Improving performance and maintainability through refactoring in C++11

J. Daniel Garcia
josedaniel.garcia@uc3m.es
Computer Science and
Engineering Department
University Carlos III of Madrid, Spain

Bjarne Stroustrup
Bjarne@Stroustrup.com
Computer Science Department
Columbia University, USA

August 27, 2015

Abstract

Abstraction based programming has been traditionally seen as an approach that improves software quality at the cost of losing performance. In this paper, we explore the cost of abstraction by transforming the PARSEC benchmark *fluidanimate* application from low-level, hand-optimized C to a higher-level and more general C++ version that is a more direct representation of the algorithms. We eliminate global variables and constants, use vectors of a user-defined particle type rather than vectors of built-in types, and separate the concurrency model from the application model. The result is a C++ program that is smaller, less complex, and measurably faster than the original. The benchmark was chosen to be representative of many applications and our transformations are systematic and based on principles. Consequently, our techniques can be used to improve the performance, flexibility, and maintainability of a large class of programs. The handling of concurrency issues has been collected into a small new library, YAPL.

1 Introduction

A traditional argument against abstraction-based programming has been loss of performance. Many believe that higher-level programming techniques necessarily introduce performance penalties. For example, it is widely assumed that C++ code must be slower than C code because very low-level hand optimizations are assumed necessary for performance. However, performing those low-level optimizations makes it harder to produce correct code and the resulting code is harder to maintain.

A typical approach for improving performance on multiprocessors is parallelization. Many approaches exist for shared memory machines from classical pragma based solutions as OpenMP [5], to library based solutions as Intel Threading Building Blocks (TBB) [20]. It is widely believed that producing a parallel version of an application requires rewriting, or at least a substantial modification of original source code. This introduces an additional burden for long term maintenance.

We challenge these assumptions with the following specific objectives:

- Explore the assumed abstraction penalty.
- Explore the capabilities added to C++ by C++11 [11] (the 2011 ISO C++ standard).
- Provide a framework for evaluating application optimizations and their associated costs.
- Allow for user source code to be agnostic about parallelism.

To evaluate our ideas, we apply extensive refactorings on an existing application: *fluidanimate*. This application is representative of physics simulations for animations.

The rest of this paper is organized as follows:

- Section 2 describes the *fluidanimate* application.
- Section 3 presents a set of basic refactorings applied to the application.
- Section 4 describes the design of the YAPL library.
- Section 5 presents additional refactorings applied to the *fluidanimate* application.
- Section 6 provides evaluation in terms of maintainability and performance.
- Section 7 provides references to related work.
- Section 8 states conclusions.

2 The *fluidanimate* application

We refactor *fluidanimate* [18], an application which is part of the publicly available PARSEC benchmark [2]. The application is documented as originally contributed by Intel and it uses an extension of the smoothed particle hydrodynamics (SPH) method to simulate an incompressible fluid.

The application reads from an external file information about a set of particles, and simulates their interactions as time advances during a number of iterations, supporting single or double precision computations. The benchmark provides a sequential version, as well as parallel versions using pthreads and Intel TBB.

The application is written in C++, but it makes hardly any use of C++ abstraction facilities. Essentially, it follows a C-style design expressed using a small, low-level, sub-set of C++:

- Data type abstraction is mostly used to define a `Vec3D` (3D-space vector) for making computation on physical magnitudes like position or velocity.
- Differences between single and double precision are managed by means of the preprocessor.
- Most functions neither take nor return parameters and communication between functions is performed via globals. There is a common belief that parameter passing hurts performance.
- Memory is managed through a highly sophisticated ad-hoc memory pool.

Fluidanimate uses `fstream` for reading and writing files. However, we concentrate on computation performance and have made no changes to I/O.

Please, note that our aim is to express the algorithms of *fluidanimate* more directly. We are not trying to provide a fundamentally different solution to the problems *fluidanimate* addresses, say by modifying the way threads and locking are used.

3 Basic refactorings

While analyzing the application we applied some general refactoring techniques that could be applied. We eliminated multiple uses of globals that could hurt both performance and maintainability. We also found that extensive use of structures of arrays were examples of premature optimizations and could be replaced by arrays of structures, again improving both performance and maintainability.

3.1 Removing globals

Many software developers assume that accessing global variables provides a performance advantage, so they use globals even when that approach seriously hurts maintainability of source code. However, on modern machine architectures and with modern compilers, global variables access may require more instructions per access than stack allocated variables. In particular, when a value is accessed multiple times a stack-allocated variables provide better performance.

We found the following uses of globals in *fluidanimate* source code:

- Application constants that are known at compile time.
- Application parameters whose values are computed on application initialization and do not change later.
- Mutable data shared among functions in the hope of improved performance.

Our first task in refactoring the *fluidanimate* code was to eliminate those globals.

3.1.1 Application constants

For application constants that are known at compile time, C++11 offers `constexpr`. For values that should not be modified after they have been initially computed at run time, C++ offers `const`. A key difference between `constexpr` and `const` is that the former by definition are evaluated at compile-time by the compiler.

Developers in many OO programming languages define constants as members of some application class. However, this approach often leads to dispersal of constants in the source code, implying dangers of duplication and inconsistency. In C++, for ease of comprehension and maintenance, we prefer to place those constants in the application namespace or in an inner namespace to that application namespace.

When an application framework is designed to be generic, the need of generic constants arises. In particular, *fluidanimate* needs single- and double-precision floating-point constants. C++11 did not offer direct support the notion of a generic constant, but C++14 does and variable templates are now supported by the major compilers.

Initially, we simulate generic constants by embedding them in a generic `struct` but with C++14 we can represent them directly (see Listing 1). Neither approach implies any cost in terms of memory usage or run time.

In summary, all the presented approaches avoid the problems of scope and tool support implied by using macros as symbolic constants.

