Vertical optimization of particle in cell code

by

Viktor Przebinda

B.S. University of Colorado, 2003

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Electrical Engineering
2005
This thesis entitled:
Vertical optimization of particle in cell code
written by Viktor Przebinda
has been approved for the Department of Electrical Engineering

________________________________________
Dan Connors

________________________________________
John R. Cary

________________________________________
Andrew Pleszkun

Date ____________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.
Przebinda, Viktor (M.S. Electrical Engineering)

Vertical optimization of particle in cell code

Thesis directed by Prof. Dan Connors

VORPAL has achieved significant performance improvements through the introduction of sorting, dynamic load balancing, parallel building, and runtime code compilation. Particle sorting allows us to increase temporal locality of data structures, thus maximizing utilization of a computer’s cache. Dynamic load balancing continuously adjusts load distribution in a parallel environment to maximize the utilization of all processors. Runtime code compilation increases application input flexibility while maintaining performance. These optimization methods are controlled by cost minimizing functions that seek to find a balance between overhead in performing them and the performance improvement they provide. These techniques combined have contributed to an overall tripling of execution time.
Acknowledgements

Michael Main, for help in developing the C++ iterator. Chet Nieter for his help with my understanding of VORPAL.
## Contents

**Chapter**

1 Introduction 1

2 Introduction to VORPAL 4

2.1 VORPAL PIC structure 4

3 Particle Sorting 6

3.1 Sorting through an STL iterator interface 7

3.1.1 A case for Interfacing of custom implementation 8

3.1.2 Data Structure 9

3.1.3 Iteration and Indexing 13

3.1.4 Returning references and values 16

3.1.5 Performance 21
3.2 Dynamic cost optimization ...................................... 23
3.3 Sorting Performance ............................................. 24

4 Dynamic Compilation .............................................. 27
4.1 System Usage ..................................................... 27
4.2 Design Pattern ................................................... 29
  4.2.1 Shared libraries and dynamic execution .................... 29
  4.2.2 Implementation of a platform independent dynamic compiler . 30
  4.2.3 Concurrency Issues in file management ..................... 31
  4.2.4 Writing and compiling ....................................... 33
  4.2.5 Loading the shared library .................................. 33
  4.2.6 Creating and destroying Instances ......................... 34
4.3 Performance analysis of dynamic compilation .................. 35

5 Dynamic Load Balancing .......................................... 39
  5.0.1 Parallel Framework ......................................... 40
  5.0.2 Implementation of a dynamic load balancer for VORPAL .... 41
  5.0.3 Load balancing: A case study ............................... 44
## 6 Parallel Build

6.0.4 Application build process ........................................... 47

6.0.5 Parallel Build algorithm .............................................. 47

6.0.6 Parallel Build Implementation ...................................... 48

6.0.7 Performance of Parallel Build ..................................... 49

## 7 Discussion and concluding remarks

## Bibliography
Tables

Table

3.1 Relative sorting times of random and sequential data arrays 22
Figures

Figure

3.1 Sorted (left) vs. unsorted particles (right) ...................................... 7
3.2 Parallel Segmented Arrays .............................................................. 9
3.3 Re-Indexing performance ................................................................. 26
3.4 Optimal sorting step makes sorting time equal to excess particle update
time ........................................................................................................ 26
4.1 Dynamic class design pattern ............................................................ 37
4.2 Compilation vs. Interpretation ........................................................... 38
5.1 Boundary adjustment is limited by direction. Direction zero divides the
entire domain. Direction one boundaries divide individual divisions made
by direction zero boundaries. Direction two boundaries divide individual
slices made by direction one. ................................................................. 40
5.2 Load Balancing ................................................................................. 43
5.3 Over Allocation .................................................. 44

5.4 Load Balancing .................................................. 45
Chapter 1

Introduction

The Particle in Cell (PIC) algorithm is a powerful and highly accurate tool for simulating physical effects for a variety of applications. PIC is also relatively computationally expensive compared to other methods. This paper discusses a few significant performance-related improvements to the PIC model that are implemented in the University of Colorado Center for Integrated Plasma Studies’ Versatile Object-oriented Plasma Simulation Code (VORPAL). These improvements have lead to an overall tripling performance improvement.

The PIC model imposes some difficult challenges for modern day processors. Particles are updated by factoring in the forces they experience from electric and magnetic fields, which themselves are updated by factoring in the electromagnetic (EM) forces generated by particle movement. Since all these physical entities are stored in entirely separate and dissimilar data structures, this kind of computation results in a large processor - memory gap. The CPU is therefore starved of work waiting on memory requests to return. Particle sorting aims to reduce this wait time by keeping the needed memory in cache, closer to the processor, thus reducing the time required to retrieve it. We use particle sorting, discussed in Chapter 3 to align each particle’s actual position in memory with its position on the grid where field data is stored. In this way, all memory
access patterns become sequential as opposed to random. This allows maximum use of hardware-based performance features such as caching and automatic pre-fetching.

In VORPAL, parallelism through explicit message passing is done by means of domain decomposition. Each section of the domain is updated by its respective processor, and synchronized at the end of each time step. Particles residing in a particular section are stored on the processor updating that section. Therefore, update time per processor depends on the changing concentration of plasma during the course of a simulation. Due to this movement of particles, load will vary during the course of a simulation. Section 5 discusses dynamic load balancing, which aims to compensate for this by adjusting the decomposition at runtime thus keeping utilization high across all processors.

To maintain flexibility, VORPAL allows users to enter arithmetic expressions in a configuration file that control various aspects of a simulation. As a result, evaluation of such expressions can be quite costly. Traditionally, VORPAL employs a custom expression parser and evaluator to perform this task. The result is an unnecessarily complex and inefficient system used simply to preserve a high degree of flexibility. Runtime code compilation offers the best of both worlds by giving VORPAL the ability to turn text source code into objects at runtime. Thus efficiency of hard coded expressions is preserved while maintaining increased flexibility.

Scientific applications are generally developed and executed in an environment with abundant computational resources. Parallel processing is only used during execution of an application. Compilation is usually done serially. Since most scientific applications are home grown, development time is a significant overhead. A parallel build tool aims to reduce development time by utilizing available parallel resources to cut back on build time. Our tool cuts the build time a fully optimized VORPAL executable by 83% when built on a ten node cluster.
Due to these and other computational challenges involving PIC, much research has been done in this area. Previous optimization-related work tends to be divided between parallel strategies, which focus on high level implementation to minimize latency, and serial optimization strategies which tend to be more low level and processor specific. Viktor Decyk[5] has illustrated some general optimization strategies for numerical computation including branch minimization, and data structure consolidation. He has also addressed the benefits of particle sorting. Kevin Bowers[3] has implemented a linear time counting sorted for PIC.
Chapter 2

Introduction to VORPAL

VORPAL (Versatile Object-oriented Plasma with Lasers) is a C++ simulation tool developed at the University of Colorado. The project’s aim is to develop next generation particle accelerators that are far more compact than present day solutions. Similar codes such as OOPIC, and WARP have also been developed to fulfill similar purposes. VORPAL stands out primarily due to its dimension freedom through use of recursive templates. Recursive templates enables VORPAL to run one, two, or three dimensional simulations simply be instantiating the required dimensionality. In this manner, a single section of code is compiled for a specified number of dimensions, preserving performance while keeping good object oriented design.

