MULTICORE PROGRAMMING

Optimizing lock-based KCAS with HTM, OpenMP, debugging/perf

Lecture 16

Trevor Brown

LAST TIME

- Try-locks
    - Naïve (incorrect) KCAS using these
- **Versioned** (try-)locks
    - KCAS using these

THIS TIME

- Accelerating version-lock-based KCAS with HTM
  - Purpose: to demonstrate accelerating a **lock-based** algorithm using HTM!
- OpenMP
- Debugging and performance
  - Checksums/validation

USING HTM TO ACCELERATE TRY-LOCK BASED KCAS

- Similar to accelerating the lock-free algorithm (but a bit simpler)
- Goal
  - HTM-based KCAS that uses **try-lock based KCAS as the fallback path**
- Approach
  - Fast path algorithm:
    - Wrap try-lock based KCAS in a transaction: xstart ; KCAS ; xend
    - Now each KCAS on the fast path is atomic, just because it is in a transaction
    - Get rid of parts of the algorithm that are unnecessary

ADDING TRANSACTIONS

```
template <int K>
bool KCAS(addr_t ** addr, val_t * expv, val_t * newv) {
  int retries = 5;
  int status;
retry:
  if ((status = _xbegin()) == _XBEGIN_STARTED) {
    bool result = KCAS_txn<K>(addr, expv, newv);
    _xend();
    return result;
  } else {      // transaction aborted
    if (--retries >= 0) goto retry;
    bool result = KCAS_locks<K>(addr, expv, newv);
    return result;
  }
}
```

Fast path: try-lock based KCAS code inside a transaction

Fallback path: try-lock based KCAS code

(We use the same **version-lock-based KCASRead** for both code paths)

KCAS_txn is initially **the same as KCAS_locks**. We will optimize it.

Before retrying, should wait a bit…

Idea: if we aborted because a lock was held, could we wait until that lock is released?

But that information was lost when we aborted…

Maybe we could sneak it out using an explicit _xabort() status code?

Could pass an index < 127 into addr[] to _xabort!

TURNING LOCK ACQUISITION INTO READING

```
template <int K>
bool KCAS_txn(addr_t ** addr, val_t * expv, val_t * newv) {
  retry:
  TryLockSet<K> tls;          // auto-unlock when this goes out of scope
  for (int i=0;i<K;++i) {     // try to acquire locks
    if (!tls.tryLock(addr[i]->addrOfLock()))
      goto retry;             // (release locks and) retry
    if (addr[i]->value != expv[i])
      return false;           // (release locks and) fail
  }
  for (int i=0;i<K;++i) {
    *addr[i] = newv[i];
  }
  return true;                // (release locks and) succeed
}
```

No need to lock!
HTM ensures atomicity.
However, we must **respect others' locks**.
This becomes a **read**.

With no locking, we don't need TryLockSet

But, be careful! Still need to increment **version numbers** so fallback path can tell we changed addresses!

We modify the versioned try-lock implementation to **expose the "lock" word…**

TURNING RETRIES INTO ABORTS

```
template <int K>
bool KCAS_txn(addr_t ** addr, val_t * expv, val_t * newv) {
  retry:

  for (int i=0;i<K;++i) {
    if (addr[i]->addrOfLock()->lock & 1)
      goto retry;
    if (addr[i]->value != expv[i])
      return false;
  }
  for (int i=0;i<K;++i) {
    *addr[i] = newv[i];
    addr[i]->addrOfLock()->lock += 2; // update version numbers
  }
  return true;
}
```

No point retrying here, since we can only make progress if the lock is released, **which will abort us**. So, just **xabort**.

So, no need for the "**retry:**" label anymore.