3.1.2 Application parameters

The second class of globals (application parameters set on initialization) can be represented as a single class with constant public non-static data members. Then, an instance of this class becomes a data member of some context class and can be passed around if needed. This technique avoids global objects and consequently avoid the extra penalty derived from accessing far global data instead of near local data (see Listing 2).

3.1.3 Global data structures

Finally, global data structures can be eliminated by defining the proper abstractions and passing them as arguments to the appropriate functions. When removing globals through abstraction, special care needs to be taken to avoid run-time penalties. This step is also a perfect opportunity to identify and eliminate instances of premature optimizations. Optimizations should be performed only after careful measurement and ideally only if they do not compromise the important abstractions or interfaces. Importantly, this abstraction process gives an opportunity for removing code duplication which hurts both performance and maintainability.

In the *fluidanimate* application, we found globals to provide direct access to an ad-hoc memory pool and to directly manage linked lists of blocks of *cells* where each one contains a block of particle information.

We analyzed the code and found the set of composable abstractions for this application to be:

Listing 1: Generic constants as variable templates (C++14).

```
namespace fluid {
  namespace constants {   // present constants as variable templates (C++14)

    template <typename T>
    constexpr T viscosity = 0.4;

    template <typename T>
    constexpr space_vector<T>
      external_acceleration { T{}, T(-9.8), T{}};
    //...
  }
}

// ...

using namespace fluid::constants;
auto x = 2 * viscosity<double>;
auto a = external_acceleration<float>;
```

Listing 2: Application parameters encapsulation.

```
template <typename T>
class params {
  public:
    params(T ppm);
  public:
    const T h_;
    const T hsq_;
    const T h6_;
    const T density_coeff_;
    const T pressure_coeff_;
    const T  viscosity_coeff_ ;
  private:
    //  Initialization  helpers ...
};
```

- **cell** A 3D region of space containing a set of particles. A *cell* does not expose its implementation details. In particular, we do not use memory pools to manage particles and we do not use linked lists of blocks.
- **grid** A 3D matrix of *cells* with application specific processing. The main objective of the *grid* is to allow generalized processing of *cells* and their particles.
- **simulation** Provides a simulation management interface providing mechanisms for loading input data, generating output, and taking statistics if needed. It manages a *grid*.

Removing globals through abstraction requires insight into the application, domain knowledge. It cannot be just a mechanical substitution of one programming language feature with another.

3.2 Sequences and structures

Many developers prefer structures of arrays (namely parallel arrays) over arrays of structures. The latter often models the problem domain better, but the former is widely reported to provide better performance [17, pp.194–196]. In particular structures of arrays are supposed to improve the cache hit ratio. However, we found that this cannot be applied as a general rule. Instead, finding the best performance requires careful analysis and measurement for each case.

In *fluidanimate*, the structures of arrays pattern is systematically applied so that a *cell* contains parallel arrays for different information elements (as position, velocity or acceleration). However, the effect of this pattern depends highly on access patterns. In *fluidanimate* most operations in particles require simultaneously access to several information elements. Thus, intra-particle locality tends to be more important than inter-particle locality.

To improve the clarity of code, we decided to convert to an array of structures, by using `std::vector<particle>` (see Listing 3).

Listing 3: Particle type after removing structure of arrays.

```
template <class T>
class particle {
  public:
    particle(const space_vector<T> & p,
      const space_vector<T> & hv, const space_vector<T> & v);
    // ...
  public:
    space_vector<T> position_;
    space_vector<T> hv_;
    space_vector<T> velocity_;
    space_vector<T> accel_;
    T density_;
};

// Example of particle operation
template <typename T>
void particle<T>::advance() {
  using namespace constants;
  space_vector<T> v_half = hv_ + accel_ * time_step<T>();
  position_ += v_half * time_step<T>();
  velocity_ = hv_ + v_half;
  velocity_ *= 0.5;
  hv_ = v_half;
}
```

Based on the literature, we expected to pay a significant performance penalty for this improvement in code clarity and maintainability. However, it turned out that we gained in performance (see Section 6).

4 YAPL: Yet Another Parallel Library

To compare the sequential version of *fluidanimate* to its parallel versions, and to compare its parallel versions to each other, we separated the specification of concurrency from the specification of the application model. The aim is to provide the concurrency model as a parameter to the application.

To do that, we designed a simple library, YAPL [10], as an experimentation framework for containers and algorithms. YAPL provides an interface to a container library with key differences from other existing libraries.

YAPL design principles are based in a small set of ideals:

1. User code should be the same in sequential and parallel modes. This ideal is not always achievable, but we strive to minimize differences.
2. There should be a clear separation between a container abstraction (a list of values) and its supporting implementation structure (e.g., a linked list of nodes or a vector.).
3. For element access, YAPL uses mappings (a generalization of ranges). It does not use the C++ standard-library iterators because iterators are inherently sequential.
4. It should be easy to select the execution policy for an application (sequential, parallel, GPU, ...).

We have developed a prototype implementation of YAPL to offer minimal support for the refactoring of *fluidanimate* supporting a sequential version as well as a parallel version based on Intel TBB. Additional modes can be easily added. Our prototype offers a 3D cube of objects and its associated operations.

4.1 YAPL structure

YAPL offers components in five categories:

- A **container** provides a minimal interface for the container abstraction. A key property of a container is that it provides mechanisms for generating mappings to that container. An example is `cube`.
- A **support structure** implements a concrete strategy that can be used by multiple containers. An example is `block`.
- A **mapping** provides an interface to operate on a subset of container elements. Different subsets may be generated through different defining criteria. Examples are `full_cube_mapping` or `plane_cube_mapping`.
- An **algorithm** is an operation that can be applied to any mapping. A general example is the algorithm `apply`.
- A **policy** allows the programmer to configure how operations are sequenced and elements allocated. Examples are `sequential_policy` or `tbb_policy`.

4.2 Cubes in YAPL

A *cube* is a generic container that provides access to a 3D matrix of objects. All containers in YAPL are parametrized in terms of their element type and a *policy*. The latter allows for easily defining containers with different policies for execution or memory management.

