low < high) {         // scan from both sides of array
151         while (low <= high && iarray[low] <= pivot) { // in correct side
152             low++;
153         }
154         while (low <= high && iarray[high] > pivot) { // in correct side
155             high--;
156         }
157         // If there are two elements on the wrong side, swap them.
158         if (low < high) {
159             int temp = iarray[high];
160             iarray[high] = iarray[low];
161             iarray[low] = temp ;
162         }
163     }
164     // We still need to place the pivot in the correct spot.
165     while (high > left && iarray[high] >= pivot) {
166         high--;
167     }
168     // swap pivot with a lower value if there is one
169     if (pivot > iarray[high]) {
170         iarray[left] = iarray[high];
171         iarray[high] = pivot ;
172         return high;
173     } else {
174         return left ;
175     }
176 }
177
178 struct thread_param {           // always allocate on a non-returned stack frame!
179     int * iarray ;             // subarray to be sorted
180     int left, right ;         // it's left and right indices
181 };
182
183 static void *threadstart(void *voidarg) {

```

```

184     thread_param *arg = (thread_param *) voidarg ;
185     // sort partition in this thread
186     quicksort(arg->iarray, arg->left, arg->right);
187     return 0 ; // thread terminates
188 }
189
190 // Implement the quick sort algorithm.
191 static void quicksort(int iarray[], int left, int right) {
192     int count = right - left + 1 ;
193     if (left < right) {
194         // Partition array so all values <= iarray[pivotlocation] are left
195         // of pivotlocation, and all values > pivotlocation are right of it.
196         int pivotlocation = partition(iarray, left, right);
197         // Sort the left partition.
198         if (count >= threadingThreshold) {
199             // Create another thread to sort the left partition.
200             // Package up arguments for the thread:
201             pthread_t child ;
202             thread_param thread_arg ;
203             thread_arg.iarray = iarray ;
204             thread_arg.left = left ;
205             thread_arg.right = pivotlocation-1 ;
206             cerr << "SPAWNING A THREAD " << (pivotlocation - left)
207                 << " ON LEFT, " << (right - pivotlocation) << " ON RIGHT."
208                 << endl ;
209             if (pthread_create(&child, 0, threadstart, &thread_arg)) {
210                 cerr << "ERROR STARTING A THREAD." << endl ;
211                 exit(THREADERR);
212             }
213             threadcount++ ;
214             // Sort the right partition while the child thread sorts the left.
215             quicksort(iarray, pivotlocation+1, right);
216             // Wait for the child to complete the left and terminate.
217             if (pthread_join(child, 0)) {
218                 cerr << "ERROR JOINING A THREAD." << endl ;
219                 exit(THREADERR);
220             }
221         } else {
222             // Sort the left partition.
223             quicksort(iarray, left, pivotlocation-1);
224             // Sort the right partition.
225             quicksort(iarray, pivotlocation+1, right);
226         }
227     }
228 }

```

~parson/multip/multip/threadlim_mrgsort/thread_mergesort.cxx

```
1  /*      thread_mergesort.cxx -- A test driver of multithreaded merge sort.
2  */
91  struct tms before, after ;
92      double rtime, utime, stime ;
93      int pre = times(&before);
94      // The main thread counts as 1 thread.
95      // printarray(intarray, numberOfIntegers);
96      // exit(0);
97      imergesort(intarray, 0, numberOfIntegers-1, nthreads);
98      // printarray(intarray, numberOfIntegers);
99      // exit(0);
100     int post = times(&after);
101     rtime = ((double)(post-pre))/(double)CLK_TCK ;
102     utime = ((double)(after.tms_utime-before.tms_utime))/(double)CLK_TCK;
103     stime = ((double)(after.tms_stime-before.tms_stime))/(double)CLK_TCK;
104     cout << "RUS," << setprecision(7) << showpoint
105          << rtime << "," << utime << "," << stime << endl ;

108
109     struct thread_param {      // always allocate on a non-returned stack frame!
110         int * iarray ;      // subarray to be sorted
111         int * aisle0 ;      // left temporary array
112         int * aisle1 ;      // right temporary array
113         int count ;
114         int runlen ;
115     } ;
116

122
123     // This is the mergesort's splitphase for the assignment.
124     static void splitphase(queueOfInts &merger, queueOfInts splitter[2],
125         int runlen) {
126         bool ignoreme ;      // We only peek on queues with data, etc.
127         int qid = 0 ;
128         while (merger.size() > 0) {      // while there are still runs to split ;
129             int rl = (merger.size() < runlen) ? merger.size() : runlen ;
130             while (rl > 0) {
131                 splitter[qid].enqueue(merger.peek(ignoreme));
132                 merger.dequeue();
133                 rl-- ;
134             }
135             qid = qid ^ 1 ; // switch to other splitter
136         }
137     }
```

