View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Texas A&M Repository

CONCURRENCY IN A SYSTEM FOR SYMBOLIC AND ALGEBRAIC COMPUTATIONS

A Senior Scholars Thesis

by

STEFAN MAI

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the designation of

UNDERGRADUATE RESEARCH SCHOLAR

April 2009

Major: Computer Engineering

CONCURRENCY IN A SYSTEM FOR SYMBOLIC AND ALGEBRAIC COMPUTATIONS

A Senior Scholars Thesis

by

STEFAN MAI

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the designation of

UNDERGRADUATE RESEARCH SCHOLAR

Approved by:

Research Advisor: Gabriel Dos Reis
Associate Dean for Undergraduate Research: Robert C. Webb

April 2009

Major: Computer Engineering

ABSTRACT

Concurrency in a System for Symbolic and Algebraic Computations. (April 2009)

Stefan Mai
Department of Computer Science and Engineering
Texas A&M University

Research Advisor: Dr. Gabriel Dos Reis
Department of Computer Science and Engineering

As miniaturization of computer components is approaching the limits of physics, researchers in computer architecture are looking for less conventional means to perpetuate Moore's law. Recent trends in hardware ve been adding more cores. Consequently multicore machines are now commodity. To help programmers benefit from Moore's dividend, researchers in programming techniques, tools and languages have been exploring several venues. A dominant theme is the design and implementation of parallel algorithms. Several programming models have been proposed, but none at the moment seem to be substantially better than others. While *general* parallel programming is a distinctively challenging task, we believe that scientific computation algorithms display algebraic structures, thanks to the rich mathematical objects they manipulate. The present work aims at exploring the extent to which algebraic properties displayed by computer algebra algorithms may be automatically exploited to take advantage of parallelism in the OpenAxiom scientific computation platform. We designed a runtime system that exploits the ubiquitous parallelism of modern CPUs; the system is also scaled to many-system clusters. By taking advantage of the existing `InputForm` domain in OpenAxiom and connecting of the standard input channel to sockets, we were able to minimize potentially hazardous modifications to the OpenAxiom source while still implementing desired functionality. Additionally, we designed and implemented FFI extensions to the OpenAxiom core to take advantage of SIMD instruc-

tions, particularly SSE2 (SIMD Streaming Extensions). The extension allowed us to nearly double the speed of common operations such as multiplying arrays of doubles. We also defined and implemented a foreign function interface for the OpenAxiom system. All of these additions were benchmarked using Berlekamp's algorithm for factorization of polynomials over integers. While much still remains to be done in parallelizing the algebra to work over many calculation nodes, mathematical annotations remain viable in unloading the burden of parallelizing code from the programmer by substituting a simpler activity.

DEDICATION

This thesis is dedicated to all those who I sacrificed time with for the sake of finishing this thesis, and all the people with excitement for the field that made it worthwhile. Thank you.

NOMENCLATURE

FFI	Foreign Function Interface - Used to interface a Lisp system or other language with native binaries or other languages.
ALU	Arithmetic Logic Unit - A module of most processors that accepts commands to do simple arithmetic operations like addition.
SIMD	Single Instruction Multiple Data - An architecture where a single instruction is applied to multiple streams of data.
MIMD	Multiple Instruction Multiple Data - An architecture where many instructions are applied to many streams of data.
Spad	The algebra language of OpenAxiom
SBCL	Steele Bank Common Lisp
GCL	GNU Common Lisp
CLISP	GNU CLISP – An ANSI Common Lisp Implemnentation
ECL	Embeddable Common Lisp

TABLE OF CONTENTS

	Page
ABSTRACT	iii
DEDICATION	v
NOMENCLATURE	vi
TABLE OF CONTENTS	vii
LIST OF FIGURES	viii
CHAPTER	
I INTRODUCTION	1
SIMD	2
Parallelization server	3
Annotations	3
II IMPLEMENTATION OVERVIEW	6
SIMD	6
FFI	7
SIMD implementation	7
Parallel server	10
Berlekamp's algorithm	13
Pre-existing loops in OpenAxiom	15
III RESULTS	21
FFI	21
SIMD	21
Parallelization server	22
Berlekamp's benchmark	29
IV CONCLUSION	32
FFI	32
SIMD	32