A *Cube* (for a given cell type and policy) can be created by specifying its size for each dimension (see Listing 4). Those sizes can be easily retrieved from a *Cube*. Alternatively, sizes and indices can be managed using a `cube_index` type.

Both styles can be used for accessing an element from the *cube* (see Listing 5).

However, explicit use of indexing should be avoided when traversing *cubes*:

- Performance is worse than traversing the cube through a *mapping*.
- Explicit traversal makes transformations to parallel implementations harder than implicit traveral.

A *mapping* is a general YAPL concept for referencing a set of elements in a container and performing an operation on those elements. In essence, a *mapping* defines a subset of elements in the container. Several functions in the container library allow to obtain mappings from a container (see Listing 6).

Combining containers and mappings it is easy to define algorithms to perform general operations on YAPL mappings. We use lambda expressions to provide operations to be applied to every element in a mapping.

Listing 4: Cube interface.

```
using cube_type = yapl::cube<cell_type, policy>;
cube_type c{10, 15, 20};

size_t  nx = c.size<0>(); // nx=10
size_t  ny = c.size<1>(); // ny=15

cube_index sz{10,15,20}:
cube_type d{sz};

auto size = d.size(); // cube_index{10,15,20}
```

Listing 5: Accessing cube elements by indices and cube_index

```
do_something(c(2,3,4));          // access using integer indices
c(2,3,4).do_something_else();

cube_index i{2,3,4};
do_something(c(i));              // access using a cube_index
c(i).do_something_else();
```

Listing 6: Obtaining different mappings from a container.

```
// m1 is a mapping to all elements in cube c
auto m1 = c.all();

// m2 is mapping to elements in plane 10
// on y-dimension of cube c
auto m2 = c.plane<1>(10);
```

Listing 7: Applying a generalized operation to a mapping through a lambda expression.

```
apply(my_cube.all(),
   [](cell_type & c) {
     c.do_whatever();
   }
);
```

Listing 8: Applying a generalized operation to a mapping including indices

```
apply_indexed(my_cube.all(),
   [](cell_type & c, const cube_index & i) {
     c.init(i);
   }
);
```

Most operations on mappings can be expressed without providing extra details. However, in a few cases we need to provide the cube with element coordinates (see Listing 8).

Many applications dealing with cubic structures must apply an operation to all the neighbours of an element. Also, those operations sometimes need to be applied only to the half of neighbours to avoid duplication in symmetric operations.

Listing 9: Application of an operation to a neighbourhood.

```
vector<cell_type*> n1;
my_cube.for_all_neighbours(i, [&n1](cell_type & c) {
  n1.push_back(&c);
});

// Avoids duplicates when traversing cubes
vector<cell_type*> n2;
my_cube.for_all_neighbours_unique(i, [&n2](cell_type & c) {
  n2.push_back(&c);
});
```

4.3 YAPL and strategy encapsulation

Policies are used in a consistent manner for all classes in a name space. Unfortunately, C++ does not offer namespaces with template parameters. In the absence of parameterized namespaces, the common approach is to use a policy parameter in all classes. However, multiple orthogonal policies tend to emerge over time, leading to a potentially confusing list of policy template parameters. We mitigate that situation by aggregating all policies in a single `policy` class (see Listing 10).

Listing 10: Policy aggregation

```
template <typename E, typename A>
class policy {
  using executor_type = E;
  using allocator_type = A;
};

template <typename T>
using sequential_policy =
  policy<sequential_executor<T>, std::allocator<T>>;

template <typename T>
using tbb_policy =
  policy<tbb_executor<T>, std::allocator<T>>;
```

The *using-declarations* provide aliases for useful policy types. This strategy of policy parameterization allows that application code can be almost identical for different versions of the application without run-time overheads (see Listing 11).

Listing 11: Policy based algorithm selection.

```
template <typename P>
void run_simulation(double ppm, size_t np) {
  // ...
  using simulation_type = simulation<double,P>;
  simulation_type sim{ppm,np};
  sim.read( file );

  for ( size_t  i=0;i<np;++i) {
    sim.advance_frame();
  }
}

void run_sequential_simulation(double ppm, size_t np) {
  run_simulation<sequential_policy<double>>(ppm,np);
}

void run_tbb_simulation(double ppm, size_t np) {
  run_simulation<tbb_policy<double>>(ppm,np);
}
```

5 Additional refactorings

5.1 Refactoring a grid with YAPL

With YAPL, the `grid` abstraction can be easily expressed in terms of a `yapl::cube` (see Listing 12).

Listing 12: Grid expressed in terms of cube

```
template <typename T, typename P>
class grid {
  public:
    grid(T ppm);
    // ...
    void rebuild_grid();
    void compute_forces();
    void process_collisions ();
    void reprocess_collisions ();
    void advance_particles();
  private:
    const params<T> params_;
    const domain<T> domain_;

    using cell_type = typename P::cell_type;
    using grid_policy = typename P::grid_policy;
    using cube_type = yapl::cube<cell_type, grid_policy>;

    cube_type cells_;
    cube_type cells2_;
};
```

As an example of how an algorithm can be expressed in terms of YAPL, we provide here the implementation of the `advance_particle` member function (see Listing 13).

Listing 13: Implementing grid algorithms with YAPL.

```
template <typename T, typename P>
void grid<T,P>::advance_particles() {
  yapl::apply( cells_.all(), []( cell_type & c) {
    c. for_all_particles ([]( particle<T> & p) {
      p.advance();
    });
  });
}
```

5.2 Avoiding Data Races

Our ideal is to make the application code agnostic in respect to concurrency. However, that this is not always possible. One example is the `cell` abstraction, where a parallel version may need concurrent accesses to cells.

To avoid data races, we protect such accesses with a `mutex`. However, that would penalize sequential versions, which would be locking without need. To avoid that, we make the *mutex* a template parameter of the `cell` and we make use of a *null mutex* for the sequential case (see Listing 14).