OPTIMIZED CODE

```
template <int K>
bool KCAS_txn(addr_t ** addr, val_t * expv, val_t * newv) {


  for (int i=0;i<K;++i) {
    if (addr[i]->addrOfLock()->lock & 1)
      _xabort(i);
    if (addr[i]->value != expv[i])
      return false;
  }
  for (int i=0;i<K;++i) {
    *addr[i] = newv[i];
    addr[i]->addrOfLock()->lock += 2; // update version numbers
  }
  return true;
}
```

User defined _xabort status code (assuming i < 127)

CLEANED UP THE SPACING/COMMENTS

```
template <int K>
bool KCAS_txn(addr_t ** addr, val_t * expv, val_t * newv) {
  for (int i=0;i<K;++i) {
    if (addr[i]->addrOfLock()->lock & 1) _xabort(i);   // read locks
    if (addr[i]->value != expv[i]) return false;       // read values
  }
  for (int i=0;i<K;++i) {
    *addr[i] = newv[i];                        // update values
    addr[i]->addrOfLock()->lock += 2; // update version numbers
  }
  return true;
}
```

USING OUR NEW _XABORT STATUS CODE

```
template <int K>
bool KCAS(addr_t ** addr, val_t * expv, val_t * newv) {
  int retries = 5;
  int status;
retry:
  if ((status = _xbegin()) == _XBEGIN_STARTED) {
    bool result = KCAS_txn(addr, expv, newv);
    _xend();
    return result;
  } else {     // transaction aborted
    if (status & _XABORT_EXPLICIT) {
      int idx = _XABORT_CODE(status);
      while (addr[idx]->addrOfLock()->read() & 1) { /* wait */ }
    }
    if (--retries >= 0) goto retry;
    bool result = KCAS_locks<K>(addr, expv, newv);
    return result;
  }
}
```

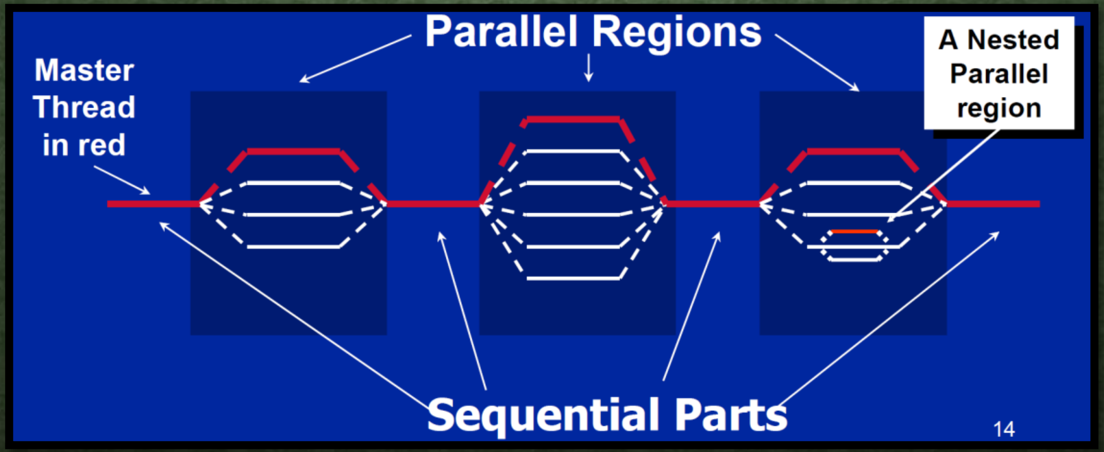
If the abort was caused by a **user _xabort** call

Get the user-provided abort code

Wait until the returned index is no longer locked

OPENMP (OPEN MULTI PROCESSING)

A library for fork-join parallelism



OVERVIEW

- Easy to use in your own C/C++ projects
- Tons of features; we just look at a couple of simple tools
    1. Parallel sections
    2. Parallel for loops
    3. Reductions
    4. OMP Single, OMP Tasks
- Usage:
    - #include <omp.h>
    - GCC: compile with **-fopenmp**

Warning: **Windows Subsystem for Linux v1** appears to use a **global lock** in its implementation of OpenMP.
It offered **no speedup** in my testing.
(Try running on a real Linux box or a virtual machine!)

