1. Introduction. This is a solver for Rubik's Cube, using Herbert Kociemba's two-phase algorithm.

```
⟨ twophase.cpp 1 ⟩ ≡
  const char *BANNER = "This␣is␣twophase␣1.1,␣(C)␣2010-2012␣Tomas␣Rokicki.␣␣All␣Rig\
      hts␣Reserved.";
#include "phase1prune.h"
#include "phase2prune.h"
#include <pthread.h>
#include <iostream>
#include <map>
#include <cstdio>
  using namespace std;
  ⟨ Data declarations 2 ⟩
  ⟨ Utility functions 10 ⟩
  ⟨ Solver class 11 ⟩
  int main(int argc, char *argv[])
  {
    double progstart = walltime();
    duration();
    ⟨ Parse arguments 3 ⟩
    ⟨ Initialize the program 6 ⟩
    ⟨ Handle the work 34 ⟩
    ⟨ Print summary 30 ⟩
  }
```

2. There is actually very little cube code in this program. Almost all the work is handled by the code in `cubepos`, `kocsymm`, `phase1prune`, and `phase2prune`; the code in this file is predominantly bookkeeping, thread management, and the like. The first thing we take up is argument parsing. Two arguments we know we need up front include a verbosity level (the default is 1, but the $-q$ option makes it 0 and the $-v$ option makes it 2), and a thread count.

```
⟨ Data declarations 2 ⟩ ≡
  int verbose = 1;
  int numthreads = 1;
  const int MAX_THREADS = 32;
```

See also sections 4, 7, 15, 19, and 28.

This code is used in section 1.

3. Parsing the arguments is boilerplate code.

```
⟨ Parse arguments 3 ⟩ ≡
  while (argc > 1 ∧ argv[1][0] ≡ '-') {
    argc --;
    argv ++;
    switch (argv[0][1]) {
    case 'v': verbose ++;
      break;
    case 'q': verbose = 0;
      break;
    case 't':
      if (sscanf(argv[1], "%d", &numthreads) ≠ 1)
        error ("!␣bad␣thread␣count␣argument") ;
      if (numthreads < 1 ∨ numthreads > MAX_THREADS)
        error ("!␣bad␣value␣for␣thread␣count") ;
      argc --;
      argv ++;
      break;
      ⟨ More arguments 5 ⟩
    default:
      error ("!␣bad␣argument") ;
    }
  }
```

This code is used in section 1.

4. The two-phase algorithm discovers solutions for each position of decreasing distance. Nominally, it runs forever for any single position, always seeking a shorter position. For this command-line program there are two ways to limit its execution. The first is to limit the count of phase two probes that it will do per position. The next is to give a target solution length, such that if a solution of that length or shorter is found, search will end. We also track how many total phase 2 probes there were.

```
⟨ Data declarations 2 ⟩ +≡
  int target_length = 0;
  long long phase2limit = #ffffffffffffLL;
  long long phase2total = 0LL;
```

5. Both of these can be set from the command line.

```
⟨ More arguments 5 ⟩ ≡
case 'M':
  if (sscanf(argv[1], "%lld", &phase2limit) ≠ 1)
    error("!␣bad␣argument␣to␣-M");
  argc−−;
  argv++;
  break;
case 's':
  if (sscanf(argv[1], "%d", &target_length) ≠ 1)
    error("!␣bad␣argument␣to␣-s");
  if (target_length ≥ MAX_MOVES) target_length = MAX_MOVES − 1;
  argc−−;
  argv++;
  break;
```

See also sections 8 and 20.

This code is used in section 3.

6. If neither of these two arguments are supplied, and we're not running in verbose mode of 2 or greater, that's an error because no output will be generated.

```
⟨ Initialize the program 6 ⟩ ≡
  if (phase2limit ≥ #ffffffffffffLL ∧ target_length ≡ 0 ∧ verbose ≤ 1)
    error("!␣must␣specify␣-M,␣-s,␣or␣-v");
```

See also sections 9 and 16.

This code is used in section 1.