Listing 14: A cell with a generalized mutex.

```
template <typename T, typename M>
class cell {
  public:
    cell();
    void add_neighbour(cell<T,M> & c);
    void add_particle(const particle<T> & p);
    void emplace_particle(const space_vector<T> & pos,
      const space_vector<T> & hv, const space_vector<T> &v);

    template <typename F>
    void for_all_particles (F f);

    template <typename F>
    void for_all_near_particles (F f);

  private:
    std::vector<particle<T>> particles_;
    std::vector<cell<T,M>*> neighbours_;
    mutable M mutex_;
};
```

With that approach, writing algorithms that lock only when needed is trivial. The programmer simply protect critical regions with a *lock_guard* and the locking operations are completely eliminated (at compile time) in serial executions (see Listing 15).

The *lock_guard* [22, 27] is a standard-library (RAII) type that ensures that a lock on a mutex is released at the end of its scope, so that no explicit unlock operation is needed (and thus cannot be forgotten).

Listing 15: Examples of generalized locking.

```
template <typename T, typename M>
void cell<T,M>::add_particle(const particle<T> & p) {
  std::lock_guard<M> l{mutex_};
   particles_.push_back(p);
}

template <typename T, typename M>
template <typename F>
void cell<T,M>::for_all(F f) {
  std::lock_guard<M> l{mutex_};
  for (auto & p : particles_) {
    f(p);
  }
}
```

6 Evaluation

6.1 Source code metrics

To evaluate the impact of refactoring on source code, we used basic source code metrics. Table 1 shows size metrics of original and refactored source code. For the original code two versions are analyzed independently: the sequential version and the parallel TBB-based version. For the refactored version figures are given for the refactored source code and the YAPL library. Note that the refactored version offers in a single code base where the original PARSEC vesion required separate sequential and the parallel versions. A key advantage of our approach is that a single code base needs to be maintained.

Table 1: Source code size metrics for original and refactored application

Metric	Seq.	Parallel	Total	Refactored	YAPL	Total
Logical LOC	717	961	1678	829	454	1283
Number of functions	35	52	87	165	110	275
LLOC per function	20.49	18.48	19.29	5.02	4.13	4.67
Number of classes	5	15	20	23	16	39

Even counting YAPL library, the resulting code base is around 25% smaller than the original code base. The reason for this is that generic programming made it easy to remove replicated source code. The refactored version uses significantly more classes and functions, reflecting a higher level of abstraction. The average length of a function dramatically shrank to 25% of the original.

To compare the original and resulting source code we used cyclomatic complexity.

Table 2 shows cyclomatic complexity measures for original and refactored source code. The refactored source code shows smaller figures for function complexity in both maximum and average values. In particular, we got a very high reduction for the maximum function complexity (from 61 to 12). We also got a drastic reduction in the average complexity (from 5.33 to 1.46).

In summary, the refactored code is simpler and (in our opinion) far more maintainable. Such improvements are often assumed to come at the cost of performance.

Table 2: Cyclomatic complexity metrics for original and refactored application

Metric	Seq.	Parallel	Total	Refact.	YAPL	Total
Maximum complexity	33	61	61	8	12	12
Aggregate complexity	198	266	464	197	204	401
Average complexity	5.66	5.11	5.33	1.19	1.85	1.46

6.2 Performance metrics of sequential version

For performance evaluation, we used two different machines of different size. The first machine was a single socket board with one Intel Core i7-2600 processor and 4GB of main memory. The processor has 4 cores plus the ability of using hyper-threading and a shared L3 cache of 8 MB. The machine runs an Ubuntu 12 (kernel 3.2) distribution.

The second machine was a dual socket board with two Intel Xeon E5-2695 processors and 128 GB of main memory. Each processor has 12 cores plus the ability of using hyper-threading and a shared L3 cache of 30 MB. The machine runs a CentOS 6.5 Linux (kernel 2.6) distribution.

For compilation, we used g++ version 4.8.2 with all optimizations activated (`-O3`) and C++11 mode (`-std=c++11`).

In all executions of the benchmark, we measured total execution time, including processing and file I/O. I/O is a fixed cost which is not increased with the number of iterations. For every given setup we executed the application independently 10 times and we took average values. We did not perform a higher number of experiments as we got very low variations (below 1%).

We performed executions of the benchmark with the standard 500K particles. Figure 1 shows global execution of both original and refactored versions when the number of iterations increases. We have performed measurements until 2000 iterations. However, the standard number of evaluations for this benchmark is 500 iterations.

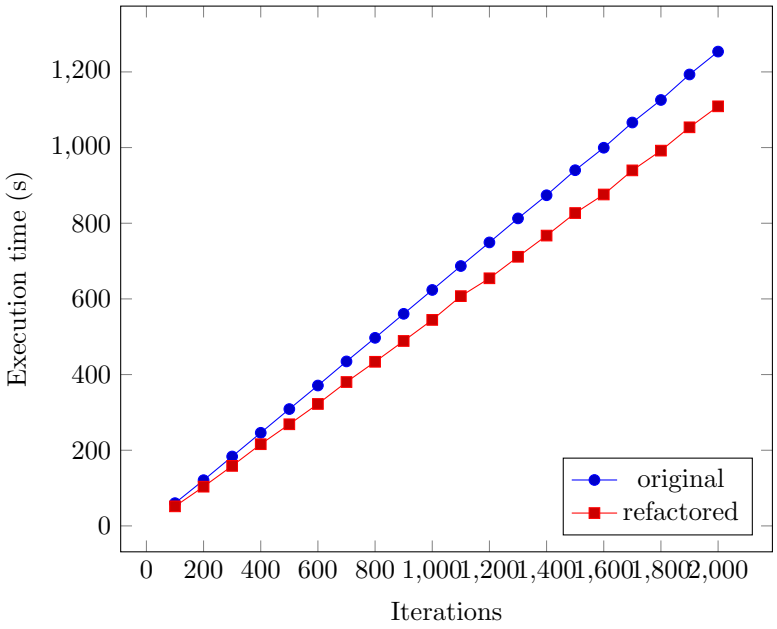

Figure 1: Global execution time versus number of iterations for sequential version

Figure 2 shows the execution time per iteration.