```

138
139 // This is the mergesort's mergephase for the assignment.
140 void mergephase(queueOfInts &merger, queueOfInts splitter[2],
141     int runlen) {
142     bool ignoreme ;
143     int qtodrain ; // The queue to drain when 1 runs ends,
144     while (splitter[0].size() > 0 && splitter[1].size() > 0) {
145         int srl[2] ; // remaining run lengths for each queue
146         srl[0] = runlen ; // both sides have contents, [0] is a full run
147         srl[1] = (splitter[1].size() < runlen) ? splitter[1].size() : runlen ;
148         while (srl[0] > 0 && srl[1] > 0) {
149             int v0 = splitter[0].peek(ignoreme);
150             int v1 = splitter[1].peek(ignoreme);
151             if (v0 <= v1) {
152                 merger.enqueue(v0);
153                 splitter[0].dequeue();
154                 srl[0] -= 1 ;
155             } else { // v1 is less
156                 merger.enqueue(v1);
157                 splitter[1].dequeue();
158                 srl[1] -= 1 ;
159             }
160         }
161         int qtodrain = (srl[0] > 0) ? 0 : 1 ;
162         while (srl[qtodrain] > 0) {
163             merger.enqueue(splitter[qtodrain].peek(ignoreme));
164             splitter[qtodrain].dequeue();
165             srl[qtodrain] -= 1 ;
166         }
167     }
168     // splitter[0] may have a leftover run or partial run
169     while (splitter[0].size() > 0) {
170         merger.enqueue(splitter[0].peek(ignoreme));
171         splitter[0].dequeue();
172     }
173 }
174
175 // This is the function specified for the assignment.
176 void mergesort(int *arrayToSort, int *aisle0, int *aisle1, int count,
177     int runlen) {
178     queueOfInts merger(arrayToSort, count, count);
179     queueOfInts splitter[] = {
180         queueOfInts(aisle0, count, 0),
181         queueOfInts(aisle1, count/2, 0)
182     };
183     while (runlen < merger.size()) {

```

```

184     splitphase(merger, splitter, runlen);
185     mergephase(merger, splitter, runlen);
186     runlen *= 2 ;
187 }
188 }
189
190 static void *threadstart(void *voidarg) {
191     thread_param *arg = (thread_param *) voidarg ;
192     // sort partition in this thread
193     mergesort(arg->iarray, arg->aisle0, arg->aisle1, arg->count, arg->runlen);
194     return 0 ; // thread terminates
195 }
196
197 // This function prepares multiplem threads to invoke mergesort.
198 void imergesort(int *iarray, int left, int right, int nthreads) {
199     if (iarray == NULL || left > right) {
200         // Don't try to do pointer arithmetic on a NULL pointer.
201         return ;
202     }
203     int *arrayToSort = iarray + left ;
204     int count = right - left + 1 ;
205     int *aisle0 = new int [ count ]; // Like an "aisle" in the classroom.
206     int *aisle1 = new int [ count/2 ]; // Never more than 1/2 the elements.
207     // Here comes the parallel adaptation of the serial algorithm.
208     // D. Parson -- the iterative version of mergesort has been easier
209     // for students to understand, especially in external sorting,
210     // so I have adapted that rather than the usual textbook-recursive sort.
211     pthread_t *child = new pthread_t [ nthreads ];
212     // Following are thread startup arguments.
213     thread_param *chargs = new thread_param [ nthreads ] ;
214     int runlen = 1 ;
215     int numruns = count ;
216     int usethreads = nthreads ;
217     while (runlen < count) {
218         int halfruns = ((numruns & 1) ? ((numruns+1)/2) : (numruns/2));
219         if (usethreads > halfruns) {
220             usethreads = halfruns ;
221         }
222         int intspertthread = count / usethreads ; // truncating divide
223         int halfintspertthread = intspertthread / 2 ;
224         int *subarray = arrayToSort ;
225         int *subaisle0 = aisle0 ;
226         int *subaisle1 = aisle1 ;
227         int leftovers = count - (intspertthread * (usethreads-1));
228         int chld ;
229         for (chld = 0 ; chld < (usethreads-1) ; chld++) {

