Appendix: Sparse Implementation

This appendix details the implementation of `libq`, including some optimization successes and failures. The full source code can be found online in directory `src/libq` in the open-source repository. It is about 500 lines of C++ code. Correspondingly, this section is very code heavy.

A.1 Register File

The register file, which holds the qubits, is defined in `libq.h` in the type `qureg_t`. Again, we use similar names in `libq` as found in libquantum to enable line-by-line comparisons. This structure will hold an array with complex amplitudes and an array with the state bitmasks.

```
typedef uint64 state_t;

struct qureg_t {
 cmplx* amplitude;
 state_t* state;
 }
```

Interesting tidbit: in an earlier version of this library that was included in the SPEC 2006 benchmarks, those two arrays were written as an array of structures, which is not good for performance, as iterations over the array to, say, flip a bit in all states has to iterate over more memory than necessary. This author implemented a quite involved data layout transformation in the HP compilers for Itanium to transform the array of `structs` into a `struct` of arrays (Hundt et al., 2006). A later version of the library then modified the source code itself, erasing the need and benefit of the complex compiler transformation.

The member `width`, which probably deserves a better name, holds the number of qubits in this register. The member `size` holds the number of nonzero probabilities, and `hash` is the actual hash table, with `hashw` being the allocation size of the hash table.

https://doi.org/10.1017/9781009099974.011 Published online by Cambridge University Press

```
 int width; /* number of qubits in the qureg */
 int size;  /* number of non-zero vectors */

 int hashw; /* width of the hash array */
 int* hash; /* hash table */
```

The operations to check whether a bit is set and to XOR a specific bit with a value are common, so we extract them into member functions:

```
 bool bit_is_set(int index, int target) __attribute__ ((pure)) {
   return state[index] & (static_cast<state_t>(1) << target);
 }
 void bit_xor(int index, int target) {
   state[index] ^= (static_cast<state_t>(1) << target);
 }

typedef struct qureg_t qureg;
```

The following operations are allowed for this quantum register.

Create a new quantum register of a given size `width` and initialize an initial single state with bitmask `initval` with probability 1.0 (one state must be defined). The function's main job is to `calloc()` the various arrays and make sure there are no out-of-memory errors.

```
qureg *new_qureg(state_t initval, int width);

.cc:
   libq::qureg* q = libq::new_qureg(0, 2);
```

Free all allocated data structures, set relevant pointers to `nullptr`.

```
void delete_qureg(qureg *reg);
```

Print a textual representation of the current state by listing all states with non-zero probability.

```
void print_qureg(qureg *reg);
```

Display statistics, such as how many qubits were stored, how often the hash table was recomputed, and, another important metric, the maximum number of non-zero probability states reached during the execution of an algorithm:

```
void print_qureg_stats(qureg *reg);
```

https://doi.org/10.1017/9781009099974.011 Published online by Cambridge University Press

For certain experiments, we cache internal state. This next function ensures that all remaining states will be flushed. This could mean a computation is completed or some pending prints are flushed to `stdout`.

```
void flush(qureg* reg);
```

A.2 Superposition-Preserving Gates

These are gates that do not create or destroy superposition. They represent the "easy" case in this sparse representation. Let us look at a few representative gates; the full implementation is in `libq/gates.cc`.

To apply the X-gate to a specific qubit, the bit corresponding to the qubit index must be flipped. Recall that the gate's function is determined by:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}.$$

If there are n states with nonzero probabilities in the system, we will have n tuples. To flip one qubit's probability according to the gate, we have to flip the bit for that qubit in each of those tuples, as that represents the operation of this gate on all the states. The probability amplitudes for that qubit are flipped by just flipping the bit; there is no other data movement. The code is remarkably simple:

```
void x(int target, qureg *reg) {
    for (int i = 0; i < reg->size; ++i) {
        reg->bit_xor(i, target);
    }
}
```

For another class of operators, we must check whether a bit is set before applying a transformation. For example, applying the Z-gate to a state acts like this:

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ -\beta \end{bmatrix}.$$

The gate only has effect if β is nonzero. In the sparse representation, this means that there must be a tuple representing a nonzero probability that has a 1 at the intended qubit location. We iterate over all state tuples, check for the condition, and only negate the amplitude if that bit was set:

```
void z(int target, qureg *reg) {
  for (int i = 0; i < reg->size; ++i) {
    if (reg->bit_is_set(i, target)) {
      reg->amplitude[i] *= -1;
    }
  }
}
```

https://doi.org/10.1017/9781009099974.011 Published online by Cambridge University Press