2.1 VORPAL PIC structure

VORPAL is a fully object oriented application. Extensive use of class structures maximizes flexibility. Simulations are spawned at runtime by instantiating classes which represent physical components such as EM fields, particles, fluids, sinks and sources. For the purposes of this paper we describe only the EM and particle classes here since they are the most relevant to the optimizations presented. The EM object simulates electromagnetic fields, providing information to other objects about electromagnetic
forces in the physical grid. A number of implementations exist, allowing the user to simulate electrostatics, electrodynamics, and constant potential fields. The grid EM field is discussed in this paper since it is the most widely used kind. Particles are used to simulate position and momentum of freely charged particles (plasma). The EM and particles objects depend on each other since the electric and magnetic fields affect particle motion, and particle motion generates EM waves. When performing optimizations, the advancing of particles is of most interest since it takes the majority of the processing time when compared to the EM update time.

In VORPALs implementation, particles are stored in groups of C++ standard template library (STL) vectors. Groups are updated by feeding them though various updater objects. Each object performs some operation on that group. These operations include position to grid translation, particle acceleration, particle movement, current weighing, charge weighing, and boundary checking. During these operations, data from other objects must be retrieved. For example, electric and magnetic field values are obtained from the EM object, which consults individual electric and magnetic field objects to interpolate the field values for a set of positions. The interconnection between grid fields and particles provides the opportunity for optimization through particle sorting, discussed in the following chapter.
Chapter 3

Particle Sorting

The particle update process involves changing each particle’s velocity according to the EM values at its position. EM field data is stored on a grid, addressed with integer indices that reference field values. Particle positions are stored with floating precision in a series of arrays and addressed serially. Thus each particle’s position references EM field values on a grid to update itself. We can improve the performance of this process by ordering the particles such that access to EM field values occurs sequentially in memory as particles are updated as shown in fig. 3.1. Doing so maximizes temporal locality since use of data in cache lines is maximized.

The sorting problem is approached from two angles: the sorting algorithm itself and the cost optimization algorithm. The sorting algorithm consists of an interface to the C++ standard template library (STL) sorting implementation. This technique allows VORPAL to harness the highly optimized STL implementation. Details of this implementation is discussed in section 3.1. Cost optimization dynamically measures update and sorting time to determine the optimal points to perform the sort. This is explained in section ???. The combined particle sorter can double or even triple overall performance. More detailed performance analysis of particle sorting is discussed in section 3.3.
3.1 Sorting through an STL iterator interface

VORPAL takes a unique approach to the sorting problem. Through use of advanced C++ techniques, we have created an STL compatible iterator for the parallel segmented array data structure used to hold particles. Alternative solutions, such as creating a custom sorting algorithm for the particles, were disregarded due to the increased performance the STL implementation can provide. This Section focuses on the details and tradeoffs involved in creating the C++ iterator. Section 3.1.1 begins by setting the framework to justify use of interfaces to pre-existing algorithms over implementation of custom algorithms. Section 3.1.2 provides an overview of the data structure used for storing particles. This consists of parallel segmented arrays and is preferred for performance reasons. As this structure does not lend itself well to random access iteration, fundamental challenges in this context are introduced. Implementation details of the iterator itself are divided between iteration algorithms, discussed in sec. 3.1.3 and dereferencing strategies, discussed in sec. 3.1.4. Iteration algorithms include the sequential access operators of pre/post increment and decrement as well as the ran-
dom access assignment operator. The two are treated separately since sequential access can be optimized to independently of random access. Sequential access can be done in constant time whereas random access is performed in $O(n \log_2 n)$ time. Dereferencing strategies cover the techniques used to return by value (Rvalue) and return by reference (Lvalue) to an element spread over multiple parallel arrays. Sec. 3.1.5 provides a performance evaluation of the data structure using the STL sorting algorithm. We compare a custom implementation with the STL implementation in an attempt to understand the performance differences. Sec. 3.1.5 also includes an analysis of the performance overhead involved in our iterator as compared to an iterator for a linear data structure such as an array or linked list.

3.1.1 A case for Interfacing of custom implementation

The C++ standard template library (STL) provides highly optimized code to perform a variety of basic tasks used in software applications. Use of these algorithms is preferred over custom implementations since they are generally faster and more reliable. Often times, however, an STL algorithm exists for a specific problem but not for a particular data structure. All too common is the desire to write custom implementations for required operations. Doing so, however, typically results in less reliable and less efficient code. When a suitable interface is created, such as an iterator for a given data structure, the door is immediately opened to a whole host of pre-written algorithms that require that interface as input to perform a wide variety of operations. When taking the custom implementation approach, one must regurgitate and adequately test a new algorithm for each operation desired. As such, adequate custom implementations typically require extensive development and test time. This can be avoided through the code reuse implied by the interface design approach. Furthermore, STL algorithms are generally optimized for the given compiler and architecture to a degree that custom
implementations typically do not achieve. For these reasons we prefer to use native library code whenever possible.

3.1.2 Data Structure

Segmented parallel arrays follow the design pattern illustrated in figure 3.2. A

![Figure 3.2: Parallel Segmented Arrays](image)

pointer array contains references to equal sized parallel arrays. These parallel arrays, while having identical size with respect to one another, have variable size with respect to the other referenced parallel arrays. The first indexed value is the set of data values in the first index of each of the first referenced parallel arrays. The second value is the second index in this set of parallel arrays, or the first value in the next set of parallel arrays if the first set has no more elements.
### 3.1.2.1 Advantages of Parallel Segmented Arrays

Before we discuss a random access iteration strategy, we present a few brief advantages to this type of structure. In this context, we treat segmentation and parallel arrays separately, discussing the advantages of each. Two principal advantages exist to segmentation: efficient updates and minimal insertion time. Efficient updates in the context of scientific computing applications such as VORPAL means that the data can be passed to updater objects within the application without suffering excessive virtual function call overhead or unnecessary cache misses. Since an update is done for an entire segment at a time, only one virtual function call need be made for the entire segment. Cache performance is also maximized since segments are sized so as not to overflow the cache. In this way, as a segment gets passed from one updater to another, it remains in the cache. After all updates have completed for a segment, the process is repeated for the next until all segments have been updated. Secondly, insertion and deletion of items within the structure requires only the resize of either a single segment or the array of segment pointers. Updaters that add or remove elements can therefore do so with minimal impact on performance.