Page

Future work 33

REFERENCES 34

CONTACT INFORMATION 35

LIST OF FIGURES

FIGURE Page

1 Ilustration of SIMD 5

2 C Prototype for a Function the Multiplies Arrays of Doubles 9

3 Spad Function For Multiplying Arrays of Doubles 9

4 Intel Intrinsic Code for Multiplying Two Arrays of Doubles 11

5 Optimized Aligned SIMD Operation Benchmark 23

6 Optimized Unaligned SIMD Operation Benchmark 23

7 Unoptimized Aligned SIMD Operation Benchmark 24

8 Unoptimized Unaligned SIMD Operation Benchmark 24

9 SIMD Function Example with Fallback 25

10 OpenAxiom Process Layout 27

11 OA-Serv Topology 27

12 OA-Serv Commands 31

CHAPTER I

INTRODUCTION

Everyone in the field of computer science is aware of the impending halt of performance gains stemming from Moore's law, the observation that the number of transistors in an integrated circuit doubles every two years. The failure of these steady gains comes from a combination of the limitations of manufacturing techniques and the speed of light, which only permits signals to travel a finite length in one clock tick.[1] This observation and the exploration of its solution is the main thrust of the field of parallel computing, which seeks to make use of multiple processing units to solve a single problem.

Meanwhile, during the past four decades, symbolic and algebraic manipulation has increasingly complemented numerical computation. Symbolic computation not only offers more accurate results compared with the typical numerical approximation techniques, but has the potential to provide insight into the relationships that underly the problems it describes (see Fateman (1972)). Additionally, solutions derived symbolically can be reused for large data sets, rather than having to perform redundant calculations. The description of these problems in symbolical form opens up the potential to perform various transformations of the problem into equivalent representations with identical solutions. For instance, one way to solve integrals is by translation into a form that is more easily looked up in a table of integrals. Similar procedures can be applied to more complex problems provided the translations do not alter the nature of the problem.

This thesis follows the style of Journal of Symbolic Computation.

[1] For instance, the speed of light limits the travel of a signal during one clock of a 3 Ghz processor to less than 10 cm

For this thesis, we look into the convergence of parallel and symbolic computation in the extension of OpenAxiom, an open-source computer algebra system (Dos Reis (2009)). One of OpenAxiom's primary advantages over the variety of computer algebra systems that are in use is its typeful nature, which provides an ideal foundation for performing provably correct transformations of sequential programs into their parallel equivalents. Thus, this thesis will take on three parts:

1. The exploration of using the SIMD capabilities of most modern processors to take advantage of a 'free' performance gain.

2. The creation of a server for use in the translation of existing sequential programs so that they can make use of multiple cores and multiple servers.

3. Using annotation information specified by the user programmer to make use of the added SIMD and parallelization facilities.

The goal of this thesis is not automatic parallelization, which would require more exhaustive research than the limited time permits, but semi-automatic parallelization based on the additional information of annotations provided by the programmer. Altogether, we seek to take advantage of all levels of parallelism, from vector processing instructions (SIMD) to node-level distributed parallelism within the OpenAxiom platform using mathematical annotations to guide the process.

SIMD

SIMD stands for single-instruction-multiple-data and in general usage refers to processor extensions like SSE, SSE2, SSE3 and MMX. All modern x86 processors include SSE support and the instructions have become the standard for SIMD operations on desktop PCs. A lot of very common operations like scalar multiplication of matrixes involve repeating

an operation on each element of a large array (see Fig. 1). This occurrence is especially prevalent in computer graphics where large arrays of pixels are operated on. Hardware manufacturers have taken this as an opportunity to design extra instructions and data paths that perform the operations in less cycles than by individually performing them on each element. OpenAxiom does not currently have any SIMD capabilities which makes this a straightforward performance benefit that can be compiled in to the binaries.

Parallelization server

Taking advantage of multiple cores and multiple CPUs (on applicable machines) is another area for potential parallelization. The mathematical foundation that underlies most scientific computations exhibits many instances of coarse-grained parallelism that lends well to multi-core operation. Where SIMD takes simple operations on large data sets, parallelization among cores, CPUs, or servers is an example of MIMD multiple-instruction-multiple-data parallism (see Bogong and Grishman (1982)). By identifying calculations that exhibit high degrees of symmetry or that operate with a small amount of data and instructions in an expensive computation, the commonplace multi-core architectures can be better utilized.