Recall that if the qubit is in superposition, there will be two tuples: one with the corresponding bit set to 0 and the amplitude set to α, and the other with the bit set to 1 and the amplitude set to β. For the Z-gate, we only need to change the second tuple. A related example is the T-gate, there are a few more gates of this nature:

```
void t(int target, qureg *reg) {
    static cmplx z = cexp(M_PI / 4.0);
    for (int i = 0; i < reg->size; ++i) {
        if (reg->bit_is_set(i, target)) {
            reg->amplitude[i] *= z;
        }
    }
}
```

The Y-gate is moderately more complex, using a combination of the methods shown above.

$$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} -i\beta \\ i\alpha \end{bmatrix}.$$

First, we flip the bit, similar to the X-gate, and then we apply i or $-i$, depending on whether or not the qubit's bit is set after being flipped:

```
void y(int target, qureg *reg) {
    for (int i = 0; i < reg->size; ++i) {
        reg->bit_xor(i, target);
        if (reg->bit_is_set(i, target))
            reg->amplitude[i] *= cmplx(0, 1.0);
        else
            reg->amplitude[i] *= cmplx(0, -1.0);
    }
}
```

A.3 Controlled Gates

Controlled gates are a logical extension of the above. In order to make a gate controlled, we only have to check whether the corresponding control bit is set to 1. For example, for the Controlled-X gate:

```
void cx(int control, int target, qureg *reg) {
    for (int i = 0; i < reg->size; ++i) {
        if (reg->bit_is_set(i, control)) {
            reg->bit_xor(i, target);
        }
    }
}
```

https://doi.org/10.1017/9781009099974.011 Published online by Cambridge University Press

For the Controlled-Z gate:

```
void cz(int control, int target, qureg *reg) {
    for (int i = 0; i < reg->size; ++i) {
        if (reg->bit_is_set(i, control)) {
            if (reg->bit_is_set(i, target)) {
                reg->amplitude[i] *= -1;
            }
        }
    }
}
```

This even works for double-controlled gates, where we only have to check for both control bits to be set. Here is the implementation of a double-controlled X-gate:

```
void ccx(int control0, int control1, int target, qureg *reg) {
    for (int i = 0; i < reg->size; ++i) {
        if (reg->bit_is_set(i, control0)) {
            if (reg->bit_is_set(i, control1)) {
                reg->bit_xor(i, target);
            }
        }
    }
}
```

A.4 Superpositioning Gates

The difficult case is for gates that create or destroy superposition. We provide a single implementation in `apply.cc` for this in function `libq_gate1`. The function expects the 2×2 gate to be passed in.

For example, for the Hadamard gate:

```
void h(int target, qureg *reg) {
    static cmplx mh[4] = {sqrt(1.0/2), sqrt(1.0/2), sqrt(1.0/2),
                          -sqrt(1.0/2)};
    libq_gate1(target, mh, reg);
}
```

The implementation itself applies the same technique we saw earlier in Section 4.5 on accelerated gates: a linear traversal over the states, except it is adapted to the sparse representation. Additionally, it manages memory by filtering out close-to-zero states. Let's dive into it. The implementation is about 175 lines of code.

https://doi.org/10.1017/9781009099974.011 Published online by Cambridge University Press

A.5 Hash Table

First, as indicated above, states are maintained in a hash table with the following hash function:

```
static inline unsigned int hash64(state_t key, int width) {
    unsigned int k32 = (key & 0xFFFFFFFF) ^ (key >> 32);
    k32 *= 0x9e370001UL;
    k32 = k32 >> (32 - width);
    return k32;
}
```

The hash lookup function `get_state` checks whether a given state exists with nonzero probability. It computes the hash index for a state a and iterates over the dense array, hoping to find the actual state. If a 0 state was found (the marker for a unpopulated entry) or if the search wraps around, no state was found and -1 is returned; otherwise, the position in the hash table is returned:

```
state_t get_state(state_t a, qureg *reg) {
  unsigned int i = hash64(a, reg->hashw);
  while (reg->hash[i]) {
    if (reg->state[reg->hash[i] - 1] == a) {
      return reg->hash[i] - 1;
    }
    i++;
    if (i == (1 << reg->hashw)) {
      break;
    }
  }
  return -1;
}
```