1. PARALLEL SECTION

- Shortcut for: **spawning n threads**, where n = # of logical processors in the system, having them all execute the **same code block**, and then **joining them**

Sequential code

```
#include <cstdio>
void main() {
  int id = 0;
  printf(" hello(%d)", id);
  printf(" world(%d)\n", id);
}
```

Output:
“ hello(0) world(0)”

Concurrent/parallel code

```
#include <omp.h>
void main() {
  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    printf(" hello(%d)", id);
    printf(" world(%d)\n", id);
  }
}
```

Spawn n threads

What is ***my*** thread ID (in 0..n-1)?

n threads execute this

Stop n threads

Output?

PARALLEL OUTPUT

```
hello(119) hello(69) hello(95) hello(15) hello(68) hello(32) hello(77) hello(61) hell
hello(138) hello(5) world(119)
world(95)
world(15)
hello(67) world(138)
hello(131) hello(27) hello(42) hello(80) world(115)
hello(112) hello(78) hello(110) hello(113) world(131)
hello(23) hello(109) hello(26) hello(12) world(110)
hello(16) hello(35) hello(55) hello(13) hello(38) world(16)
hello(45) hello(99) hello(105) world(45)
hello(40) world(40)
hello(4) world(4)
hello(3) world(3)
hello(82) world(82)
hello(1) world(1)
hello(11) world(11)
hello(21) world(21)
hello(39) world(39)
hello(81) world(81)
hello(22) world(22)
hello(2) world(2)
hello(9) world(9)
hello(17) world(17)
hello(83) world(83)
hello(36) world(36)
hello(10) world(10)
```

WHY IS THIS USEFUL?

Using OpenMP

```
#pragma omp parallel
doSomething();
```

Equivalent code using *pthreads*

```
const int n = SomehowGetNumLogicalProcessors();
pthread_t *threads = new pthread_t[n];
for (int i=0;i<n;++i) {
  if (pthread_create(&threads[i], NULL, doSomething)) {
    std::cerr<<"ERROR: could not create thread"<<std::endl;
```

Equivalent code using *std::thread*

```
const int n = SomehowGetNumLogicalProcessors();
std::thread * threads = new std::thread[n];
for (int i=0;i<n;++i) {
  threads[i] = new std::thread(doSomething)
}
for (int i=0;i<n;++i) {
  threads[i]->join();
}
delete[] threads;
```

2. PARALLEL FOR

```
matrix * multiply(matrix * o) {
    matrix * ret = new matrix(h, o->w);
    #pragma omp parallel for
    for (int y=0; y < ret->h; ++y) {
        for (int x=0; x < ret->w; ++x) {
            for (int k=0; k < w; ++k) {
                ret->data[y][x] += data[y][k] * o->data[k][x];
            }
        }
    }
    timer.split("multiply call finished");
    return ret;
}
```

OpenMP *automatically* decides **how many** threads (***n***) to spawn

It splits work evenly between the **n** threads (each does approx. $\frac{ret \to h}{n}$ iterations of this loop)

Implicit fork (thread spawning)

Implicit join

WHY IS THIS USEFUL?

Using OpenMP

```
#pragma omp parallel for
for (long i=0;i<n;++i)
  loop_body(i);
```