Annotations

As it stands currently, most programs written in low-level languages lack the structural information concerning dependencies and side-effects necessary to perform the necessary transformations to parallel. The manual transformation by a knowledgable programmer involves the injection of additional insight about the problem into the program that was not in the source code initially. The absence of this information precludes attempts to have the compiler do this processing on its own. By adding features to a language to fill in these holes in information, this approach obviates the difficulty of discerning these properties from the program's tree itself, which is expensive computationally (see Rabhi and Gorlatch (2002)).

These annotations ideally will provide the missing information required to make trivial parallelizations of the code. For example,

```
for x in 1..5 repeat
 b.x := gcd(a.x,poly)
```

is trivially parallelized if we can be certain that the function `gcd` has no externally visible side effects. Spad provides an ideal testbed for these operations as its feet are firmly planted in symbolic computation and its rigid structure allows for much easier transformations. By exploiting the additional information provided by the annotations, the compiler is able to transform the program to take advantage of areas that might be successfully parallelized. The SIMD extensions and parallelization server provide the vehicle for these parallelizations to be turned into tangible performance benefit.

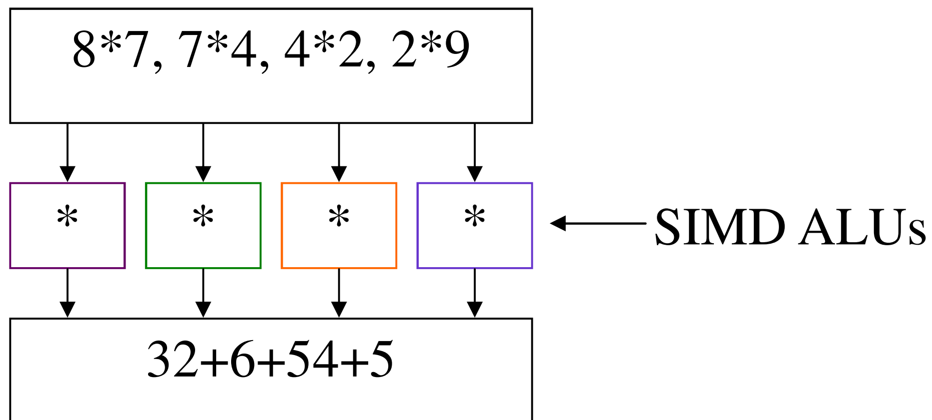

Fig. 1. Ilustration of SIMD

CHAPTER II

IMPLEMENTATION OVERVIEW

One of the traditional ways optimizing a program in a sequential programming framework is to take a certain body of code and profile its execution time by running it in such a way as to approximate its most general use. By gathering the times taken to execute various function calls, the bottlenecks of code can be tweaked as needed. With parallelization, the process is different because the goal is greater utilization of the CPU resources, and only indirectly an increase in speed. This increase is most often achieved through a flattening of execution across computation units like processing cores or compute nodes rather than reducing the number of instructions required to arrive at a solution. Profiling the execution is useful in finding the bottlenecks that would be most valuable in execution time, but the modifications are not nearly as simple. One of the most trivial ways of stretching execution across more computational units is by delegating iterations of loops, which can take a number of forms.

SIMD

SIMD architectures focus on building hardware with vector computation units and are typically geared toward graphics and simulations. The central idea is to speed up simple operations of lots of data by multiplying the amount of hardware available and pushing all the data through in one pass. Technologies such as SSE, SSE3, MMX, and others take advantage of arrays of arithmetic pipelines in order to increase the computational throughput of a single processor. Looping structures and operations on array-like data structures are the most obvious targets of SIMD solutions. Given that matrix multiplication is a very big part of numerical computation and a lot of the practical applications of computer algebra, these optimizations can lead to considerable, practical performance gains.

Yet, getting SIMD functionality into OpenAxiom brought several challenges with it. The first obstacle was finding SIMD functions for each of the Lisp systems supported by OpenAxiom.