There is, of course, a function to add a state to the hash table:

```
void libq_add_hash(state_t a, int pos, qureg *reg) {
  int mark = 0;

  int i = hash64(a, reg->hashw);
  while (reg->hash[i]) {
    i++;
    if (i == (1 << reg->hashw)) {
      if (!mark) {
        i = 0;
        mark = 1;
      }
```

https://doi.org/10.1017/9781009099974.011 Published online by Cambridge University Press

```
    }
  }
  reg->hash[i] = pos + 1;
  // -- Optimization will happen here (later).
}
```

The most interesting function from a performance perspective is the one to reconstruct the hash table. Since the gate apply function will filter out states with probabilities close to 0.0, after gate application, we have to reconstruct the hash table to ensure it only contains valid entries. This is the most expensive operation of the whole `libq` implementation. We show some optimizations below, where the first loop is being replaced with a `memset()`, and also in Section A.8.

```
void libq_reconstruct_hash(qureg *reg) {
    reg->hash_computes += 1;   // count invocations.

    for (int i = 0; i < (1 << reg->hashw); ++i) {
        reg->hash[i] = 0;
    }
    for (int i = 0; i < reg->size; ++i) {
        libq_add_hash(reg->state[i], i, reg);
    }
}
```

The first thing to note is the first loop, which resets the hash array to all zeros:

```
    for (int i = 0; i < (1 << reg->hashw); ++i) {
        reg->hash[i] = 0;
    }
```

You might expect that the compiler transforms this loop to a vectorized `memset` operation. However, it does not. The loop-bound `reg->hashw` aliases with the loop body, meaning that the compiler cannot infer whether or not the loop body would modify the loop bound. Manually changing this to a `memset` speeds up the whole simulation by about 20%.

```
    memset(reg->hash, 0, (1 << reg->hashw) * sizeof(int));
```

This `memset` is still the slowest part of the implementation. Later, we will show how to optimize it further.

A.6 Gate Application

Here is the routine to apply a gate. It assumes that something changed since last invocation, so the first task is to reconstruct the hash table:

https://doi.org/10.1017/9781009099974.011 Published online by Cambridge University Press

```
void libq_gate1(int target, cmplx m[4], qureg *reg) {
    int addsize = 0;
    libq_reconstruct_hash(reg);
    [...]
}
```

Superposition on a given qubit means that both the states with a 0 and a 1 at a given bit position must exist. So the function iterates, and counts how many of those states are missing and need to be added:

```
/* calculate the number of basis states to be added */
for (int i = 0; i < reg->size; ++i) {
   /* determine whether XORed basis state already exists */
   if (get_state(reg->state[i] ^
        (static_cast<state_t>(1) << target), reg) == -1)
         addsize++;
 }
```

If new states need to be added, the function reallocates the arrays. It also does some bookkeeping and remembers the largest number of states with nonzero probability:

```
 /* allocate memory for the new basis states */
 if (addsize) {
   reg->state = static_cast<state_t *>(
     realloc(reg->state, (reg->size + addsize) * sizeof(state_t)));
   reg->amplitude = static_cast<cmplx *>(
     realloc(reg->amplitude, (reg->size + addsize) * sizeof(cmplx)));

   memset(&reg->state[reg->size], 0, addsize * sizeof(int));
   memset(&reg->amplitude[reg->size], 0, addsize * sizeof(cmplx));
   if (reg->size + addsize > reg->maxsize) {
     reg->maxsize = reg->size + addsize;
   }
 }
```

This is all for state and memory management. Now on to applying the gates. We allocate an array `done` to remember which states we've already handled. The `limit` variable will be used at the end of the function to remove states with close to zero probability.

```
    char *done =
       static_cast<char *>(calloc(reg->size + addsize, sizeof(char)));
    int next_state = reg->size;
    float limit = (1.0 / (static_cast<state_t>(1) << reg->width))
                  * 1e-6;
```

https://doi.org/10.1017/9781009099974.011 Published online by Cambridge University Press