Arrays are the preferred data structure for each segment over more dynamic structures such as linked lists due to their lack of per-element metadata and perfect iteration efficiency. Furthermore, the sequential access they offer makes good use of any caching facilities available to the executing machine. Parallel arrays are used over a single array of aggregate structures to minimize cache stride during updates. In each of the parallel arrays data is stored sequentially in memory. Therefore iteration over any given array requires access of sequential memory addresses. Updaters will typically perform some action on just one array. This makes good use of cache performance since stride is kept at a minimum.
3.1.2.2 Random access to parallel, segmented arrays with a C++ iterator

For a number of algorithms it is highly desirable to have random access functionality to a container. Sorting and searching are just a few. Any algorithm over a dataset that operates in less than $O(n)$ time will require random access capability. We address the need for random access at the source with a custom iterator. Alternatively, one could adapt the necessary set of algorithms to suit the nature of the container. We consider the former approach to be less efficient and less reliable.

Construction of a random access iterator presents two major challenges. First, the data structure inherently supports random access only within each segment. As a whole, only sequential access is supported. Second is the problem of providing suitable return types for the dereference operator “*”. Recall that parallel arrays share an implicit but not actual data type consisting of the set of data values where the indices are equal. Both return by reference and return by value must be supported. The former presents a significant challenge, since we must find a way to return a single handle, or reference to something that doesn’t exist.

To achieve compatibility with pre-existing STL algorithms, our iterator must conform to STL standards. In particular this implies that the following conditions are met.

- Iterator must have a single value-type. This is returned either by reference or value through the dereference and index operators.

- Iterator must have a distance-type. This is a metric indicating the degree of separation between two iterators. It is returned by the difference operator and used by increment and decrement operators.

- All comparison operators must be defined. Specifically, the operators for $<$, $<=$, $>$, $>$=, and $==$ should be defined with appropriate meaning.
• Index and dereference operators must be defined. These must return the value-type by both reference and value.

• Increment and decrement operators taking distance-type must be defined. This provides for the ability to increment and decrement by units greater than one.

• Difference operator returning a distance-type must be defined. This provides the number of unit increments or decrements needed to make one iterator equal to another.

• Assignment operator taking a distance-type must be defined. Similarly to the index operator, this allows the iterator to be randomly positioned.

The general format for a C++ random access iterator is as follows:

```cpp
template<class value_type, class distance_type = size_t>
class MyIterator : public std::iterator<std::random_access_iterator_tag,
    value_type, distance_type> {
    // All necessary operators overloaded here.
};
```

Inheriting from std::iterator with proper template arguments properly defines the necessary characteristics of the iterator [11]. These are the iterator type (random access in this case), the value-type, the difference-type, the pointer type (defaults to value_type*), and the reference type (defaults to value_type&). Among these, those of concern are the value-type and distance-type. The value-type is returned by any kind of dereferencing operator such as *, −>, and []. The distance-type is given as a parameter to all random indexing operators: [], +=, −=, and =. It is also returned by the difference operator. A class conforming to these standards satisfies the STL requirements of a random access
iterator. In the following sections we focus on the nontrivial aspects of implementing this type of iterator for segmented parallel arrays: indexing and value-type.

3.1.3 Iteration and Indexing

We define three “location” variables to navigate the iterator through the data space: index, gindex, and offset. The index refers to the overall element number in the data set. The first index is 0 and the last index is equal to the number of elements in the data structure. The group index, gindex, refers to the current segment consisting of two or more parallel arrays. This value ranges from 0 to the last index of the pointer array. The offset refers to the current index within the current segment, it ranges from 0 to the last element in the current segment. These values are manipulated in unison to keep the iterator in a valid state. The random access iterator must perform efficiently for unit increments, but also support random access to a structure not organized in a sequential manner. Two types of iteration are separately handled and will be discussed here: sequential access and random access. Sequential access to a data array denotes the accessing of data in some order inherent to the array. Implementation details of sequential access operations are covered in sec. 3.1.3.1. Random access denotes the capability to access an $i^{th}$ element though an index. Random access is covered in sec. 3.1.3.2. These two methods are treated separately since sequential access can be optimized to outperform random access.

3.1.3.1 Sequential Access

Sequential access is composed of only increment and decrement operations which we discuss here. These both run in constant time, $O(ln(n))$. Unit increments and
decrements are achieved by bumping the index, gindex, and offset in a synchronous fashion. Algorithm 3.1.1 illustrates the increment and decrement operations. In either case the first operation is to adjust the global index and check for the upper bound condition in which case we cannot query for current array size. If the upper bound condition is not met then the algorithm sequentially bumps across the segments until one is found with a nonzero size. The iterator then assumes this as the current value.
Algorithm 3.1.1: increment(void)

procedure operator++()

index ← index + 1

offset ← offset + 1

if isUpperBoundValid()

then return (void)

while offset = data[gindex].size()

do

\begin{align*}
& \quad \text{gindex} ← \text{gindex} + 1 \quad \text{return (void)} \\
& \quad \text{offset} ← 0
\end{align*}

procedure decrement()

index ← index − 1

while offset = 0

do

\begin{align*}
& \quad \text{gindex} ← \text{gindex} − 1 \quad \text{offset} ← \text{offset} − 1 \\
& \quad \text{offset} ← \text{data}[\text{gindex}].\text{size()}
\end{align*}

procedure isUpperBoundValid()

if index >= totalElements

then \begin{align*}
& \quad \text{offset} ← 0 \\
& \quad \text{gindex} ← \text{data}.\text{size()} \quad \text{return (false)} \\
& \quad \text{return (true)}
\end{align*}

3.1.3.2 Random Access

Location of the $i^{th}$ element within the structure presents somewhat of a challenge since the segmented arrays may vary in size. To optimize the search, an indexing array
is kept up to date which holds a running sum of the number of data elements in the current and all previous arrays. Formally,

\[ R_i = \sum_{j=0}^{i} D_j \]

where \( R_i \) is the indexing array and \( D_j \) is the size of the \( j^{th} \) array. The \( n^{th} \) element in the structure can then be located with a simple binary search for the lower bound of the desired index where the lower bound is defined as the first value greater than or equal to the key. This effectively yields the index of the array where the \( n^{th} \) element is stored. The offset into this array is then trivially computed. Algorithm 3.1.2 computes the index.

