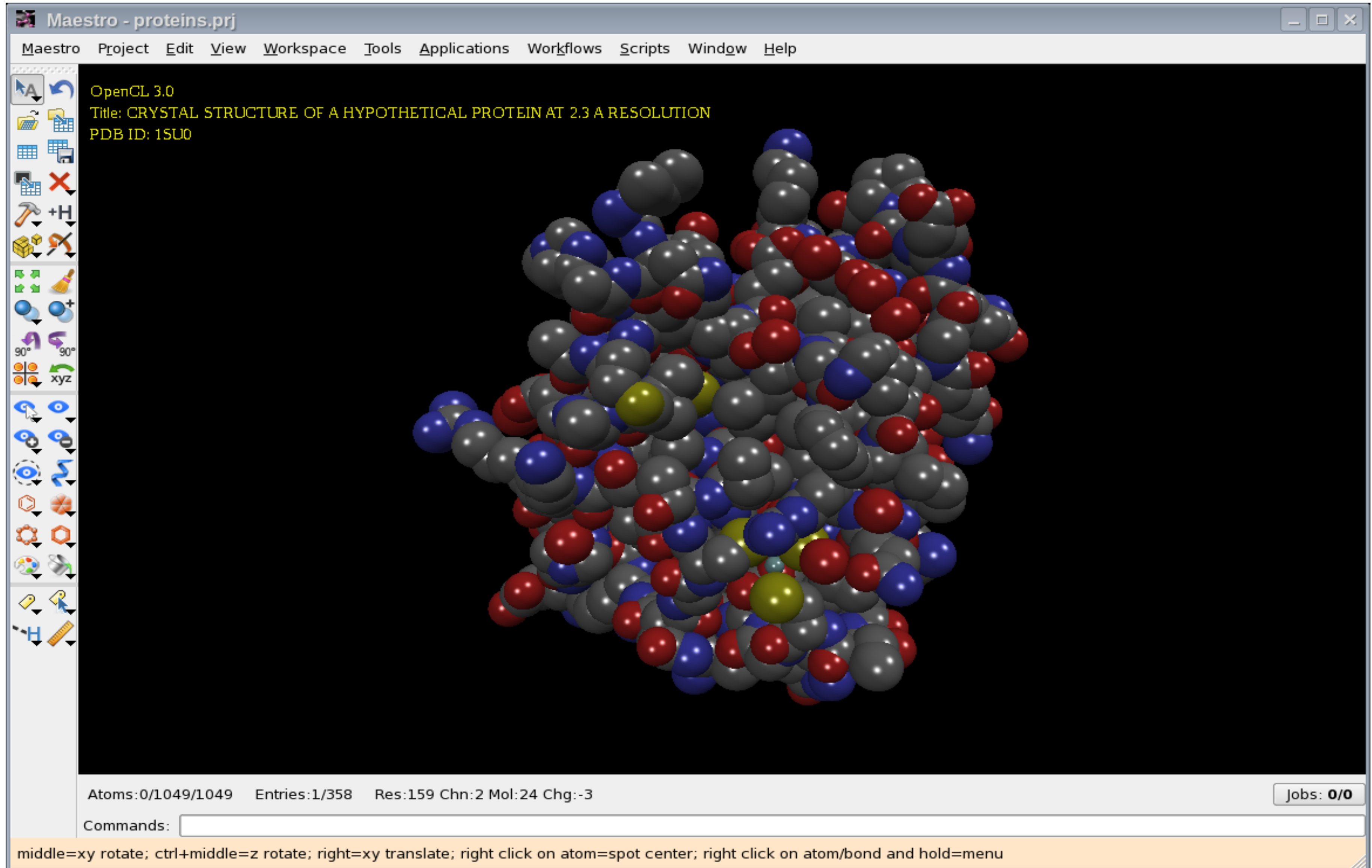
Optimizing C++ Applications

Scott MacHaffie
Schrödinger, Inc.

http://www.schrodinger.com

Maestro Graphics Example

This is a screen shot of our application.



Measure

If you don't know what's slow, how can you speed it up?

- Create targets for optimization (e.g. increase speed of function X by 20%).
- Make sure you're working on an important piece, something which truly needs to be fast (from the user's point of view).
- Implement easy-to-use measurements which you can track on a long-term basis.
- Store timing from different versions of the software, running on representative data.

Make High-Level Changes

High-level changes can give you orders of magnitude improvement.

- Algorithms
- APIs

The goal is to be able to wrap your head around the code easily. If you can keep the code in your brain, it's much easier to come up with ways to speed it up.

Improve The Algorithm

- A different algorithm beats line-level changes.
- Going from $O(n^2)$ to $O(n)$ is huge.
- Find a better algorithm (e.g. bubble sort -> quick sort).
- Do a Google search--someone may have solved the problem already.

Create An API

If you need to optimize scattered code, consider repackaging it as a library or an API.

For example, we converted our graphics code from code scattered across multiple independent classes to a graphics library with a well-defined API.

This allows for better encapsulation of the code and makes the process of reasoning about it and optimizing it simpler--you can get a handle on the code inside the API.