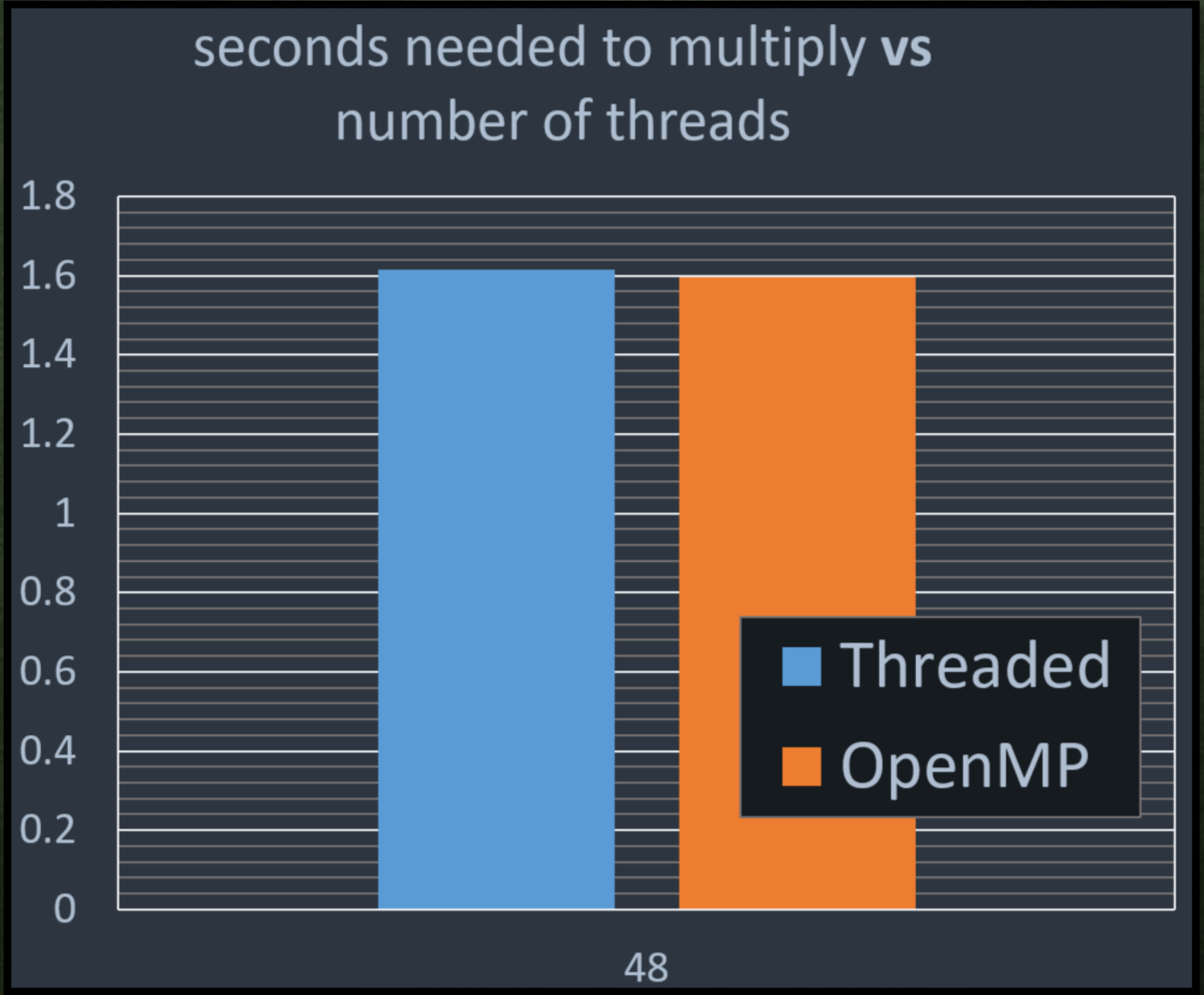
Without using OpenMP: starting threads to run loop_body

```
const int numThreads = GetNumberOfLogicalProcessors();
std::thread * threads = new std::thread[numThreads];
for (int i=0;i<numThreads;++i) {
  threads[i] = new std::thread(loop_body);
}
for (int i=0;i<numThreads;++i) {
  threads[i].join();
}
delete[] threads;
```

Without using OpenMP: loop_body

```
int id = getMyThreadID();
int istart = id * n / numThreads;
int iend = (id+1) * n / numThreads;
if (id == numThreads-1) iend = n;
for (int i=istart; i<iend; i++)
  original_loop_body(i);
```

PERFORMANCE OF OPENMP VS MANUAL THREADING



3. REDUCTIONS

- A reduction takes a vector (array) and turns it into a scalar (single number)

Example: summing an array

```
long sum = 0;
for (long i=0;i<n;++i) {
  sum += array[i];
}
```

Naïve use of OpenMP

```
long sum = 0;
#pragma omp parallel for
for (long i=0;i<n;++i) {
  sum += array[i];
}
```

Problem: many threads do this, and this increment is not atomic!