**Algorithm 3.1.2: Determine location of index**(void)

**procedure** reindex()

  if isUpperBoundValid()\textbf{return} (void)

  \( gindex \leftarrow \text{lower bound}(\text{index} + 1) \)

  \( i \leftarrow R_{gindex} - D_{gindex} \)

  \( \text{offset} \leftarrow \text{index} - i \)

### 3.1.4 Returning references and values

The second essential function of an iterator is to provide read and write access to the data elements over which the iterator is iterating. In C++ this is accomplished with the dereference operator ('*'), the indirection operator ('->'), and the index operator ('[]'). In the context of this paper we consider only the dereference operator ('*') since the indirection and index operators trivially follow from the same concepts outlined
here. Implementation of the iterator’s dereference, copy, and assignment operators is
given below.

class iterator {

public:

    DataHolder operator*() const {
        DataHolder ret;
        ret.tA=arrayA[gindex][offset];
        ret.tB=arrayB[gindex][offset];
        ret.tC=arrayC[gindex][offset];
        return ret;
    }

    DataHolder& operator*() {
        ref.A=&(arrayA[gindex][offset]);
        ref.B=&(arrayB[gindex][offset]);
        ref.C=&(arrayC[gindex][offset]);
        return ref;
    }

    iterator operator=(const iterator& x) {
        // Copy all members except ref
        return *this;
    }

    DataHolder(const DataHolder& x) {
        // Copy all members except ref
    }

    DataHolder ref;
}
The data value at an index is distributed across parallel arrays, however STL algorithms need a single value_type to move data elements from one location to another. This value_type must be provided as both a reference and value so it can be used to read, make comparisons, and overwrite elements. These tasks are accomplished with a special DataHolder class defined as follows.

```cpp
class DataHolder {
    public:
        DataHolder() {
            A=&tA;
            B=&tB;
            C=&tC;
        }
        DataHolder operator=(const DataHolder& x) {
            *A=*x.A;
            *B=*x.B;
            *C=*x.C;
        }
        bool operator<(const DataHolder& x) {
            return A<x.A;
        }

    TYPE* A;
};
```
TYPE* B;
TYPE* C;

TYPE tA;
TYPE tB;
TYPE tC;
};

This class possesses the ability to behave both as a value and a reference. When a non-const iterator object is used the non-const dereference operator is used to return a reference to the value at the current location. The iterator instead returns a reference to a DataHolder member of the iterator with the pointer values setup to point to the actual values in the parallel arrays. If this object is used as an rvalue in an assignment or initialization to another DataHolder as shown below the assignment operator or copy constructor will be called respectively.

DataHolder& y=*iter; // y is a reference to actual data
DataHolder x=y;       // x now holds a copy of values referenced by y

From the code for the assignment operator and copy constructor we see that the result is to copy the values from the reference to the members of the given DataHolder instance. The assigned value effectively holds a copy of the data referenced by the y. In this context y behaves as a reference to actual data though it is in itself not and x behaves as a value holding a copy of actual data. The assignment and copy constructors within the DataHolder make this possible.

When a const iterator object is used the const dereference operator is called. This function returns a DataHolder that contains copies of, not references to, the data values.
This is necessary since an optimization allowed by the ANSI C++ standard permits the following initialization without an assignment operator or copy constructor being called provided the return value is not a reference:

```cpp
DataHolder temp=*iter;
```

In this case the address of temp on the stack is the same address where the DataHolder was initially constructed in the iterator dereference operator, and thus the copy constructor is not called. In the case, if *iter returns a DataHolder behaving as a reference, temp will also effectively behave as a reference. We overcome this problem by returning the DataHolder by reference in which case either the assignment or copy constructor will always be called when an assignment or initialization is made. Since a return by reference has no existing value of DataHolder to return a reference to, we include a DataHolder data member in each instance of the iterator class and return a reference to it. In this way return by reference can be made to a named variable who’s lifetime exceeds that of the dereference operator.

We must also preserve the integrity of any value returned by reference even in the face of subsequent changes to the iterator. The following code fragment,

```cpp
DataHolder& a=*iter1;
iter1=iter2;
a=*iter3; //error, a could now refer to iter2’s value
```

illustrates a potential problem. We overcome this problem by overloading the iterator’s assignment operator (‘=’) and copy constructor (’(const DataHolder&)’) to copy all members except the DataHolder.
A caveat of this overall approach still to returning by reference still remains. The lifetime of any returned reference must meet but cannot exceed the lifetime of the iterator that returns the reference. For example, the following legitimate code sequence may fail:

```cpp
DataHolder& getIndex(size_t x) {
    iterator ret;
    ret=x;
    return *ret;
}
```

We have found no solution to this problem. Fortunately, we have also not found any code in the STL which exposes it.

3.1.5 Performance

Performance considerations include both an experimental and a mathematical running time analysis. As mentioned in sec. ??, performance is one reason to prefer an STL implementation over a custom implementation. Sec. 3.1.5.1 experimentally compares sorting times using both STL and custom sorting implementations to justify this statement. Of equal concern is a mathematical understanding of the running time as a function of input size, also known as the bigO time. Sec. 3.1.5.2 presents a bigO time for the STL implementation using our new iterator.
3.1.5.1 STL versus custom sorting algorithms

The custom qsort implementation uses a simple textbook quicksort [4] algorithm. With a custom algorithm, swapping of the multiple elements in a parallel array can be implemented simply as part of the algorithm. As a result the DataHolder class is no longer necessary. As seen in table 3.1.5.1 for the randomized array, the added overhead of DataHolders is observed. Recall that a randomized array provides the best case scenario for quicksort since pivots are more likely to divide the subarray into relatively balanced proportions. The STL sort on the other hand is far more complex, making use of numerous sorting algorithms including insertion sort, quicksort, and heap sort [10]. The STL implementation monitors the progress of the sort and changes algorithms depending on its progress. Additionally, when the quicksort algorithm is applied, special consideration is given to choosing a suitable pivot so as not to pathologically partition the data. For the in-order array, the STL implementation holds a decent time whereas the quicksort gives horrible performance as it is the worst case scenario.

Note that surpassing an STL implementation is not a primary concern. Certainly, for a specific problem a specific solution can always be made to outperform a general solution. Typically, however, a programmer does not have resources available to achieve this and a makeshift solution is quickly applied instead. The performance impact of this will usually be worse than more general solutions.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Randomized Array</th>
<th>Sequential Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>STL</td>
<td>1</td>
<td>0.39</td>
</tr>
<tr>
<td>custom qsort</td>
<td>0.87</td>
<td>232.56</td>
</tr>
</tbody>
</table>

Table 3.1: Relative sorting times of random and sequential data arrays
3.1.5.2 BigO Running time

Reindexing is of particular concern since it is the only non-constant time operation in the iterator. Its use may affect the overall running time of a sorting operation. Figure 3.3 illustrates the frequency of calls to reindex made necessarily as a result of operations performed by the STL sort. The frequency increases roughly linearly with time. A reindexing can be done in $O(ln(n))$ time. We therefore add a factor of $ln(n)$ to the average case performance yielding a running time of $O(n * ln(n) * ln(n))$.