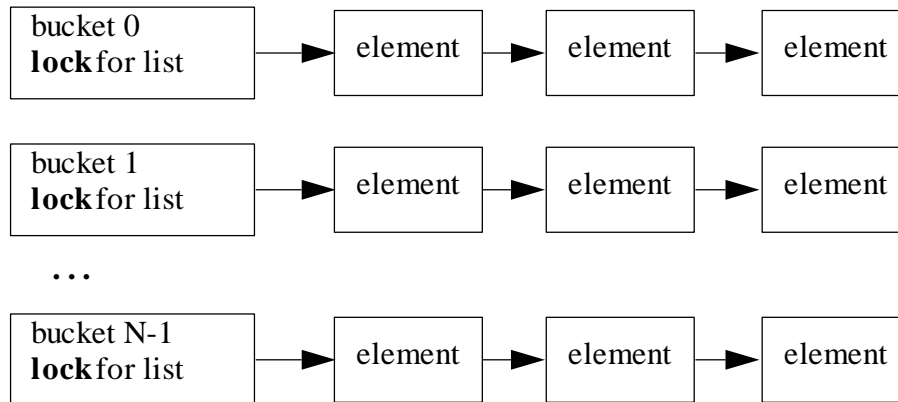
```

```

230         chargs[chld].iarray = subarray ;
231         chargs[chld].aisle0 = subaisle0 ;
232         chargs[chld].aisle1 = subaisle1 ;
233         chargs[chld].count = intspertthread ;
234         chargs[chld].runlen = runlen ;
235         subarray += intspertthread ;
236         subaisle0 += intspertthread ;
237         subaisle1 += halfintspertthread ;
238     }
239     // The final partition for the main thread might be bigger.
240     chargs[chld].iarray = subarray ;
241     chargs[chld].aisle0 = subaisle0 ;
242     chargs[chld].aisle1 = subaisle1 ;
243     chargs[chld].count = leftovers ;
244     chargs[chld].runlen = runlen ;
245     for (chld = 0 ; chld < (usethreads-1) ; chld++) {
246         if (pthread_create(&(child[chld]),0,threadstart,&(chargs[chld]))) {
247             cerr << "ERROR STARTING A THREAD." << endl ;
248             exit(THREADERR);
249         }
250     }
251     // Final partition goes to main thread.
252     mergesort(chargs[chld].iarray, chargs[chld].aisle0,
253             chargs[chld].aisle1, chargs[chld].count, chargs[chld].runlen);
254     for (chld = 0 ; chld < (usethreads-1) ; chld++) {
255         if (pthread_join((child[chld]), 0)) {
256             cerr << "ERROR JOINING A THREAD." << endl ;
257             exit(THREADERR);
258         }
259     }
260     runlen = intspertthread ;
261     numruns = usethreads ;
262 }
263 delete [] aisle0 ;
264 delete [] aisle1 ;
265 }

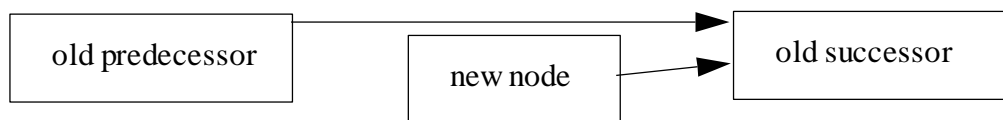
```

A closed address (linked list) hash table can place a lock on every linked list, allowing only one thread at a time into each bucket's list. As long as concurrent accesses are scattered, the probability of colliding on a lock -- which entails waiting -- is low. A *striped* approach to locking allocates a fixed number **NL** of locks shared by one or more buckets, where **NL** does not grow with table growth. A bucket's lock is found by $\mathbf{NL} = \mathbf{bucketNumber} \% \mathbf{NL}$. This approach reduces memory overhead for locks with an increase in likelihood of waiting for a busy lock.



The reason that quicksort and mergesort as coded above do not need locks is because concurrent threads do not access shared data. Each partition of an array in quicksort or mergesort is separate from every other partition, allowing unsynchronized access. The only synchronization occurs when two threads *join* after they complete their work.

Some data sharing does not require the storage overhead and potential waiting overhead of locks. It is possible to avoid locking nodes in a linked list when they are being read, using locks to *serialize* writes to the nodes. Writer threads first act as readers until they find the node(s) that they must mutate. They then acquire a lock on any node to be mutated, re-check to ensure that no other concurrent writer thread has mutated the node in the meantime (e.g., by using a mutation counter in the node), and then perform the mutation. The mutation must be done in a series of steps that will not send readers to invalid nodes. Lockless mutation is easier in a language with garbage collection (Java or Jython), because a reader thread that accesses a deleted node will not crash. (It may access recently stale data -- temporal sequence becomes a partial ordering in concurrent processes.) In C++ if a mutator deletes a departing node, then a concurrent reader might crash when accessing that node. C++ requires reader locks or more complex solutions in such cases.



Insertion adjusts old predecessor's *next* link after setting the new node's link to avoid read locks.

Deletion adjusts the predecessor's *next* link around the node to be deleted, allowing the garbage collector to recover the node after all references (possibly by reader threads) to the node being deleted are gone. This approach requires link updates to be *atomic operations*, requires an *atomic isDeleted* boolean flag on a node, an requires complicated locking when 2 writers collide, e.g., a per-node *mutation counter*. Skiplists are preferable to balanced trees for concurrent threading, because lists support concurrency better. Tree balancing requires a global lock.