How about using fetch&add?

```
long sum = 0;
#pragma omp parallel for
for (long i=0;i<n;++i) {
  __sync_fetch_and_add(&sum, array[i]);
}
```

Problem: correct, but not very scalable!

Ideally: want a thread to maintain a **local sum** while processing a **batch**, and fetch&add its local sum into the **global sum** at the end of the batch

OPENMP REDUCTIONS

- OpenMP natively supports reductions over numerous operators (+, *, &, |, …)
- Must tell OpenMP which variable will be used to store the reduction

Example: summing an array

```
long sum = 0;
for (long i=0;i<n;++i) {
  sum += array[i];
}
```

1 thread, time to complete: **4330 ms**

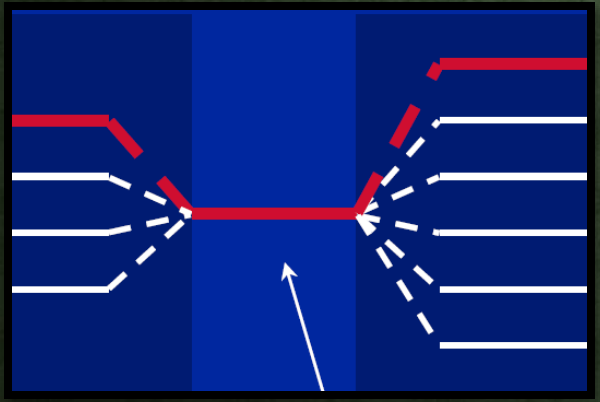
Correct OpenMP reduction

```
long sum = 0;
#pragma omp parallel for reduction (+:sum)
for (long i=0;i<n;++i) {
  sum += array[i];
}
```

48 threads, time to complete: **185ms**

4(A). OPENMP SINGLE

- Sometimes you want a **single threaded** computation in the middle of your parallel computation



```
#pragma omp parallel
{
    multiThreadedStuff1();

    #pragma omp single
    {
        singleThreadedStuff();
    }

    multiThreadedStuff2();
}
```

4(B). OPENMP TASK

- Define a **task** that should be completed by **any available thread** in a parallel section
- Common design pattern: one thread generates & launches tasks, tasks run in parallel

```
#pragma omp parallel
{
    #pragma omp single
    {
        for (linked_list_node n = head; n != NULL; n = n->next) {
            #pragma omp task
            processTheNode(n);
        }
    }
}
```

Spin up **n** threads to run tasks

Single thread generates tasks

This function call, with this argument, becomes a **task**, and is run in the background (by one of the **n-1 other threads**)

This closing brace **waits** for all tasks to complete

Can also **manually** wait for all tasks, whenever you like, with **`#pragma omp taskwait`**

OMP task tutorial: https://openmp.org/wp-content/uploads/sc13.tasking.ruud.pdf

TOOLS FOR DEBUGGING AND PERFORMANCE

- Debugging and optimizing concurrent programs is **very** hard. Tools can help!

- Debugging
  - GNU Debugger (GDB)
    - Segfaults, infinite loops
  - Address Sanitizer (ASan)
    - Segfaults, memory leaks
    - 1~2x slowdown
  - Valgrind
    - Segfaults, memory leaks, **memory access errors**
    - many-x slowdown
  - Graphviz
    - **Visualizing** pointer based data structures

- Performance
  - Linux Perftools (perf)
    - Studying cycles, cache misses, instructions, stalled cycles
    - At the whole-application level
  - C/C++ Performance API (PAPI)
    - Precise information from perf, but recorded **within** your program
  - VTune Amplifier
    - Powerful (and now free!) profiler

A lot of errors in concurrent programs manifest as memory access errors! For example, a thread may write a bad value into a pointer because of a concurrency bug, and another thread may then read it.