3.2 Dynamic cost optimization

Due to the overhead involved, particle sorting must be implemented with special care so as not to worsen performance. A decent sorting algorithm will have an average running time of $O(n * ln(n))$ where $n$ is the length of the data set or number of particles in our case. Note that the particle update itself has a fixed run time of only $O(n)$.

Sorting should only be done when its performance impact outweighs the cost required to perform the sort. To this end VORPAL uses a cost minimization algorithm to control when a sort should be performed based on the following criteria:

- Last time, $tSort$, it took to sort, per particle.
- Last time, $tSorted$, that it took to push the particles after sorting, per particle.
- Total time, $tSumPush$, that it took to push the particles since last sort, per particle.

\[1\] Kevin Bowers has implemented a linear $O(n)$ sorting algorithm for PIC based on a counting sort, however the algorithm has memory requirements proportional to the number of particles [3]
• Number of steps, $n_{SinceSort}$, since the last sort.

Whenever

$$t_{SumPush} > n_{SinceSort} \cdot t_{Sorted} + t_{Sort}$$

then it is time to sort again, since the time invested in the sort will be recovered in the next $n_{SinceSort}$ steps. This result can be derived under the assumption that the update time increases linearly with increasing timestep $y = mx$ where $m$ is the slope constant, $x$ is the step, and $y$ is the excessive update time for the step. In other words, the difference between the total update time and the update time with particles perfectly sorted. We further assume that sorting takes a constant amount of time $p$. We can see from figure 3.4 that sorting is best done when the total time spent sorting is equal to the total time spent doing extra processing as a result of not having particles sorted. If we deviate by sorting too soon or too late, we can see that the total area under the curve show in figure 3.4 will increase. So we are looking for a value $x$ where $p = \frac{1}{2}m \cdot x^2$ or $x = \sqrt{\frac{2p}{m}}$. In the code this condition is satisfied whenever

$$t_{SumPush} > n_{SinceSort} \cdot t_{Sorted} + t_{Sort}$$

### 3.3 Sorting Performance

The performance impact of particle sorting varies widely with processor type and simulation parameters. Since particle sorting aims to improve performance by minimizing cache misses, the impact is best seen on machines with small cache sizes relative to the number of particles in the simulation. The results we present here are for 500 time steps of a 2-D simulation with 1800x100 cells. Full Yee EM solve and relativistic boris push were used. Five particles per cell were loaded at the start of the
simulation. The table below shows average particle push time in micro-sections per particle.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Push time with sorting</th>
<th>Push time without sorting</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Athlon 2.8GHz</td>
<td>0.534</td>
<td>1.579</td>
<td>3</td>
</tr>
<tr>
<td>IBM SP</td>
<td>3.769</td>
<td>4.66</td>
<td>1.24</td>
</tr>
<tr>
<td>G4 800MHz</td>
<td>2.88</td>
<td>4.45</td>
<td>1.55</td>
</tr>
<tr>
<td>Intel P4 2.4GHz</td>
<td>1.169</td>
<td>2.813</td>
<td>2.4</td>
</tr>
</tbody>
</table>
Figure 3.3: Re-Indexing performance

Figure 3.4: Optimal sorting step makes sorting time equal to excess particle update time
Chapter 4

Dynamic Compilation

Many variables and otherwise parameters for a simulation package such as VORPAL are not available at runtime. Some of these parameters include expressions and otherwise customized directions for handling of specific features. Instructions on customized handling of features provided in an input file must typically be interpreted by some makeshift interpreter that is slow and inflexible. In this chapter we provide an object oriented method allowing specific user level configurations to be provided at execution time in the form of C++ code. This strategy is currently used in VORPAL for evaluation of expressions provided within a simulation configuration file. Opposed to interpreted solutions, this technique is highly flexible and very fast. Sec. 4.1 begins with a user-level overview of the dynamic compiler. Details on how programmers might incorporate dynamic compilation is also discussed. The design pattern and implementation details are explained in sec. 4.2. A performance analysis of runtime compilation compared to runtime interpretation is provided in sec. 4.3.

4.1 System Usage

We begin with a brief user level overview. Our solution has been implemented as a C++ factory class called ClassGen. ClassGen is capable of returning objects of a
class given C++ source code for the class. A program making use of ClassGen will use it to generate implementations, or subclasses of some base (abstract) class. These will be used by a consumer seeking an implementation for a specific interface. The following example code below illustrates a typical case.

class baseInterface {

public:

    virtual void someFunc() = 0;

};

void consumer(baseInterface& x) {

};

void factory( ) {

    ClassGen<baseInterface> cg;

    cg.compile(codeblock, classname);

    baseInterface* x=cg.getInstance();

    consumer(*x);

}

Notice that ClassGen is templated over the base class for which it will generate subclasses. A call to the compile() member of ClassGen prepares the class to return any number of instances of the given implementation. The implementation text is passed in the first argument (codeblock) as a string and the desired instantiation is passed as the second argument. For example a configuration file for some program may provide two arguments for a dynamic class, a filename where code is kept and an instantiation line:

    file = custom.cpp
instantiate = custom<float,2>

This would instruct the factory to read code from custom.cpp and instruct ClassGen to prepare to return instantiations of class “custom” with template arguments <float,2> upon successive calls to ClassGen::getInstance().

4.2 Design Pattern

At the heart of the system is the ClassGen class which takes a string of C++ code and produces instances of a class defined within that code. This instance can then be passed off to any component that uses features defined in its abstract base class. The object oriented design for this system is illustrated in figure 4.1. Sec. 4.2.1 begins with an introduction to shared libraries and dynamic loading using the dlfcn library. We then present a strategy to use dlfcn to instantiate C++ objects in sec 4.2.2. Sec. 4.2.3 illustrates a file management scheme to ensure uniqueness among temporary file names used to pass information between the system compiler and dlfcn. Sec. explains how files are written and compiled at runtime. Techniques used to make maximize platform independence using GNU autoconf are also described here. After compilation, use of dlfcn to create and destroy class instances is explained in sections 4.2.5 and 4.2.6.

4.2.1 Shared libraries and dynamic execution

Shared object libraries consist of code that can be loaded and executed at runtime by one or more programs. In contrast to code generated for static libraries, shared library code is generated by the compiler such that it’s execution does not depend on where the code is loaded in memory. This implies that all intentions to use absolute memory addressing must be specially handled, resulting in a significant performance
4.2.2 Implementation of a platform independent dynamic compiler

We focus now on implementation details of the ClassGen class. The process of converting code in text form to objects is given below and detailed in the following sections.

1. We begin with the addition of helper functions to the class text to circumvent C++ name mangling issues.

2. We address concurrency issues involving multiple instances of ClassGen as well as multiple instances of the same application running in parallel.