7. Usually the pruning tables are read from disk; if they don't exist, they are created, and then written to disk. If for some reason you do not want to write the pruning tables to disk, you can use the $-W$ option to inhibit this.

```
⟨ Data declarations 2 ⟩ +≡
  int skipwrite = 0;
```

8. Parsing this argument is easy.

```
⟨ More arguments 5 ⟩ +≡
case 'W': skipwrite++;
  break;
```

9. If we are not running in quiet mode, we always print the banner as the first part of initialization, before we load the pruning tables since those can be slow. We use both a phase 1 and a phase 2 pruning table, so we initialize these in turn.

```
⟨ Initialize the program 6 ⟩ +≡
  if (verbose) cout ≪ BANNER ≪ endl ≪ flush;
  phase1prune :: init(skipwrite);
  phase2prune :: init(skipwrite);
```

10. We know that no solution will ever be longer than 32 moves. The maximum distance from phase 1 is 12, and from phase 2 is 18. We give it two extra just for good measure.

```
⟨ Utility functions 10 ⟩ ≡
  const int MAX_MOVES = 32;
```

See also sections 17, 22, 27, 29, and 32.

This code is used in section 1.

11. Multithreading. This program is intended to batch-solve many positions, rather than just try to solve a particular position; for this reason, we use threads and assign each position a thread (up to the maximum number of threads specified by the user). We do not solve a single position by multiple threads, because of the complexity of managing communication between the threads. Thus, the solution work and all solution state is maintained in a worker thread called a *twophasesolver*. Each instance of this class is small, so we preallocate all needed instances. We introduce a bit of padding so multiple threads don't fight over the same cache lines.

```
⟨ Solver class 11 ⟩ ≡
  class twophasesolver {
  public:
    twophasesolver( )
    { }
    cubepos pos;    /* position to solve */
    long long phase2probes;
    int bestsol;    /* length of the best solution */
    int finished;    /* set to true to terminate */
    int curm;    /* what orientation we are working on */
    int solmap;    /* the orientation the solution is in */
    int seq;    /* a serial number for this position */
    ⟨ Solver data 12 ⟩
    ⟨ Solver methods 13 ⟩
    char pad[256];
  } solvers[MAX_THREADS];
```

This code is used in section 1.

12. As we perform the searches, we place the moves we have needed into the following array.

```
⟨ Solver data 12 ⟩ ≡
  unsigned char moves[MAX_MOVES];
  unsigned char bestmoves[MAX_MOVES];
```

See also section 18.

This code is used in section 11.

13. Our basic method for solving is here. This initializes the structure and starts the solution process. The result is stored in the fields of the solver structure.

```
⟨ Solver methods 13 ⟩ ≡
  void solve(int seqarg, cubepos &cp)
  {
    ⟨ Initialize the solver 14 ⟩
    ⟨ Set up position invariants 21 ⟩
    ⟨ Solve one position 23 ⟩
    ⟨ Check and report solution 26 ⟩
  }
```

See also sections 24, 25, 31, and 33.

This code is used in section 11.

14. We initialize the fields we have declared so far.

⟨ Initialize the solver 14 ⟩ ≡

```
pos = cp;
phase2probes = 0;
bestsol = MAX_MOVES;    /* no solution found */
finished = 0;
seq = seqarg;
```

This code is used in section 13.

15. Access to the output stream, or any other shared state, requires a global mutex.

⟨ Data declarations 2 ⟩ +≡

```
pthread_mutex_t my_mutex;
```

16. We initialize the mutex.

⟨ Initialize the program 6 ⟩ +≡

```
pthread_mutex_init(&my_mutex, Λ);
```

17. We call these methods to acquire and release the mutex.

⟨ Utility functions 10 ⟩ +≡

```
void get_global_lock()
{
  pthread_mutex_lock(&my_mutex);
}
void release_global_lock()
{
  pthread_mutex_unlock(&my_mutex);
}
```

18. This program uses a new, six-axis technique, to find a solution as quickly as possible. Normally the two-phase method just repeats phase 1 and phase 2 for increasing depths of phase 1 from the given input position. Some positions, however, are recalcitrant, and do not have a nice short phase 1 solution leading to a short phase 2 solution. To make it more likely to find a solution quickly, we consider all three axis reorientations. Further, we also consider the inverse position with all three axis reorientations, for a grand total of six different starting positions. For positions with symmetry or antisymmetry, we do not want to cover the same ground multiple times, so we need to calculate which of these six combinations are distinct.