DEBUGGING TOOLS

USING VALGRIND TO FIND MEMORY ACCESS ERRORS

```
$ valgrind --fair-sched=yes ./a1code_segfault/workload_timed.out 4 1000 naive
==107893== Command: ./a1code_segfault/workload_timed.out 4 1000 naive
==107893==
==107893== Use of uninitialised value of size 8
==107893==    at 0x510F0D4: std::thread::join() (in /.../x86_64-linux-gnu/libstdc++.so.6.0.22)
==107893==    by 0x1092DC: void runExperiment<CounterNaive>(...) (workload_timed.cpp:46)
==107893==    by 0x108E3B: main (workload_timed.cpp:70)
==107893==
==107893== Invalid read of size 8
==107893==    at 0x510F0D4: std::thread::join() (in /.../x86_64-linux-gnu/libstdc++.so.6.0.22)
==107893==    by 0x1092DC: void runExperiment<CounterNaive>(...) (workload_timed.cpp:46)
==107893==    by 0x108E3B: main (workload_timed.cpp:70)
==107893==  Address 0x190 is not stack'd, malloc'd or (recently) free'd
...
```

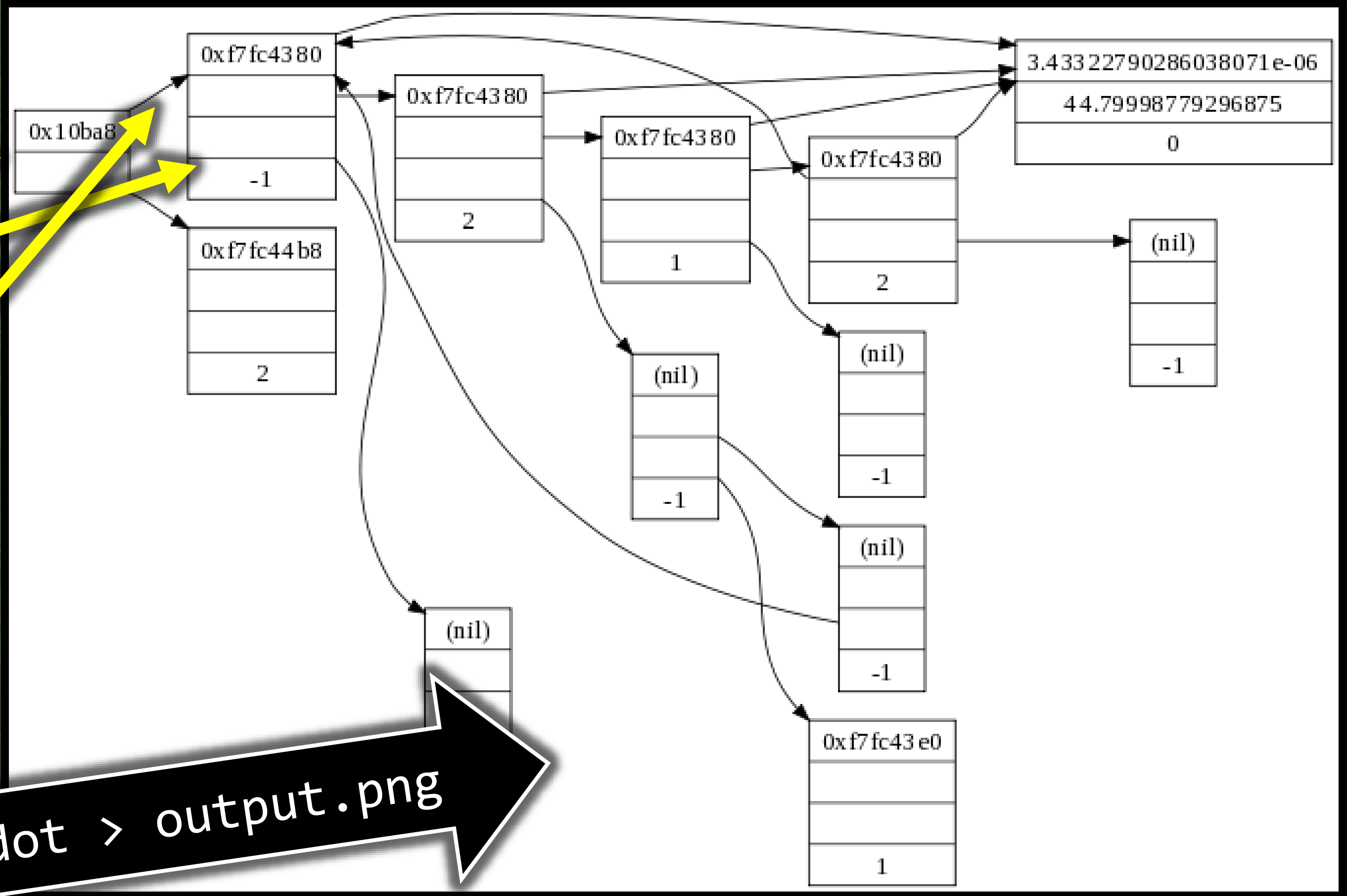
- Typical first step in debugging any error that isn't obvious:
  - Ensure that valgrind runs without any such errors.
  - If there **are** such errors, **fix those first!**