Data Layout

How your data is arranged can make a huge difference.

- Modern architectures are optimized to repeatedly run a small bit of code (that fits in the instruction cache) against a block of sequential memory.
- This isn't how many C++ progams are written.

```
for (i = 0; i < objects.size(); i++)
    // recalc is virtual
    objects[i]->recalc();
```

Here, the data for each object isn't sequential (the pointers to the objects are), and the functions are virtual, meaning the CPU may have to load a different set of instructions for each object.

Better Data Layout

To fix this, inline the data needed for the calculation and the result.

```
struct _recalc {
    double result;
    double input[3];
    object *obj;
};

for (i = 0; i < new_objs.size(); i++)
  new_objs[i].result =
    new_objs[i].input[0] *
    new_objs[i].input[1] *
    new_objs[i].input[2];
```

Memory

If your application allocates and deallocates objects of many different sizes, then you can't really do better than a general purpose allocator.

However, if your memory allocation / deallocation is dominated by a small number (e.g. 5-10) of classes, then there are a couple of techniques to get better performance. Assuming, of course, that you've measured and the allocations are taking a significant amount of time.

- Use a free list.
- Couple a free list with a region ("reap").

Free List

```
class object {
    public:
        double data;
        void *operator new(size_t);
        void operator delete(void *);
};

struct free_node { free_node *next; }
static free_node *free_list = NULL;

// Assumes object is never subclassed
void *object::operator new(size_t size) {
    if (free_list) {
        free_node *tmp = free_list;
        free_list = free_list->next;
        return tmp;
    }

    void *storage = malloc(size);
    if (!storage)
        throw std::bad_alloc();
    return storage;
}

void object::operator delete(void *ptr) {
    // Assumes sizeof(free_node) <= sizeof(object)
    free_node *tmp = reinterpret_cast<free_node *>(ptr);
    tmp->next = free_list;
    free_list = tmp;
}
```

Free List (cont.)

This code is designed as a basic representation of the idea--there are lots of variations. A real implementation might farm this off to templated functions. Also, common free lists could be used for objects of the same size.

Memory - Regions

Regions provide fast allocation by doing large block allocations and doing trivial allocations by moving a pointer forward.

```
static char *region = NULL;
static char *ptr = NULL;

void *object::operator new(size_t size)
{
    if (!region) {
        // Allocate enough space for 1000 elements
        region = static_cast<char *>(malloc(size * 1000));
        ptr = region;
    }

    void *storage = ptr;
    ptr += size;
    return storage;
}
```

Obviously, this code is not a complete solution, because it doesn't handle the problem of allocating past the end of the region--this is illustrative only.

Memory - Reaps

From Andrei Alexandrescu[1], some key insights on memory allocation:

- Regions provide fast allocation.
- Regions can't do deallocation--you can deallocate the entire region, but not pieces within the region.
- So use free lists to handle the deallocations within a region.

This provides fast allocation and deallocation for a specific size of object. The region block size should be consistent with the number of allocations you're doing (e.g. if you allocate 100K - 1M objects over time, then a block size in the 1K-10K range might be appropriate).

[1] http://erdani.com/

Watch For Hidden Copying

If you're trying to speed up code, then you need to watch for hidden, expensive copying.

```
void foo(vector<int> data) {
}
void bar(vector<int> data) {
    foo(data);
}
```

There are two unnecessary copies in this code: one is whoever calls bar(), and the other is the call from bar() -> foo(). Better to replace these declarations with `const vector<int> &`

Step Through Code

Another way to optimize code is to step through it in a debugger, watching for anything unexpected. Stepping through the code live under a debugger can give you a better feel for what the code is actually doing.

- Does the code take unexpected paths?
- Are functions being called more times than they should be?
- Is the code optimized for the end case rather than the common case?
- Is any unnecessary work being done? (e.g. is there anything inside a loop which can be moved out?)

Organizing Data

If you are operating on huge amounts of data, then it can be worthwhile to look at the size of your data. If you have to pull a lot of data into the CPU from main memory, and you can reduce the size of your data, it can speed up your function.

- Can you scale your data down to a smaller size (float vs. double, uint16_t vs. long)?
- Is your data ordered from largest to smallest?

```
struct _a {
    char c;
    int i;
    char c2;
    double d;
};

sizeof(_a) == 24
```

```
struct _b {
    double d;
    int i;
    char c;
    char c2;
};

sizeof(_b) == 16
```

Expensive Operations

If the code makes several changes, X, Y, and Z, which all require doing expensive recalculations, then batch them up.

Set a dirty flag on each change (X, Y, and Z), then have a higher level function do one recalculation if the dirty flag is set.

Another approach is to split out expensive operations (e.g. recalculating data) from less expensive operations (e.g. using the data). Then only do the expensive operations when needed--use the less expensive operations when you can.

References

- *Effective C++*, 3rd edition*, Scott Meyers, Addison-Wesley, 2005.*
- *Exceptional C++*, Herb Sutter, Addison-Wesley, 1999.
- *Inside the Machine*, Jon Stokes, No Starch Press, 2006.