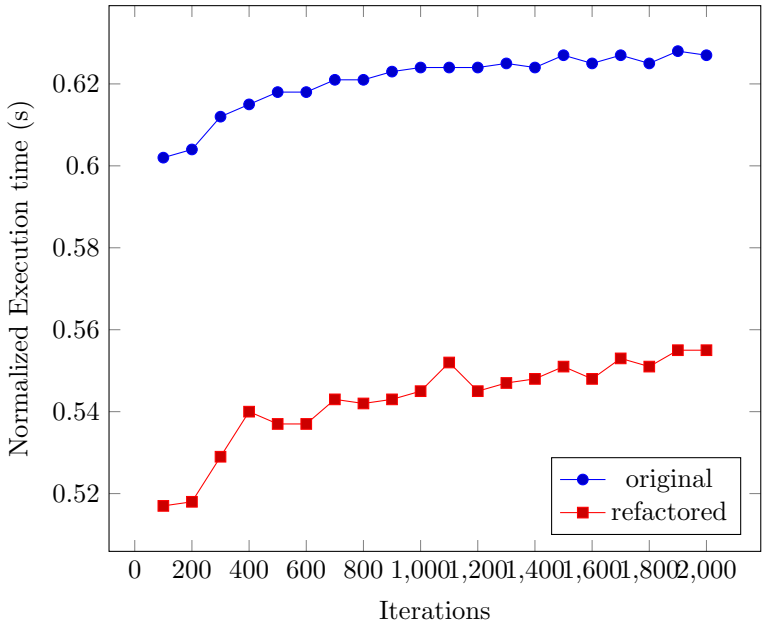

Figure 2: Normalized execution time per iteration for sequential version

For very low numbers of iterations, the original version performs better than for higher number of iterations because initial conditions of the problem locate most particles in very few cells. This leads to less dynamic memory operations. As simulation time progresses to realistic values, particles tend to move to different cells requiring more memory management operations and increasing execution time. Then, the refactored version consistently requires less time per iteration.

This improvement is in stark contrast to the conventional "wisdom" that we need to carefully hand-optimize using low-level features to get acceptable performance.

Figure 3 shows the speedup of the refactored application over the original for different number of iterations. While for very low number of iterations the speedup is more than 1.16 (0.52s vs. 0.6s), this value decreases when the number of iterations increases up to 1.13 (0.54s vs. 0.62s) for 2000 iterations. For the standard number of iterations for the benchmark we got a speedup of 1.15. The point here is not the amount of speedup obtained for the higher-level refactored version, but that there is an improvement at all. Consequently we did not find a penalty due to the introduction of higher level abstractions and the elimination of lower level optimizations.

6.3 Performance metrics of a parallel version

To compare the parallel versions, we used the original Intel Threading Building Blocks version from the PARSEC benchmark as our baseline. In our application, we took the same source code and compiled it using the parallel implementation of YAPL based on Intel Threading Building Blocks. This required no major application source code modification. To contrast, in the original source code base, the sequential and parallel versions differ greatly in many places, which could easily become a maintenance nightmare.

To measure the effect of the number of threads, we fixed the number of iterations to 500 and we varied the number of threads. Figure 4 shows the execution time when threads are varied from 1 to 64 for the 24-core machine. For both our version and the original version, the maximum speedup compared to the serial version (about 10 times) is found for 32 to 64 threads. We observed a small speedup for our version compared to the original version for small number of threads (12% for 4 threads). However, in general performance is essentially

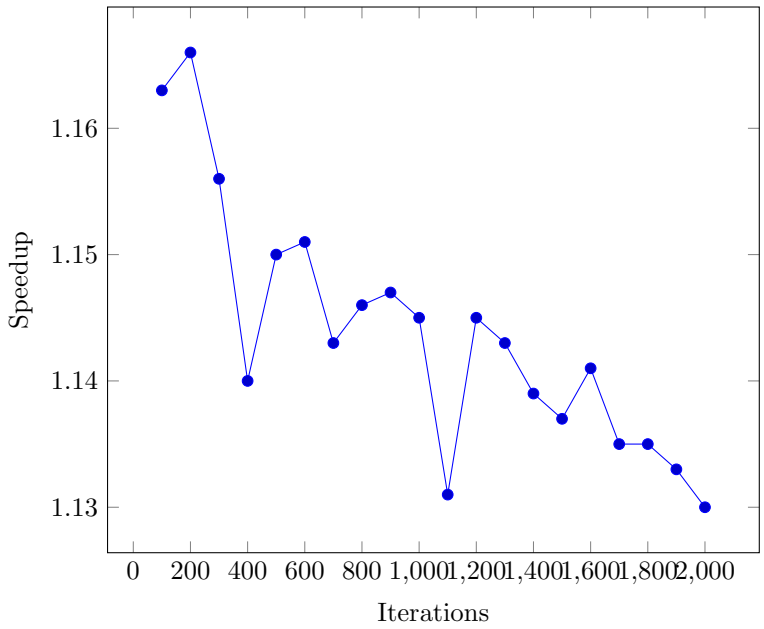

Figure 3: Speedup of refactored application for sequential version

the equivalent for both versions of the application with no significant loss of performance for the much simpler source code of the refactored version.

Figure 5 shows the execution time when threads are varied from 1 to 8 for the 4-core machine. We observed much higher speedups in this case, which reaches to values higher than 40% for the best cases with 4 and 8 threads.