We then we iterate over all states and check if a state has not been handled. We check whether a target bit has been set and obtain the other base state's index in variable `xor_index`. The amplitudes for the $|0\rangle$ and $|1\rangle$ basis states are stored in `tnot` and `t`.

```
/* perform the actual matrix multiplication */
for (int i = 0; i < reg->size; ++i) {
   if (!done[i]) {
        /* determine if the target of the basis state is set */
        int is_set = reg->state[i] & (static_cast<state_t>(1) <<
        ↪  target);
        int xor_index =
             get_state(reg->state[i] ^
             (static_cast<state_t>(1) << target), reg);
        cmplx tnot = xor_index >= 0 ? reg->amplitude[xor_index] : 0;
        cmplx t = reg->amplitude[i];
    }
    [...]
}
```

The matrix multiplication follows the patterns we've seen before for the fast gate application in Section 4.5. If states are found, we apply the gate. If the XOR'ed state was not found, this means we have to add a new state and perform the multiplication:

```
    if (is_set) {
      reg->amplitude[i] = m[2] * tnot + m[3] * t;
    } else {
      reg->amplitude[i] = m[0] * t + m[1] * tnot;
    }

    if (xor_index >= 0) {
      if (is_set) {
        reg->amplitude[xor_index] = m[0] * tnot + m[1] * t;
      } else {
        reg->amplitude[xor_index] = m[2] * t + m[3] * tnot;
      }
    } else { /* new basis state will be created */
      if (abs(m[1]) == 0.0 && is_set) break;
      if (abs(m[2]) == 0.0 && !is_set) break;

      reg->state[next_state] =
          reg->state[i] ^ (static_cast<state_t>(1) << target);
      reg->amplitude[next_state] = is_set ? m[1] * t : m[2] * t;
      next_state += 1;
    }
```

https://doi.org/10.1017/9781009099974.011 Published online by Cambridge University Press

```
    if (xor_index >= 0) {
      done[xor_index] = 1;
    }
```

As a final step, we filter out the states with a probability close to 0. This code densifies the array by moving up all nonzero elements before finally reallocating the amplitude and state arrays to a smaller size (which is actually a redundant operation):

```
reg->size += addsize;
free(done);

/* remove basis states with extremely small amplitude */
if (reg->hashw) {
  int decsize = 0;
  for (int i = 0, j = 0; i < reg->size; ++i) {
    if (probability(reg->amplitude[i]) < limit) {
      j++;
      decsize++;
    } else if (j) {
      reg->state[i - j] = reg->state[i];
      reg->amplitude[i - j] = reg->amplitude[i];
    }
  }

  if (decsize) {
    reg->size -= decsize;

    # Note that these 2 realloc's are redundant and not needed.
    // reg->amplitude = static_cast<cmplx *>(
    //   realloc(reg->amplitude, reg->size * sizeof(cmplx)));
    // reg->state = static_cast<state_t *>(
    //   realloc(reg->state, reg->size * sizeof(state_t)));
  }
}
```

A.7 Premature Optimization, Second Act

Here is an anecdote that might serve as a lesson. After implementing the code and running initial benchmarks, it appeared obvious that the repeated iteration over the memory just had to be a bottleneck. Some form of mini-JIT (Just-In-Time compilation) should be helpful, which first collects all the operations and then fuses gate applications into the same loop iteration. The goal would be to significantly reduce repeated iterations over the states to avoid the memory traffic which, again, just had to

https://doi.org/10.1017/9781009099974.011 Published online by Cambridge University Press

be the problem. The code is available online. It might become valuable in the future, as other performance bottlenecks are being resolved.

The goal of the main routine was to execute something like the following, with just one outer loop and a switch statement over all superposition-preserving gates:

```
[...]
void Execute(qureg *reg) {
   for (int i = 0; i < reg->size; ++i) {
     for (auto op : op_list_) {
       switch (op.op()) {
         case op_t::X:
           reg->bit_xor(i, op.target());
           break;

         case op_t::Y:
           reg->bit_xor(i, op.target());
           if (reg->bit_is_set(i, op.target()))
             reg->amplitude[i] *= cmplx(0, 1.0);
           else
             reg->amplitude[i] *= cmplx(0, -1.0);
           break;

         case op_t::Z:
           if (reg->bit_is_set(i, op.target())) {
             reg->amplitude[i] *= -1;
           }
           Break;
[...] }}}}
```

As a complete surprise, running the JIT'ed version produced a performance improvement of 0%. Simple profiling then revealed that about 96% of the execution time was spent in reconstructing the hash table. Gate application wasn't a performance bottleneck at all. Lesson learned – intuition is good; validation is better. Didn't we mention this before?

