The Type of STL Iterators should be Independent of Containers' Comparators and Allocators

[paper outline]

Dan Tsafrir and Ziv Balshai
School of Computer Science and Engineering
The Hebrew University
Jerusalem 91904, Israel

Abstract

A collection of items is often maintained sorted by various criteria, such that iteration according to a specific order can be immediately performed on demand (without the overhead of sorting). A set of "for each" algorithms that differ only in the order in which the items are traversed, can be elegantly generalized into a single algorithm. Probably the most popular way to obtain this is by using the "iterator" design pattern, where the iteration is done using an abstract iterator, sub-classed by concrete iterators that are associated with the various sorting criteria. Unfortunately, this method might incur a heavy cost in terms of performance, due to the overhead of virtual functions.

Other alternatives do exist (surveyed in this paper), but can all introduce significant performance penalties and, in some cases, considerable complexity to the application. There is however one technique that is virtually drawback-free. This relies on the assumption that (e.g.) `set<Item*, cmp1>::iterator` and `set<Item*, cmp2>::iterator` are of the same type, as is the case for GCC's STL and STLPort. We therefore suggest that this assumption be made standard.

1 The Problem

Consider a batch-scheduler of a parallel supercomputer that maintains a queue of waiting jobs ("batch" means that jobs run to completion without being interrupted or preempted by other jobs). A newly submitted job is started immediately if there are enough free processors, or placed in a waiting queue if this is not the case. When a running job terminates and frees up its assigned processors, the scheduler traverses the waiting queue and attempts to start as many jobs as possible. The specific order of this traversal is determined in runtime and may change from time to time. In contrast, the code that actually starts the execution of a selected job is always the same.

Assume that the waiting queue can be very long and that arrival / termination of jobs is a very frequent event. Our goal is therefore to perform the scheduling as efficiently as possible, to prevent the scheduler form turning into a bottleneck. The following sections each offer a different solution to the problem, but only the last is drawback free.

2 The "Iterator" Design Pattern :(

Obviously, sorting the waiting jobs upon each and every scheduling decision is not an option, as this is a costly operation. Thus, we continuously maintain the jobs sorted according to the predefined sorting criteria. For simplicity, assume there are only two such criteria: jobs' arrival time and jobs' estimated runtime:

```
struct CmpA { bool operator(Job* a, Job *b) {return a->arrival  < b->arrival ;} };
struct CmpE { bool operator(Job* a, Job *b) {return a->estimate < b->estimate;} };
```

And so whenever a job is forced to wait it is inserted to the following waiting queues:

```
std::set<Job*,CmpA> wqA;
std::set<Job*,CmpE> wqE;
```

The scheduling algorithm may be implemented as follows (assume `p` is the required sorting criterion):

```
struct IterBase                { virtual Job* next()=0;} }; // return next Job, or 0 at end
struct IterA : public IterBase { /* wrapper around set<Job*,CmpA>::iterator */ };
struct IterE : public IterBase { /* wrapper around set<Job*,CmpE>::iterator */ };
IterBase i = (p==ARRIVAL) ? new IterA(wqA) : new IterE(wqE);
for( Job *job=i->next() ; job!=0 ; job=i->next() )
    if( can_start(job) )
        // start the job...
```

While this well known design pattern is indeed very elegant, the call to `i->next()` cannot be inline-d (it is virtual) and so there can be a significant penalty in terms of performance.

3 Pointer to Comparison Function :(

```
bool cmpA (Job *a, Job *b) { return a->runtime  < b->runtime ; }
bool cmpE (Job *a, Job *b) { return a->estimate < b->estimate; }
typedef bool (*CmpFunc) (Job *a, job *b);
typedef std::set<Job*,CmpFunc> WaitQ;
WaitQ wqA( &cmpA );
WaitQ wqE( &cmpE );
WaitQ::iterator begin = (p==ARRIVAL) ? wqA.begin() : wqE.begin();
WaitQ::iterator end   = (p==ARRIVAL) ? wqA.end()   : wqE.end()  ;
for(WaitQ::iterator job_i=begin ; job_i!=end ; ++job_i )
    if( can_start(*job_i) )
        // start the job...
```

The drawback of this solution is that calls to the `CmpFunc` cannot be inline-d, and so, while the iteration is indeed fast, the inserting/deleting of jobs to/from the waiting queues becomes expensive.