The key difference between the 4-core and the 24-core machines is the size of L3 cache (30 MB versus 4 MB). On the 24-core machine, the better cache performance of our implementation is neutralized by the larger cache. This is to be expected: better use of the cache matters only when the amount of data is large relative to the size of the cache.

6.4 Analysis of performance counters

To better explain the reasons behind the better performance of our approach, we took hardware information by means of performance counters. We fixed the number of iterations to 500 and obtained performance counters of the original and refactored sequential versions for both single and double precision versions. We repeated each measurement 10 times and took average values.

Table 3 shows measurements of instruction execution for double precision. Very similar results are obtained for single precision.

Table 3: Instruction performance counters

Counter	Orig.	Refact.	Ratio
Instruction Count	1,551,283,161,425	1,229,633,929,251	0.793
Instructions per cycle	1.580	1.437	0.909
Branch instructions	192,188,523,664	115,360,411,922	0.600
Branch miss ratio	0.80%	1.03%	1.281

We consistently obtained a reduction in the number of instructions effectively executed of about 20%. We

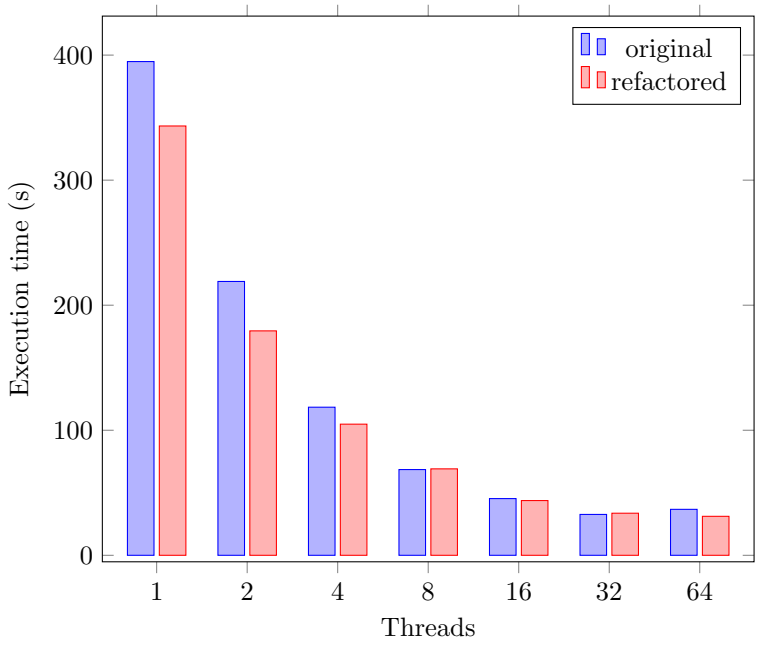

Figure 4: Global execution time for different number of threads in a large machine

observed that a fraction of this reduction came from a 40% reduction of branch instructions, although branch miss ratio increased from 0.8% to 1.0%. Thus, a key factor in the performance improvement was an important reduction in the number of instructions being executed. There are several changes that contributed to this instruction reduction.

First, the use of `constexpr` for true constants improves code generation when those constants are used in expressions. This is especially true for constants of non-primitive types (e.g., constants of `space_vector<T>` in this application.

A second source of instruction reduction comes from the traversal of the grid cells for the cases where an operation needs to be applied to all cells in the grid. Our generic algorithm uses a single loop instead three nested loops needing less branch instructions.

Finally, our grid data structure uses contiguous memory, instead of a linked list of nodes. This greatly simplifies the code for traversing the grid. We recognize that our data cache performance shows some degradations. However, that degradation is more than compensated by the fact that traversing a linear data structure is much simpler to optimize than traversing a linked list of blocks, where each block contains a small array.

Table 4 shows measurements of cache performance for double precision. We focus here in access to data showing counters for L1D cache and LL cache.

In terms of L1 cache data access, the refactored solution requires more data loads (an increase of 19.5%) and stores (an increase of 36%). Globally, cache references increases in 21% with an increase in miss rate of about 35%. The net effect is that L1 cache data miss ratio goes from 2.5% in the original application to 2.8% in the refactored application.

However, at the last level cache the number of loads is roughly the same with a reduction of load misses of about 10%, which gives a small reduction in load miss ration (from 48% to 44%). However, the number of stores is drastically reduced to 32% of original stores, and misses are also reduced to 25% of original misses. The net effect here is there are less LLC references (71% of original references with the miss ration going down from 56% to 46%.

The composed cache effect is that our solution introduces some overhead in terms of data access. For example our application `grid` is a `yapl::cube` of cells (where each cell keeps a dynamic `std::vector` of particles). Thus

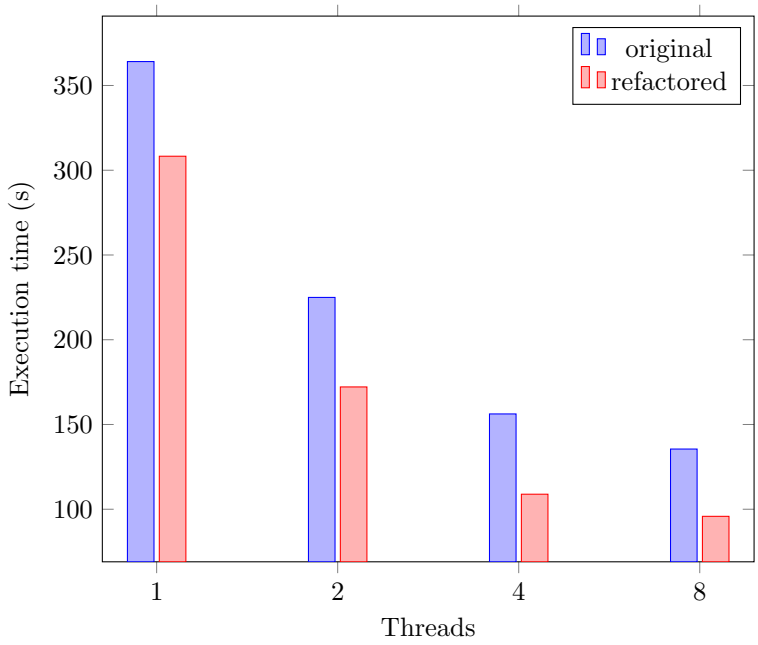

Figure 5: Global execution time for different number of threads in a smaller machine