Using Address Sanitizer to check for memory leaks

```
$ g++ -pthread -g -fsanitize=address -static-libasan -fopenmp -O3 ex2_mmult_threads.cpp
$ ./a.out 24
matrix created: 0.02s
randomize call finished: 0.10s
...
multiply call finished: 2.82s

=================================================================
==76549==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 192 byte(s) in 24 object(s) allocated from:
    #0 0x555b294992d8 in operator new(unsigned long) (/home/.../a.out+0xc82d8)
    #1 0x555b294dab9e in matrix::multiply(matrix*, int) (/home/.../ex2_mmult_threads.cpp:12)

SUMMARY: AddressSanitizer: 192 byte(s) leaked in 24 allocation(s).
```

GRAPHVIZ: WHEN YOU JUST NEED TO SEE IT

```
digraph g {
  node [
    fontsize = "16"
    shape = “record"
  ];
  edge [];
  "node0" [
    label = "<f0> 0x10ba8|<f1>“
  ];
  "node1" [
    label = "<f0> 0xf7fc4380|<f1>|<f2>|-1"
  ];
  [...]
  "node0":f0 -> "node1":f0 [
    id = 0
  ];
  [...]
}
```



`$ dot -Tpng input.dot > output.png`

Tip: try to get your toughest bugs to happen in SMALL data structures, so you can graphviz them

SANITY CHECKING: EXPERIMENT CHECKSUMS

- Important to perform sanity checks wherever you can!
  - Helps to catch obvious (and non-obvious) mistakes
- One good sanity check: checksum based validation
  - Reduce the **data structure** to a number (a **data structure checksum**)
  - Reduce each threads' completed operations to a number (a **thread checksum**)
  - verify that thread checksums **"match"** the data structure checksum
  - (I.e., the work the threads **think** they've done is reflected **in the data structure**!)
- **Creativity needed to come up with good checksum functions**

EXAMPLE: SYNTHETIC KCAS BENCHMARK

- **n** threads repeatedly do the following for 3 seconds
  - Pick **K** uniform random slots in an array
  - Read integers stored in those slots
  - Do a KCAS to change each of the K slots from the value **exp** that we read, to a new value of **exp + 1**
- Report average throughput (KCAS operations/sec) over all trials

EXAMPLE: CHECKSUM VALIDATION FOR SUCH A BENCHMARK

- **Data structure checksum**
  - Sum of all array entries
  - Each successful KCAS increments **k** array slots by 1
    - Should add **k** to the **data structure checksum**
- **Thread checksum**
  - kX where X = # of successful KCAS operations performed by the thread
    - Each successful KCAS should **add k** to the **thread checksum**
- **Validation**
  - sum(thread checksums) == data structure checksum **?**
  - (If a KCAS operation is lost, or screws up the array, validation [hopefully] fails)