FFI

OpenAxiom, like many computer algebra systems, is run on top of a Lisp subsystem, though this is not theoretically essential as languages like spad operate on top of the Lisp system and are the primary languages. In particular, OpenAxiom supports many different Lisp implementations, such as GCL, SBCL, ECL, and Clisp (see Dos Reis (2009)). Each Lisp implementation brings its own peculiarities, and OpenAxiom operates on a subset of the Lisp functionality common to them all with extra functions that operate as abstractions of functionalities that differ per Lisp system. Additionally, some Lisp systems depend on the operating system they are compiled for, with specific system functionality differing greatly between Windows and Linux or BSD[2]. Another area where the Lisp systems differ significantly is in their FFI or foreign function interface. An FFI allows a Lisp system to interface with code written in other languages. This is useful for the vast library of scientific computation libraries written in Fortran or C but also because, for our purposes, it's easier to write a C library that provides SIMD functionality than to write SIMD code for each Lisp system and choose it at runtime. The actual implementation of our FFI for OpenAxiom is detailed in a technical report (Dos Reis and Mai (2009)).

SIMD implementation

While SIMD extensions do exist for some Lisp systems (see Herring (2008)), our decision was to write a C library that took advantage of the FFI we added to OpenAxiom. In addition to this C library, we wrote an Axiom domain that used the functions provided by the library.

[2] For instance, Windows' path names use \ while Linux systems use /

The domain eases the requirements of the underlying C library. One simple example is to automatically pass the length of an array, since the FFI supports arrays but makes no explicit declaration of their length and the C function needs the length in order to do the operation. Thus, where the C function prototype might look like Fig. 2, the Spad function can be simpler without an explicit declaration of length as in Fig. 3.

Writing the fastest C library to be compiled with GCC was also the subject of investigation. There are several ways to write SSE code from within GCC. To explore which would result in the fastest code for our purposes, a trivial example of multiplying two variable length arrays and placing the result in a third was created. Then code was written in 'naive' C to use as a control for the SIMD instructions. The tests were created by applying GCC's native optimizations (`-O3`), taking advantage of Intel Intrinsics for GCC, and by writing GCC inline assembly. Afterwards, integration testing was done from within OpenAxiom, where it was revealed that the Lisp systems did not provide 16-byte aligned data. Without aligned data, the number of possibilities for our SIMD library was quickly cut down, but the benchmarks provided a basis for making the best decision. The results of the benchmarks are explained in Chapter III.

Native optimizations

GCC's native optimizations are enabled by compiling with the `-O3` option in addition to alerting GCC to the presence of SSE instructions with `-msse -msse2`. In spite of being a remarkably easy way to optimize programs, GCC's native optimizations actually performed well in comparison to the other options we explored.

Intel intrinsics

Although termed Intel intrinsics, GCC's implementation is cross platform in that it will work equally well with AMD processors or any other platform that supports the SSE assem-

```
int simdMultiplyDoubleArrays(int len, double* a, double* b, double*
dest);
```

Fig. 2. C Prototype for a Function the Multiplies Arrays of Doubles

```
import simdMultiplyDoubleArrays : (SingleInteger,
  ReadOnly PrimitiveArray DoubleFloat,
  ReadOnly PrimitiveArray DoubleFloat,
  WriteOnly PrimitiveArray DoubleFloat)
  -> SingleInteger from Foreign C

multiplyDoubleArrays (a,b,dest) ==
  (#a ~= #b) or (#b ~= #dest) =>
    error "Array sizes must be the same."
  simdMultiplyDoubleArrays(#a::SingleInteger,a,b,dest)
  return dest
```

Fig. 3. Spad Function For Multiplying Arrays of Doubles

bly instructions. Intel intrinsics alleviate the duty of allocating the 128 bit XMM registers for the purpose of SSE instructions. Additionally, by specifying the type of the data stored in a register, the compiler can do type-checking on the non-standard instructions. Intel intrinsics essentially allow the programmer to type-safely embed a subset of SSE asssembler instructions into their code without actually writing assembly. A sample of Intel intrinsic code is included in Fig. 4.

GCC inline assembly

C's primary purpose is writing operating systems, and GCC's inline assembly feature makes this obvious. It provides the capability to write assembly code inside a C program, specifying which registers will need to be replaced after the code runs and what local variables to put the result registers into. This option gave us the capability to write optimized assembly instructions for the test benchmark. Unfortunately, as powerful as it is, assembly isn't portable and while code written for x86 architectures will work across systems, on any other system it will not. The assembly capability freed us to use certain unaligned instructions, which show up in the benchmarks because other manners of writing the SSE instructions do not allow for non 16-byte aligned data.