Table 4: Cache performance counters

Counter	Original	Refactored	Ratio
L1D loads	307,475,745,493	367,497,050,132	1.195
L1D load misses	6,675,053,456	8,145,499,302	1.220
L1D load miss ratio	2.17%	2.22%	1.021
L1D stores	33,899,889,447	46,123,150,112	1.361
L1D store misses	1,829,699,692	3,402,246,983	1.859
L1D store miss ratio	5.40%	7.38%	1.367
L1D references	341,375,634,940	413,620,200,244	1.212
L1D misses	8,504,753,148	11,547,746,285	1.358
L1D miss ratio	2.49%	2.79%	1.121
LLC loads	3,450,090,601	3,415,907,907	0.990
LLC load misses	1,670,299,537	1,503,452,051	0.900
LLC load miss ratio	48.41%	44.01%	0.909
LLC stores	772,690,374	250,333,416	0.324
LLC store misses	703,538,509	181,604,136	0.258
LLC store miss ratio	91.05%	72.55%	0.797

this incurs an additional memory management overhead. It is however remarkable that the data access overhead is mostly handled in the cache hierarchy and that the number of data references resulting in a miss is lower that in the original application.

7 Related work

The C++ programming language has evolved over the years. Its latest approved standards [11, 12] are usually referred as C++11 and C++14. A good overview of these new versions can be found in [24] while a more complete description of the language and its accompanying standard library can be found in [25]. There are a number of recent features incorporated into C++ that made this work possible:

- Compile-time constant expression data definition [8] through `constexpr`, allows for symbolic constants that do not need to be initialized at program startup.
- Compile-time evaluation [8, 7] of functions (also denoted as *constexpr functions*), allows a way to initialize `constexpr` objects at compile time. This feature also provides a way to simulate generic compile-time constants. This workaround is not needed as the next version of C++ becomes available.
- Definition of synonyms for generic types introducing bindings for some of their parameters, namely *template aliases* [26, 6] made the code of YAPL as well as the client code much simpler.
- An extensive use of type deduction through the use of `auto` [14, 15] as type specifier also simplified the code and made it more easy to evolve.
- The possibility of defining *lambda expressions* [28, 13] as an easy way to pass around client code that needs to be invoked from the library.

A first try to refactoring and global variables can be found in [9]. A recent work on refactoring of global variables in C can be found in [21]. Another context in which global elimination has been used is thread safety [23]. However, while their approach focuses in thread safety the work presented here shows that performance is not hurt.

The array-of-structures versus structures of arrays duality can be solved by refactoring approaches. Condit [4] propose data slicing as a mechanism for preserving interfaces while improving cache locality. Kjolstad [16] split structs in hot and cold fields by splitting a struct into two, where one part contains the fields that are accessed often and the other contains the fields that are accessed less often. The structs are then stored in different arrays.

Both *Microsoft Parallel Pattern Library* [3] and *Intel Threading Building Blocks* take a common approach for providing a library solution to parallelism with a set of generic parallel algorithms (e.g. `parallel_for`, `parallel_find`, ...). To traverse data structures they provide the concept of a *range* which is less constrained than the *iterator* concept. YAPL's *mapping* is a generalization of such concept. A similar approach is taken by *Thrust* [1] for providing access to parallelism in CUDA capable GPUs. In all these cases, the libraries preserve a high degree of resemblance to C++ Standard Template Library.

Another parallel library with similarities with YAPL is STAPL [19]. While STAPL addresses both shared and distributed memory, YAPL only focuses in shared memory programming models. Both designs define their own set of *containers* instead of adding restrictions to be STL compatible. However a key difference in YAPL is the distinction between the interface (the *container*) and the representation infrastructure (the *support structure*). STAPL uses two combined concepts for traversing: *views* and *pRange*. In contrast YAPL defines single concept: a *mapping* for that purpose.

8 Conclusions

Many software developers make the assumption that abstraction-based programming necessarily incur a significant performance cost. An example is the belief that low-level hand-optimized C code always give better performance than a higher-level C++ counterpart. To challenge these assumptions we applied several refactoring techniques to a hand-optimized physics simulation application, the PARSEC benchmark *fluidanimate*. We removed the many globals from the original source code and simplified it by removing optimizations. For

example, we replaced arrays with implicit relationships with a single arrays of structures. We also eliminated the extensive use of hand-optimized memory pools. Besides, we defined an experimental application framework designed to allow that the same source code may express the sequential and the parallel version of an application: YAPL.

In the design of YAPL we applied a small set of design principles *mode agnosticism* (no key difference between sequential and parallel version of an application), *separation of concerns* (a layer for containers and a another layer for support structures of those containers), *no-iteration* (instead of making adaptations and workarounds a different concept of a *mapping* is used), and *use of policies* (making easy to select the execution policy to be used). We made extensive use of these principles in the whole design of YAPL.

We were able to have a single version of the *fluidanimate* application code leaving the execution mode as a setup parameter. Thus, we encapsulated the differences among versions in the YAPL library, while all the problem domain logic remains in the application code. Our refactoring led to an important reduction in the number of lines of code in the application (23%), while we increased the number of functions (316%) and classes (195%), indicating an increase in the level of abstraction. At the same time, the complexity of functions was reduced (by 72% for the average and 80% for the maximum) as well as the number of logical lines per function (by 75%). In summary, these refactorings produced shorter, simpler, and more easily maintainable source code.