For each of the six starting positions, we construct a **kocsymm** and a **cubepos** to carry the state, calculate the minimum phase one depth, and check which are unique. We need to declare these fields.

⟨ Solver data 12 ⟩ +≡

```
kocsymm kc6[6], kccanon6[6];
cubepos cp6[6];
permcube pc6[6];
int mindepth[6];
char uniq[6];
int minmindepth;
```

19. Sometimes we want to run the twophase algorithm only working with a specific subset of the possible axes. This mask gives that information.

⟨ Data declarations 2 ⟩ +≡

```
int axesmask = 63;
```

20. We use the -a option to set this.

```
⟨ More arguments 5 ⟩ +≡
case 'a': axesmask = atol(argv[1]);
  argv++;
  argc--;
  break;
```

21. Initializing these is fairly easy. We need to know what reorientations change the axis; from **cubepos** we know that each group of sixteen consecutive remappings maintains the up/down axis, so we use reorientations 0, 16, and 32. When comparing positions for equality, we want to use the 16-way canonicalization that preserves the up/down axis; the **kocsymm** canonicalization preserves this, so we use this to preface a more involved comparison.

```
⟨ Set up position invariants 21 ⟩ ≡
  minmindepth = MAX_MOVES;
  cubepos cpi, cp2;
  pos.invert_into(cpi);
  int ind = 0;
  for (int inv = 0; inv < 2; inv++)
    for (int mm = 0; mm < 3; mm++, ind++) {
      int m = KOCSYMM * mm;
      if (inv) cpi.remap_into(m, cp2);
      else pos.remap_into(m, cp2);
      cp6[ind] = cp2;
      kc6[ind] = kocsymm(cp2);
      pc6[ind] = permcube(cp2);
      kc6[ind].canon_into(kccanon6[ind]);
      mindepth[ind] = phase1prune::lookup(kc6[ind]);
      if (mindepth[ind] < minmindepth) minmindepth = mindepth[ind];
      uniq[ind] = 1 & (axesmask ≫ ind);
      for (int i = 0; i < ind; i++)
        if (uniq[i] ∧ kccanon6[ind] ≡ kccanon6[i] ∧ sloweq(cp6[ind], cp6[i])) {
          uniq[ind] = 0;
          break;
        }
      if (verbose > 1) {
        get_global_lock();
        cout ≪ "Axis␣" ≪ ind ≪ "␣depth␣" ≪ mindepth[ind] ≪ "␣uniq␣" ≪ (int) uniq[ind] ≪ endl;
        release_global_lock();
      }
    }
```

This code is used in section 13.

22. We need a utility method that does a slow comparison between two **cubepos** and returns true if there's any reorientation, preserving the up/down axis, of one that is the same as the other.

```
⟨ Utility functions 10 ⟩ +≡
  int sloweq(const cubepos &cp1, const cubepos &cp2)
  {
    cubepos cp3;
    for (int m = 0; m < KOCSYMM; m++) {
      cp2.remap_into(m, cp3);
      if (cp1 ≡ cp3) return 1;
    }
    return 0;
  }
```

23. Once we have a minimum depth, we can start solving.

```
⟨ Solve one position 23 ⟩ ≡
  for (int d = minmindepth; d < bestsol ∧ ¬finished; d++) {
    for (curm = 0; curm < 6; curm++)
      if (uniq[curm] ∧ d < bestsol ∧ ¬finished ∧ d ≥ mindepth[curm]) {
        if (verbose > 1) {
          get_global_lock();
          cout ≪ "Orientation␣" ≪ curm ≪ "␣at␣depth␣" ≪ d ≪ endl;
          release_global_lock();
        }
        solvep1(kc6[curm], pc6[curm], d, 0, ALLMOVEMASK, CANONSEQSTART);
      }
  }
```

This code is used in section 13.