3. The code is passed to the system compiler to generate a shared library. Platform dependent issues are addressed.

4. The resulting shared library is loaded into memory.

5. Instantiations are returned to calling objects and unused objects are deleted.

4.2.2.1 Adding helper functions to create and destroy objects.

Since dlfcn does not internally capable of performing the name mangling necessary to load class constructors, we append “helper” functions to the existing code which interface to the class text in that code. Shown below is an example helper function that returns a new instance the the class.

```c
extern "C" void* getInstance() { return new className; }
```

Some architectures allow generation of absolute references in shared library code, giving the dynamic loader the responsibility to fill in these references at runtime.
When compiled, getInstance contains assembly level code to generate an instance of className and return a pointer to it. The extern “C” directive instructs the C++ compiler not to perform any name mangling on the getInstance label. Typically name mangling is done to support class member functions and function overloading. A similar helper function is also issued to destroy the instance.

```c
extern "C" void delInstance(className* x) { delete x; }
```

### 4.2.3 Concurrency Issues in file management

Since ClassGen relies on the filesystem and externally on a system compiler to perform much of its work, two concurrency related issues must be addressed: multiple ClassGen instances and multiple process instances. We address each of these issues in detail below.

#### 4.2.3.1 Generating a Hash

Multiple instances of ClassGen may be used to dynamically compile and instantiate different objects. These instances must not conflict with each other by writing code and object libraries to the same files. At the same time we do not want to give ClassGen any sense of other instances of itself that may be running as this unnecessarily increases complexity and is generally poor OO design. This issue is handled by generating a unique hash for each file to be compiled using the file’s contents. This ensures that the system will be free from filename conflicts.
4.2.3.2 Locking for multiple process instances

Since our technique is implemented on parallel scientific computing software, we must address redundancy issues that exist and may result in file conflicts. Specifically, it is necessary to have a scheme to prevent multiple pieces of the same job to conflict with one another when generating shared libraries. In scientific computing applications, for example, a single job will be divided among many processors sharing a common filesystem. Each of these jobs will perform the same operations on different pieces of the problem. As such, each process will need to have access to shared libraries containing dynamic classes. It is therefore necessary to employ a scheme to prevent multiple processes from conflicting with one another over the same files. One possible solution is to mangle all the filenames written to disk by tagging each with a unique ID for each job executing. Aside from the unnecessary replication of work and massive file cluttering this would case, it also requires ClassGen to have knowledge about it’s parallel operating environment. As part of good OO programming practice, we aim to minimize interfaces between components, so not giving ClassGen any knowledge about its operating environment is preferable. To solve the problem we employ a simple locking scheme using the C open function to create a lock file as soon as an instance of ClassGen has entered the compile stage. If an instance gains a lock, it proceeds with the compilation process, else it polls until the lockfile is removed at which point it assumes the necessary library has been created and skips the compile step, proceeding directly to load the resulting dynamic library. Pseudo-code below summarizes the process.


```cpp
ClassGen::compile() {
    if (getLock(hash)) {
        compile;
    } else while(!getLock(hash)); //poll until lock released
```
releaseLock(hash);
load library;
}

4.2.4 Writing and compiling

With helper functions attached, and concurrency issues adequately addressed, writing out a C++ file and issuing a call to the compiler is a straightforward matter. A typical call might look like as follows:

```cpp
string cmd=CXX + " " + OPTI_ARGS + " " + DYNLIB_ARGS + " " + INC_ARGS + "-o" outfile
system(cmd);
```

Notice the native compiler name and path, as well as command line arguments are given as macro variables via autoconf. This helps to increase platform independence. During configuration, the appropriate compiler is filled into the CXX variable. Depending on the desired level of optimization for the given configuration and platform, OPTI_ARGS will be filled correctly. Different compilers may have different arguments to create a shared library so DYNLIB_ARGS contains these for the given compiler. Finally, since we will most likely be using ClassGen to instantiate subclasses of existing classes, we add the include path via INC_ARGS to the existing source code so that these subclasses can find base class interfaces.

4.2.5 Loading the shared library

Once compiled, loading shared libraries is a trivial matter. As shown below, a call is made to dlopen and a pointer to the generator helper function is acquired.
class ClassGen {
    typedef void* (*newinst)(void);
    typedef void (*delinst)(void*);

    void compile(...) {
        ...
        void *lib_handle = dlopen(filename);
        creator=(newinst)dlsym(lib_handle, "getInstance");
        destructor=(delinst)dlsym(lib_handle, "delInstance");
    }
};

In this example getInstance and delinstance are the helper functions that were originally embedded within the code. They can be addressed here by these names since they were declared extern "C" before compilation. The variables “creator” and “destructor” are members of ClassGen and are later used when returning new instances of the compiled class.

4.2.6 Creating and destroying Instances

With creator and destructor function pointers set, returning new instances and deleting old ones is a trivial matter.

class ClassGen {
    ...
    T* getInstance() { return (T*)creator(); }
    void delInstance(T* x) { destructor(x); }
}
We include a destructor as well as a creator for consistency purposes. The creator is required since locating the constructor for a given class via dlfcn would otherwise be overly complex. If one were to use delete as opposed to calling delInstance() then potentially overloaded new and delete operators would mismatch. The default new operator would be use for creating instances but the overloaded delete operator would be used for destroying them.\(^2\)

4.3 Performance analysis of dynamic compilation

Although performance of this strategy is far greater than solutions based on runtime interpretation, a few drawbacks exist. Code for shared libraries must usually be position independent, allowing it to produce the same results regardless of where it is loaded into memory. Occasionally this means that generated code must include instructions to locate its location in memory. The performance impact of this overhead is significant.

Secondly, all class member functions will be executed as virtual functions. This is necessary since the locations of these functions is not known until runtime. Therefore the usual performance issues surrounding virtual function calls must be considered.

In spite of all this, the pros greatly outweigh the cons when use of ClassGen is compared to interpreted solutions. We present here a simple case study comparing performance differences in expression evaluation time as a function of expression size for both methods. A typical implementation of ClassGen is used to generate the compiled evaluator and a tree-based expression parser/evaluator generates the interpreted evaluator. The following seven non-reducible polynomials were evaluated using each method and the

\(^2\) http://www.isotton.com/howtos/C++-dlopen-mini-HOWTO/C++-dlopen-mini-HOWTO.html
Case 1: \( x \)

Case 2: \( x + 1 \)

Case 3: \( 2 \cdot x^2 + x + 1 \)

Case 4: \( 3 \cdot x^3 + 2 \cdot x^2 + x + 1 \)

Case 5: \( 4 \cdot x^4 + 3 \cdot x^3 + 2 \cdot x^2 + x + 1 \)

Case 6: \( 5 \cdot x^5 + 4 \cdot x^4 + 3 \cdot x^3 + 2 \cdot x^2 + x + 1 \)