4 Code Repetition :(

```
if( p == ARRIVAL ) {
    for(set<Job*,CmpA>::iterator j=wqA.begin() ; j!=wqA.end() ; ++j)
    if( can_start(*j) )
        // start the job...
}
else{   // p == ESTIMATE
    for(set<Job*,CmpE>::iterator j=wqE.begin() ; j!=wqE.end() ; ++j)
    if( can_start(*j) )
        // start the job...
}
```

The drawback of this solution is obviously the code repetition, which is out of the question as `p` can potentially have a lot of values (that is, there can be a lot of sorting criteria).

5 Ordered Lists of Iterators :(

Whenever the scheduler decides a job must wait, it is inserted into the correct place within the following waiting queues:

```
typedef std::list<Job*> WaitQ;
WaitQ wqA; // jobs are inserted so that queue is sorted by arrival
WaitQ wqE; // jobs are inserted so that queue is sorted by estimate
WaitQ::iterator begin = (p==ARRIVAL) ? wqA.begin() : wqE.begin();
WaitQ::iterator end   = (p==ARRIVAL) ? wqA.end()   : wqE.end()  ;
for(WaitQ::iterator job_i=begin ; job_i!=end ; ++job_i )
    if( can_start(*job_i) )
        // start the job...
```

(Recall that if jobs are kept in an array, `Job*` can serve as an iterator of a `Job`). This solution is actually worse than the one mentioned in Section 3: While iteration is indeed fast, the inserting/deleting of jobs to/from the waiting queues becomes very expensive — O(n). Also, there's the added complexity of maintaining the lists sorted.

6 Eat the Cake and Leave it Whole :)

```
struct CmpA { bool operator(Job* a, Job *b) {return a->arrival  < b->arrival ;} };
struct CmpE { bool operator(Job* a, Job *b) {return a->estimate < b->estimate;} };
set<Job*,CmpA> wqA;
set<Job*,CmpE> wqE;
set<Job*>::iterator begin = (p==ARRIVAL) ? wqA.begin() : wqE.begin(); // [*]
set<Job*>::iterator end   = (p==ARRIVAL) ? wqA.end()   : wqE.end()  ; // [*]
for(set<Job*>::iterator i=begin; i!=end; ++i)
    if( can_start(*job_i) )
        // start the job...
```

Note that in [*] we assign an iterator of (say) the type

```
set<Job*,CmpA>::iterator // as returned from wqA.begin() / wqA.end()
```

into an iterator of the type:

```
set<Job*>::iterator      // the definition of begin/end
```

This works for GCC's STL and for STLPort because these are indeed the same type (the comparator's type doesn't influence the iterator's type). Note that in the above code, both the iteration and insertion code can be inline-d. Unfortunately, this is not the case for other distributions (e.g. for RW), for which these types are different. We therefore suggest considering adding to STL a requirement like:

```
TYPE[ container<T,cmp1>::iterator ] = TYPE[ container<T,cmp2>::iterator ]
```

(as it seems there's nothing to loose and a lot to be gained by adding it). The same argument may also be applied to other template parameters namely allocators.

Author's background

I'm a PhD student at the school of computer and engineering in the Hebrew University. My main research topics are operating systems, parallel & distributions systems, performance evaluation, and scheduling. In addition, in the past few years, I have been teaching several programming courses (C++, Java, UNIX Scripting) in the Hebrew University of Jerusalem and in Jerusalem's Hadassah College of Technology. Further derails can be found at: *http://www.cs.huji.ac.il/~dants*