A.8 Actual Performance Optimization

As noted above, reconstructing the hash table is the most expensive operation in this library. The hash table is sized to hold all potential states, given the number of qubits. However, even for complex algorithms, the actual maximal number of states with nonzero probability can be quite small. For example, for two benchmarks we extract from quantum arithmetic (`Arith`) and order finding (`Order`), we show the maximum number of nonzero states reached (8,192) and, given the number of qubits involved, the theoretical maximal number of states. The percentage is 3.125% for `Order`, and only 0.012% for `Arith`. It has a lot more qubits and, hence, a very large potential number of states.

https://doi.org/10.1017/9781009099974.011 Published online by Cambridge University Press

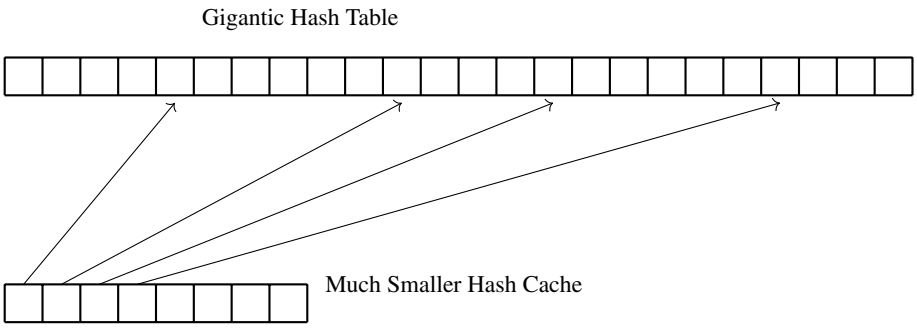

Figure A.1 A caching scheme to accelerate hash table zeroing.

```
Arith:  Maximum of states: 8192, theoretical: 67108864, 0.012%
Order:  Maximum of states: 8192, theoretical: 262144, 3.125%
```

During execution, the number of states changes dynamically in powers of two as `libq` removes states that are very close to 0.0. Therefore, there is an opportunity to augment the hash table and track, or cache, the addresses of elements that have been set, up to a given threshold, for example, up to 64K elements.

To reset the hash table, we iterate over this *hash cache* and zero out the populated elements in the hash table, as shown in Figure A.1. There will be a crossover point. For some size of the hash cache, just linearly sweeping the hash table will be faster than the random memory access patterns from the cache because of hardware prefetching dynamics. We picked 64K as cache size, which, for our given examples, improves the runtime significantly. This is an interesting space to experiment in, trying to find better heuristics and data structures.

In function `libq_reconstruct_hash`, we additionally maintain an array called `hash_hits` which holds the addresses of states in the main hash table, along with a counter `reg->hits` of those. Then, we selectively zero out only those memory addresses in the hash table that we cached. If the hash cache was not big enough, we have to resort to zeroing out the full hash table:

```
void libq_reconstruct_hash(qureg *reg) {
 reg->hash_computes += 1;

 if (reg->hash_caching && reg->hits < HASH_CACHE_SIZE) {
   for (int i = 0; i < reg->hits; ++i) {
     reg->hash[reg->hash_hits[i]] = 0;
     reg->hash_hits[i] = 0;
   }
   reg->hits = 0;
 } else {
   memset(reg->hash, 0, (1 << reg->hashw) * sizeof(int));
   memset(reg->hash_hits, 0, reg->hits * sizeof(int));
```

https://doi.org/10.1017/9781009099974.011 Published online by Cambridge University Press

```
    reg->hits = 0;
  }
  for (int i = 0; i < reg->size; ++i) {
    libq_add_hash(reg->state[i], i, reg);
  }
}
```

All that's left to do is to fill in this array `hash_hits` whenever we add a new element in `libq_add_hash` using the following code at the very bottom:

```
[...]
reg->hash[i] = pos + 1;
 if (reg->hash_caching && reg->hits < HASH_CACHE_SIZE) {
   reg->hash_hits[reg->hits] = i;
   reg->hits += 1;
 }
```

The performance gains from this optimization can be substantial, depending on the characteristics of an algorithm. Anecdotal evidence points to improvements in the range of 20–30% for `Arith` and `Order`, as long as the nonzero states fit in the hash cache.

https://doi.org/10.1017/9781009099974.011 Published online by Cambridge University Press