Given the introduction of a higher level of abstraction and many small functions, a loss of performance would be widely expected and a slight loss would be acceptable. However, for the sequential version of the application the refactored version gave a slight performance improvement (about 1.10 for a standard number of iterations in the benchmark). Moreover, for the parallel version of the application, the refactored version gives a speedup around 1.45 compared with a hand-written parallel version. Both versions use Intel TBB. The YAPL-based version got better performance than the original PARSEC version for any number of underlying threads.

We credit our improvements to (generated) code reduction and improved cache locality. Looking at hardware counters we find that our refactorings eliminated almost 20% of instructions executed, with a similar reduction in the number of branch instructions executed. This gave a small reduction in instruction cache loads and also improved the instructions cache hit ratio. We found that our solution required more cache data references. However, the increase in load misses was compensated with a drastic reduction in store misses.

Acknowledgments

The authors would like to thank original authors and maintainers of the PARSEC benchmark.

J. Daniel Garcia's work was partially supported by Fundación CajaMadrid through their grant programme for Madrid University Professors.

Bjarne Stroustrup's work was partially supported by NSF grant #0833199.

References

[1] N. Bell, J. Hoberock, and C. Rodrigues. Thrust: A Productivity-Oriented Library for CUDA. In D. B. Kirk and W. mei W. Hwu, editors, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, Burlington, MA, USA, 2nd. edition, Dec. 2012.

[2] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, NJ, USA, Jan. 2011.

[3] C. Campbell and A. Miller. *Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, Apr. 2011.

[4] J. Condit and G. C. Necula. Data slicing: Separating the heap into independent regions. In R. Bodik, editor, *Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science*, pages 172–187. Springer, Berlin, Heidelberg, Germany, 2005.

[5] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan. 1998.

[6] G. dos Reis and B. Stroustrup. Template aliases (Revision 3). ISO/IEC JTC1/SC22/WG21 Working Paper N2258, ISO/IEC, Apr. 2007.

[7] G. dos Reis and B. Stroustrup. General Constant Expressions for System Programming Languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2131–2136, New York, NY, USA, 2010. ACM Press.

[8] G. dos Reis, B. Stroustrup, and J. Maurer. Generalized Constant ExpressionsRevision 5. ISO/IEC JTC1/SC22/WG21 Working Paper N2235, ISO/IEC, apr 2007.

[9] R. Fanta and V. Rajlich. Restructuring legacy c code into c++. In *IEEE International Conference on Software Maintenance*, ICSM '99, pages 77–85, Los Alamitos, CA, USA, Aug. 1999. IEEE Computer Society.

[10] J. D. Garcia. Yapl – yet another parallel library. https://github.com/jdgarciauc3m/yapl, 2013. [Online; available since 18-Jul-2015].

[11] ISO/IEC. Information Technology – Programming Languages – C++. International Standard ISO/IEC 14882:2011, ISO/IEC, Geneva, Switzerland, Aug. 2011.

[12] ISO/IEC. Information Technology – Programming Languages – C++. International Standard ISO/IEC 14882:2014, ISO/IEC, Geneva, Switzerland, Dec. 2014.

[13] J. Järvi and J. Freeman. C++ lambda expressions and closures. *Science of Computer Programming*, 75(9):762–772, Sept. 2010.

[14] J. Järvi, B. Stroustrup, and G. dos Reis. Deducing the type of variable from its initializer expression. ISO/IEC JTC1/SC22/WG21 Working Paper N1721, ISO/IEC, Oct. 2004.

[15] J. Järvi, B. Stroustrup, and G. dos Reis. Deducing the type of variable from its initializer expression (revision 4). ISO/IEC JTC1/SC22/WG21 Working Paper N1984, ISO/IEC, Apr. 2006.

[16] F. Kjolstad, D. Dig, and M. Snir. Bringing the HPC Programmer's IDE into the 21st Century through Refactoring. In *SPLASH 2010 Workshop on Concurrency for the Application Programmer*, CAP' 10, New York, NY, USA, Oct. 2010. ACM Press.

[17] M. McCool, A. D. Robinson, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, Waltham, MA, USA, July 2011.

[18] M. Müller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, 2003. Eurographics Association.

[19] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard templates adaptive parallel library (stapl). In D. R. O'Hallaron, editor, *Languages, Compilers, and Run-Time Systems for Scalable Computers*, volume 1511 of *Lecture Notes in Computer Science*, pages 402–409. Springer, Berlin, Heidelberg, Germany, 1998.

[20] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, Sebastopol, CA, USA, July 2007.

[21] H. Sankaranarayanan and P. A. Kulkarni. Source-to-source refactoring and elimination of global variables in c programs. *Journal of Software Engineering and Applications*, 6(5):264–273, May 2013.

[22] D. C. Schmidt. Strategized locking, thread-safe decorator, and scoped locking: Patterns and idioms for simplifying multi-threaded c++ components. *C++ Report*, 11(9), Sept. 1999.

[23] A. R. Smith and P. A. Kulkarni. Localizing globals and statics to make c programs thread-safe. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '11, pages 205–214, New York, NY, USA, Oct. 2011. ACM Press.

[24] B. Stroustrup. *A Tour of C++*. C++ In Depth Series. Addison-Wesley, Boston, MA, USA, Oct. 2013.

[25] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, USA, 4th edition, May 2013.

[26] B. Stroustrup and G. dos Reis. Template aliases for C++. ISO/IEC JTC1/SC22/WG21 Working Paper N1489, ISO/IEC, Sept. 2003.

[27] M. Suess and C. Leopold. Generic locking and deadlock-prevention with c++. In C. H. Bischof, H. M. Bcker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, and F. J. Peters, editors, *Parallel Computing: Architectures, Algorithms and Applications*, ParCo '07, pages 211–218, Aachen, Germany, Sept. 2007. IOS Press.

[28] J. Willcock, J. Jarvi, D. Gregor, B. Stroustrup, and A. Lumsdaine. Lambda expressions and closures for C++. ISO/IEC JTC1/SC22/WG21 Working Paper N1968, ISO/IEC, Feb. 2006.