Case 7: \( 6 \cdot x^6 + 5 \cdot x^5 + 4 \cdot x^4 + 3 \cdot x^3 + 2 \cdot x^2 + x + 1 \)

Figure 4.2 compares the evaluation time for each of the seven cases. There is no competition between the two methods, for any size of polynomial the compiled expression evaluates orders of magnitude faster than the interpreted expression.
Figure 4.1: Dynamic class design pattern
Figure 4.2: Compilation vs. Interpretation
Chapter 5

Dynamic Load Balancing

A significant benchmark for any parallel application is the measure of its parallel resource utilization. Under-utilization is caused by bottlenecks that can drastically reduce performance. For particle in cell (PIC) codes, synchronization at the end of each time-step is the limiting factor to achieving maximum utilization. Processors lagging behind are bottlenecks, forcing others to remain idle until bottlenecks can complete their update cycle. A good load distribution will ideally make the range of update times zero. The situation grows more complex, however, if the load distribution changes in an unpredictable way as the simulation progresses. PIC codes exhibit this characteristic since particle distribution is continuously changing and processor utilization is a function of particle distribution. Since particle distribution changes at each time-step, it is not possible to set a single ideal decomposition for the simulation. A more dynamic approach is needed. We describe here a dynamic load balancing algorithm for VORPAL. Our method is based on load distribution through decomposition adjustment. Similar techniques have been used on other codes [6]. Sec. 5.0.1 sets the stage with a brief overview of VORPAL’s internal framework with respect to aspects that relate to load balancing. Readers interested in a more complete description of VORPAL’s workings are referred to a more thorough explanation of the VORPAL framework [9]. Sec. 5.0.2 describes the load balancing implementation. In Sec. 5.0.3 we demonstrate
load balancing functionality and its performance in a case study.

5.0.1 Parallel Framework

Parallel processing under VORPAL is accomplished through a global grid decomposition. The global grid is divided among available nodes either automatically or according to a user-provided description. Internally, a decomposition is described in VORPAL through a list of boundary positions for each direction. The initial (zero) direction divides the entire grid any number of times. Each of the resulting slices can then be further divided independently along direction one. Finally, each slice along direction one can be independently divided along direction two. This concept is illustrated in fig. 5.1.

Figure 5.1: Boundary adjustment is limited by direction. Direction zero divides the entire domain. Direction one boundaries divide individual divisions made by direction zero boundaries. Direction two boundaries divide individual slices made by direction one.

Such a decomposition places some restrictions on how load balancing can be performed. We see from fig. 5.1 that adjustments in a boundary on direction zero result in all slices on either side of that boundary changing size.
Synchronization over MPI is handled through a series of objects that automatically synchronize local grid boundaries with neighboring processors. This makes the addition of load balancing convenient, since following an adjustment synchronization is performed automatically. However, this also limits the adjustment of each boundary to no more than one unit at a time.

5.0.2 Implementation of a dynamic load balancer for VORPAL

The load balancing process is divided into six steps.

1. Update time for the last step on each processor is measured.
2. From the load times, a decision is made on whether to perform load balancing.
3. A decomposition adjustment is determined.
4. The adjustment is broadcast to all nodes and applied at each node.
5. Data structures holding grid values are resized to accommodate new capacity requirements.
6. A final synchronization step fills in missing boundary grid data and particles.

Using the system clock, update time for each step is measured independently on each node. At the end of the time-step these update times are reported to the root node. The root collects and sorts the update times. Since a certain level of overhead is associated with performing a decomposition adjustment, a cost minimization algorithm is used to decide whether or not to perform an adjustment. VORPAL performs an adjustment only if it is estimated to reduce the total execution time. The decision process is made as follows. Given
• time difference, $cPtimeDiff$, between update times of most and least busiest processors,

• change in $cPtimeDiff$, $lastDLBsavings$, as result of last decomposition adjustment,

• remaining number, $numRemSteps$, of steps to take for this simulation,

• last time, $lastDLBtime$, it took to adjust the decomposition,

• an experimentally chosen error margin, $Merror$, to avoid unnecessary adjustments.

A decomposition adjustment is made if the following condition is met.

$$lastDLBsavings \times numRemSteps > lastDLBtime + Merror$$

If the decision is made to perform an adjustment, step three determines the boundaries to adjust. Since total update time per step is limited by the bottleneck, our adjustment strategy uses a greedy algorithm to reduce its workload. The decomposition is adjusted by shrinking boundaries around all sides of the bottleneck node first. The second busiest processor then shrinks all of its boundaries, provided that they have not already been adjusted by busier processors. This is done for all processors in the simulation. The result is a decomposition adjustment description object as shown in fig. 5.2 that is broadcast and applied on all processors as indicated in step four. The application process is simple; a new decomposition object replaces the old one. The automatic effect is that subsequent time-steps will update data over differently sized grids.

To accommodate newly sized grids, data structures on the grid must be resized accordingly. The algorithm, however, must be flexible so as to avoid unnecessary re-
allocation. To this end, VORPAL supports over-allocation of its data structures. All grid data structures are over-allocated to a larger region, but iterated over a smaller active region which contains meaningful data. Using this strategy, it is possible to expand or contract the size of a local region without the overhead involved in reallocating and copying data. Whenever the allocated space runs out, VORPAL reallocates with overflow and copies all data structures to the new size as shown in fig. 5.3.

The final step in decomposition adjustment is synchronization. Data values on boundaries must be synchronized with neighboring processors. Since ownership of grid cells changes, the data on these cells must be handed off to the neighboring processors. Additionally, particles that no longer belong on a certain processor as a result of the adjustment must be transferred to the correct node. VORPAL uses the same procedures to accomplish this that are used to synchronize data at the end of a time-step. As a result, additional code is unnecessary for this step. A single call is made to send/receive all boundary values after the decomposition adjustment is applied.
New memory block requested, values are copied over

Memory is over allocated to prevent future resizes.