Parallel server

The primary goal for the server architecture that serves as a foundation for this new system is to manage a number of OpenAxiom kernels with low latency and overhead while maximizing the utilization of all systems designated for use. More optimization can be achieved at runtime by allowing information concerning the speed of execution to be recorded that can inform future passes. The server needs to be able to take advantage of any number of CPUs on the host system, other available systems on the network, and the architectures of these systems to exploit SIMD opportunities. Our solution involves creating server instances (hereafter called OAServs) that can intercommunicate to determine network la-

```
//This is a function using the GCC "Intel Intrinsics", the
//loop is unrolled such that 4 doubles are multiplied for each
//run through the loop
void intel_intrinsics_4(double* a,double* b,double* c,int arraysize){
  __m128d a1r, b1r, c1r, a2r, b2r, c2r;
  unsigned int x=0;
  for(x=0;x<arraysize;x+=4){
    a1r = _mm_load_pd(a+x);
    b1r = _mm_load_pd(b+x);
    a2r = _mm_load_pd(a+x+2);
    b2r = _mm_load_pd(b+x+2);
    _mm_store_pd(c+x,_mm_mul_pd(a1r,b1r));
    _mm_store_pd(c+x+2,_mm_mul_pd(a2r,b2r));
  }
  //This ensures that odd sized arrays are completely multiplied
  for(x=1;x<arraysize%4+1;++x){
    c[arraysize-x] = a[arraysize-x]*b[arraysize-x];
  }
}
```

Fig. 4. Intel Intrinsic Code for Multiplying Two Arrays of Doubles

tencies and the shortest path to execution.

Kernel management

Each OAServ will maintain an optimal number of kernel processes (usually equal to the number of cores times the number of cpus) as well as information regarding their availability, response times, and historical information about the excution of blocks of code. In addition, the OAServ keeps connections to other OAServ processes on the network with “remote” kernels that are available if the local kernels become saturated. OpenAxiom processes can connect to the OAServ to dispatch commands packaged with data. These packets are delegated to the nearest and/or most readily available computation kernel, with local kernels favored over remote kernels. If it is estimated that all of the local kernels will finish their tasks in 10ms, then it doesn’t make sense to push the task to another server 50ms away that has free kernels. Meanwhile, if the local tasks might take 500ms, then the trip becomes more economical.

Client-side library

A parallelization library will be written for OpenAxiom that takes in a syntax tree and outputs a functionally equivalent tree with built in communication to the OAServ. As each task is dispatched in the source from OpenAxiom, they are given a unique identifier to allow the OAServ to profile the operations. Moving averages should be kept as to the computation time required to complete the tasks which will allow the server to dynamically adapt to (or reject) jobs as they arrive.

Serialization facilities

The process of serialization is currently accomplished through the InputForm facilities in OpenAxiom. This domain models objects in textual form such that they can be accurately recreated on another process. This works well for simple lists and symbols but fails utterly

when extended to more complicated structures, particularly with cycles. Additional work will need to be done into establishing and implementing a protocol for the communication of mathematical structures between OpenAxiom and the parallelization server. There already exist a few protocols such as MathML that may work, but research will have to be conducted into the overhead of packing and unpacking MathML structures (as well as the time expense of developing a compatible implementation) over something closer to the OpenAxiom native representation.

Pattern matching

One of the primary strengths of OpenAxiom is the typing facilities that are built in. By serializing data and instructions to be pushed to remote nodes, the resulting data does not always have an easily assumable type. Many functions in OpenAxiom will return a string detailing their failure or the correct result. Pattern matching allows us to recreate the type at compile time in the destination.

```
case r is
 i@Integer => -- Do operations on i as an Integer
 d@DoubleFloat => -- Do operations on d as a DoubleFloat
 otherwise => error "r is not of a prepared type."
```

The pattern matching serves two purposes. The first is that it executes different code depending on the type of the incoming data. The second is that it allows the compiler to assume the type of the data following its assertion by the earlier case statement. This feature lets OpenAxiom maintain type safety within the uncertain environment of executing code on a remote host where the result is not of a definite type.