24. The phase one solver.

⟨ Solver methods 13 ⟩ +≡

```
void solvep1(const kocsymm &kc, const permcube &pc, int togo, int sofar, int movemask, int
        canon)
{
  if (togo ≡ 0) {
    if (kc ≡ identity_kc) solvep2(pc, sofar);
    return;
  }
  if (finished) return;
  togo−−;
  kocsymm kc2;
  permcube pc2;
  int newmovemask;
  while (¬finished ∧ movemask) {
    int mv = ffs(movemask) − 1;
    movemask &= movemask − 1;
    kc2 = kc;
    kc2.move(mv);
    int nd = phase1prune::lookup(kc2, togo, newmovemask);
    if (nd ≤ togo ∧ (togo ≡ nd ∨ togo + nd ≥ 5)) {
      pc2 = pc;
      pc2.move(mv);
      moves[sofar] = mv;
      int new_canon = cubepos::next_cs(canon, mv);
      solvep1(kc2, pc2, togo, sofar + 1, newmovemask & cubepos::cs_mask(new_canon), new_canon);
    }
  }
}
```

25. The phase two code just uses the phase two solver.

```
⟨ Solver methods 13 ⟩ +≡
  void solvep2(const permcube &pc, int sofar)
  {
    phase2probes++;
    int d = phase2prune::lookup(pc);
    if (d + sofar < bestsol) {
      moveseq ms = phase2prune::solve(pc, bestsol − sofar − 1);
      if ((int)(ms.size()) + sofar < bestsol ∧ (ms.size() > 0 ∨ pc ≡ identity_pc)) {
        bestsol = ms.size() + sofar;
        for (unsigned int i = 0; i < ms.size(); i++) moves[sofar + i] = ms[i];
        memcpy(bestmoves, moves, bestsol);
        if (verbose > 1) {
          get_global_lock();
          cout ≪ "New␣solution␣for␣" ≪ seq ≪ "␣at␣" ≪ bestsol ≪ endl;
          release_global_lock();
        }
        solmap = curm;
        if (bestsol ≤ target_length) finished = 1;
      }
    }
    if (phase2probes ≥ phase2limit ∧ bestsol < MAX_MOVES) finished = 1;
  }
```

26. When we have a solution, and we have decided to present it to the user, we need to first remap it, and then check it.

```
⟨ Check and report solution 26 ⟩ ≡
  moveseq sol;
  int m = cubepos::invm[(solmap % 3) * KOCSYMM];
  for (int i = 0; i < bestsol; i++) sol.push_back(cubepos::move_map[m][bestmoves[i]]);
  if (solmap ≥ 3) sol = cubepos::invert_sequence(sol);
  cubepos cpt;
  for (unsigned int i = 0; i < sol.size(); i++) cpt.move(sol[i]);
  if (cpt ≠ pos)
    error ("!␣move␣sequence␣doesn't␣work") ;
  report(pos, seq, phase2probes, sol);
```

This code is used in section 13.

27. When we display a solution, we give some basic statistics as well as the solution itself. If we missed a target, we display the orignal position in Singmaster format as well.

```
⟨ Utility functions 10 ⟩ +≡
  void display(const cubepos &cp, int seq, long long phase2probes, moveseq sol)
  {
    phase2total += phase2probes;
    if (verbose ∨ (int) sol.size() > target_length) {
      if ((int) sol.size() > target_length)
        cout ≪ "WARNING:␣missed␣target␣for␣" ≪ cp.Singmaster_string() ≪ endl;
      cout ≪ "Solution␣" ≪ seq ≪ "␣len␣" ≪ sol.size() ≪ "␣probes␣" ≪ phase2probes ≪ endl;
      cout ≪ cubepos::moveseq_string(sol) ≪ endl;
    }
  }
```