Figure 5.3: Over Allocation

5.0.3 Load balancing: A case study

To illustrate the effect of load balancing we present the results of two identical simulations with and without load balancing enabled. The simulation consists of two nodes. Fig. 5.4 shows the update time for each step for each of the two nodes. The simulation is designed so that initially the pink node has a high load, but as the simulation progresses the load is naturally transferred to the blue node. Without load balancing to compensate for changes in load we see the update time initially high for the pink node and low for the blue node. Midway the situation is reversed. At only one time-step are the update times equal and efficiency is at 100%. Every other time-step one of the nodes is waiting on the other. When load balancing is enabled, the system compensates for changes in load and we see the load across both processors remains equal.
Figure 5.4: Load Balancing
Chapter 6

Parallel Build

Build time is often overlooked in the area of performance. Large applications with many interwoven dependencies, such as VORPAL, require lengthy build times.\(^\text{1}\) A minor change to the source code could take more than half an hour to compile. Applications such as GNU Make\(^\text{1}\) have a certain degree of parallel functionality, but only at the machine level. We propose a cluster-level build application that can drastically reduce application build time. For compatibility purposes, we use MPI as the message passing scheme of choice so that the parallel make software integrates directly into existing clusters. Several variations of this technique have also been implemented\(^\text{8}\)|\(^\text{7}\)|\(^\text{2}\). These strategies generally integrate into Make itself, relying on Make’s internal dependency graph to determine components that can be executed in parallel. Instead, our technique obtains a list of jobs that make would issue and searches for those jobs that can be executed in parallel. This increases the pool of available parallel jobs and keeps disk intensive jobs such as linking on the local machine. In this section we describe a parallel make application and its applicability to VORPAL. Section 6.0.4 sets a foundation for the problem by describing VORPAL’s build process. Sec. 6.0.5 details the parallel build algorithm in the context of a VORPAL build.

\(^{1}\) Extensive use of recursive templates makes compilation of VORPAL source code unusually lengthy.
6.0.4 Application build process

We describe here the general build process for the majority of C++, C, and Fortran applications. Parallel components are then identified and an optimal build process given parallel resources is quantified. We begin with a review of the three step build process.

1. Compilation: Convert each source file to an object file.
2. Archiving: Assemble object files into libraries.
3. Link: Combine object files and libraries into executables.

A large portion of the work in these tasks is independent and, therefore, can be done in parallel. The steps above must occur in the given order but each step can be done in parallel. For example, object files are never used as input to the compilation process. Hence all source files can be compiled simultaneously given available resources. The same argument holds for archiving and linking. Generally most time is spent on the compilation phase. This is even more true if compiler optimizations are enabled. They will increase source to object translation time and decrease link time since fewer symbols will be present in resulting objects due to source level inlining and overall smaller object sizes. Therefore, we consider only the first phase of this process for parallel execution on a cluster. The remaining steps are performed serially on the calling machine.

6.0.5 Parallel Build algorithm

Our parallel build algorithm greatly reduces application build time given sufficient computing resources. The software makes use of the widely accepted message passing interface (MPI) available on nearly all clusters used for scientific computing. Sec. 6.0.6
details the parallel build process step by step. Implementation and compatibility issues are discussed. Sec. 6.0.7 compares serial vs. parallel build times for a selected set of applications.

6.0.6 Parallel Build Implementation

The parallel make algorithm is described below in the following steps.

1. Determine all calls to the compiler to convert source to object code. This is done using the command line arguments -n and -k to GNU make. A perl script locates the specific calls to the compiler, as well as the directories in which each call is to be made. Since no linking is done during source to object translation, we are assured that these calls will always be independent and can be executed in any order.

2. Estimate compile time for each source to object conversion. A rough estimation is achieved by counting the number of characters in the pre-processed source. This is typically obtained through use of the -E command line option to the compiler. The size of the file to be compiled is usually a good relative indication of compile time. All source files are then sorted from greatest to least according to execution time.

3. In sorted order, each source file is dispatched to an available processor for translation to an object file. As processors become available, subsequent source files are dispatched for translation. This continues until all source files have been translated.

4. Object files are bundled into libraries and executables are linked.
This build process is fully automated. Errors during translation are detected and cause automatic termination of the build process. Two software components make up the parallel build application: a perl script and a parallel job manager. The perl script collects, formats, and sorts the compile commands. The parallel job manager takes the list of commands and dispatches them to available processors using MPI. Compilers read source files through the cluster’s shared file system and write back object files. The return values from the compiler, indicating success or failure, are reported back to the dispatcher to facilitate premature termination in the event of a failure. The process is fast and efficient, with little overhead.

6.0.7 Performance of Parallel Build

Parallel build performance depends on available computational resources and build characteristics of the application. We present here a case study comparing build time for VORPAL using parallel and serial methods. VORPAL’s code consists of 275 source files that can be compiled in parallel. To improve link time, which cannot be done in parallel, we enabled full compiler optimization to minimize object sizes. Using a 10 node cluster we found a six-fold increase in performance over the serial build. A serial compilation of VORPAL on a 2.8GHz Athlon takes 30 minutes, compared to five minutes when using the 10 node cluster. This includes archive and link time, which are both run in serial and take half a minute to complete on the Athlon. The resulting binary is 16M in size. Without compiler optimization the link stage alone takes five minutes and outputs a 120M binary. These results are not meant to be typical and are only an example when tested with VORPAL. Actual performance improvements vary dramatically depending on available computing resources and on the nature of the application. Theoretically, the source to object translation process should take no more time than the time to compile the largest single source.
Chapter 7

Discussion and concluding remarks

We have presented an overview of three optimizations added to VORPAL which have resulted in an overall quadrupling of performance. Particle sorting yields outstanding improvements, especially on machines with small cache sizes. Viktor Decyk’s results show a substantially smaller impact as a result of sorting. This is because his simulations load particles in sorted order whereas VORPAL does not. Kevin Bowers’ sorting implementation, which relies on the fact that each particle’s position corresponds to an index on a finite grid, is elegant and efficient, however the additional memory required makes it less desirable.

Software caching of particle indices and weights is an optimization very much specific to VORPAL and may be less applicable to other implementations. Since VORPAL is written with an Object Oriented structure, functionality of various operations is finely divided, despite the natural flow of data from one operation for the next. It is for this reason that we divide particles into groups so that individual operations within an update are applied to a group of particles at a time. Hence, software caching of otherwise re-computed values like indices and weights, is a natural step.

In practice we have found that dynamic load balancing is extremely costly due to the communication latency involved in re-synchronizing data values after a decom-
position adjustment. We believe that our one cell per step decomposition adjustment limitation is mainly responsible, as we are forced to pay a large performance penalty in exchange for minimal change in load. A more flexible implementation could prove far more powerful.

Runtime cost minimization is a feature that we have found to be nicely suited for VORPAL's Object Oriented structure. We are able to easily scatter timer objects throughout the code to measure wall clock time, user time, and idle time. Then, cost analysis routines are able to query their values and act on them. Without cost minimization we would be forced to experimentally determine optimal static points at which to sort and adjust the decomposition. As a result, we are assured that performance-related enchantments are exactly that, enhancements, and not potential bottlenecks for a small subset of test cases.

Runtime code compilation and execution has many advantages to performance critical applications where the demand for speed often exceeds the demand for flexibility. In situations where flexibility is preferred, performance suffers. We have presented here a method which makes the best of both features, combining flexibility in runtime configuration options with the performance of native compiled code. Through use of the GNU autoconf tool, we helped address platform dependent issues that this method inherently contains. Finally, we demonstrated how the performance of our method far exceeds the performance of even a sophisticated expression interpreter.
Bibliography