28. Reporting solutions. We would like to report solutions in sequential order. Yet, with threads, they may have their solutions found in non-sequential order. To manage this, we keep track of the most recently printed sequence, and only print a solution if it is the most recent sequence. If it is not we queue it up and print it when the missing sequence shows. We define a structure to hold the relevant information. We also define a couple of variables to contain statistics.

```
⟨ Data declarations 2 ⟩ +≡
  class solution {
  public:
    solution(const cubepos &cparg, int seqarg, long long p2parg, moveseq &solarg)
    {
      cp = cparg;
      seq = seqarg;
      phase2probes = p2parg;
      moves = solarg;
    }
    solution()
    { }
    cubepos cp;
    int seq;
    long long phase2probes;
    moveseq moves;
  };
  map⟨int, solution⟩ queue;
  int next_sequence = 1;
  int missed_target = 0;
  int solved = 0;
```

29. Our reporting function does the main work, with the global lock.

```
⟨ Utility functions 10 ⟩ +≡
  void report(const cubepos &cp, int seq, long long phase2probes, moveseq sol)
  {
    get_global_lock();
    solved++;
    if ((int) sol.size() > target_length ∧ target_length) missed_target++;
    if (seq ≡ next_sequence) {
      display(cp, seq, phase2probes, sol);
      next_sequence++;
      while (queue.find(next_sequence) ≠ queue.end()) {
        solution &s = queue[next_sequence];
        display(s.cp, s.seq, s.phase2probes, s.moves);
        queue.erase(next_sequence);
        next_sequence++;
      }
    }
    else {
      queue[seq] = solution(cp, seq, phase2probes, sol);
    }
    release_global_lock();
  }
```

30. When we are all done, we print this summary.

```
⟨ Print summary 30 ⟩ ≡
  if (missed_target)
    cout ≪ "WARNING:␣␣missed␣target␣on␣" ≪ missed_target ≪ "␣sequences." ≪ endl;
  phase1prune :: check_integrity( );
  phase2prune :: check_integrity( );
  cout ≪ "Solved␣" ≪ solved ≪ "␣sequences␣in␣" ≪ duration( ) ≪ "␣seconds␣with␣" ≪
      phase2total ≪ "␣probes." ≪ endl;
  cout ≪ "Completed␣in␣" ≪ (walltime( ) − progstart) ≪ endl;
  exit(0);
```

This code is used in section 1.

31. Each thread has a routine that gets the next sequence to solve, reports its solution, and moves on to the next, until there are no more positions to solve. This is that routine.

```
⟨ Solver methods 13 ⟩ +≡
  void dowork( )
  {
    cubepos cp;
    int seq;
    while (1) {
      seq = getwork(cp);
      if (seq ≤ 0) return;
      solve(seq, cp);
    }
  }
```

32. The getwork routine grabs the global lock, gets another input line, parses it, and returns the position and the sequence.

```
⟨ Utility functions 10 ⟩ +≡
  int getwork(cubepos &cp)
  {
    static int input_seq = 1;
    const int BUFSIZE = 1000;
    char buf[BUFSIZE + 1];
    get_global_lock();
    if (fgets(buf, BUFSIZE, stdin) ≡ 0) {
      release_global_lock();
      return −1;
    }
    if (cp.parse_Singmaster(buf) ≠ 0) {
      cp = identity_cube;
      const char *p = buf;
      moveseq ms = cubepos::parse_moveseq(p);
      if (*p)
        error ("!␣could␣not␣parse␣position") ;
      for (unsigned int i = 0; i < ms.size(); i++) cp.move(ms[i]);
    }
    int r = input_seq++;
    release_global_lock();
    return r;
  }
```

33. Pthreads wants a function, not a method, so we define this routine to help it out.

```
⟨ Solver methods 13 ⟩ +≡
  static void *worker(void *s)
  {
    twophasesolver *solv = (twophasesolver *) s;
    solv→dowork();
    return 0;
  }
```

34. Our main body spawns the number of threads requested by the user and waits for them to finish. As a minor optimization, we use the main thread for the thread zero work.

```
⟨ Handle the work 34 ⟩ ≡
  pthread_t p_thread[MAX_THREADS];
  for (int ti = 1; ti < numthreads; ti++)
    pthread_create(&(p_thread[ti]), Λ, twophasesolver::worker, solvers + ti);
  solvers[0].dowork();
  for (int ti = 1; ti < numthreads; ti++) pthread_join(p_thread[ti], 0);
```

This code is used in section 1.

TWOPHASE