Berlekamp's algorithm

Berlekamp's algorithm (see Berlekamp (1967)) serves as the motivating example for all the

parallelization work. At present, only SIMD functionality has been added. An approximate overview of Berlekamp's algorithm is as follows:

1. Make sure that the polynomial is squarefree by taking the GCD of the polynomial against its derivative. If the result is not one, you will need to remove the squared roots first.

2. Create a Berlekamp Q matrix for the polynomial, of size $N \times N$ where N is the degree of the polynomial to be factored. Each row is given by $x^{np} \bmod u(x)$ where n is the 0-based index of the row, p is the prime number of the field we are factorizing under and $u(x)$ is the polynomial being factorized.

3. Take the Q matrix and subtract the corresponding identity matrix, then find the null space basis of the result.

4. For each vector in the basis, calculate $\gcd(u(x), v(x) - s)$ where s is all integers between 0 and the prime number representing the field. The factorization is the product of all nonzero results.

For conceptualization purposes, the algorithm can be broken down into three main sections (see Schreiner (2001)):

1. Create the Q Matrix

2. Find the Null Space Basis

3. Test All Possible Factors

Much work has been done relating to the generation of the Q matrixGeddes et al. (1992) as well as heuristics to find those $v(x) - s$ that are mostly likely to be factors. While these

findings are both significant and yield substantial performance gains, for the purposes of our testing we sought optimizations related to the machine rather than the algorithm itself.

SIMD opportunities

For the generation of the Q matrix, each row is generated by a transformation of the previous row such that $a_{k+1,j} = a_{k,j-1} - a_{k,n-1}u_j$(from Knuth (1981)). Each successive row can be generated by subtracting a scalar multiplication of u by the previous row from the previous element in the current row, generating the row from the lowest element to the highest.

In taking the null space of the Q matrix, the matrix is column reduced by traversing each row and pivoting on a non-zero column to eliminate other columns that have elements on that row. In the resultant matrix, the nonzero rows describe polynomials to test in the next section. Each column reduction is performed by subtracting the original value of the column from a scalar multiple of the pivot column. Both scalar multiplication and element-wise subtraction can be performed using SSE instructions.

No SIMD optimizations were attempted in the testing of possible factors.

Pre-existing loops in OpenAxiom

The OpenAxiom software has a wide array of libraries that are shipped with the binaries or can be readily obtained from the internet. Facilities exist to query the libraries that are loaded to examine their code or the domains they provide, making it easy to get an idea as to the nature of the loops that already exist. Specfically, the Syntax domain allows the user (or programmer) to manipulate and inspect the code produced by a function. Static analysis of these loops should give rudimentary information as to which can be broken into pieces or parallelized as is. While annotations of functions have not yet made their way into the OpenAxiom kernel, once they have been added it should be straightforward to write Spad

code using the Syntax domain to inspect loops and discover whether or not transformations can be accomplished depending upon the properties of the function calls and assignments made inside. This is future research work.

Mathematical optimizations

On of the easiest functions to do parallelization of is the GCD function. GCD is commutative and associative, and has an identity element of 1 and a neutral element of 1. These properties give us a number of distinct advantages:

Associativity and Commutativity Together these properties fulfil the requires for divide and conquer. This allows us to spread out function exections without concern for the order in which they are taken apart or reassembled.

Identity Elements Operations performed on member elements of a set that are identity elements are inconsequential to the aggregrate computation. By eliminating function calls that should surely result in the same value as the input $(5+0)$, it's possible to eliminate redundant calculations.

Annhilihators or Neutral Elements Neutral elements, once encountered, reveal the solution without calculation. The solution of $\gcd(5x^2+2x+1, 6x^7-9x^2, 8x^4, 1)$ is clearly 1. These are the lowest hanging of optimization fruits.

Loop optimizations

The process of transforming loops at compile time is relatively straightforward for easily conceptualized programs, but grows in difficulty as new concepts are brought in. The burden of mental gymnastics involved can be mitigated through the use of careful abstractions, a technique not far from anyone involved with computers. Part of the abstraction-forming process is realizing the limitations of the underlying code that need to be encompassed. By

dividing code into pieces that can be rearranged using certain rules if specific conditions are met, the problem of parallelization can be reduced to that of symbolic manipulation. This allows us to create analogs to mathematical concepts (like associativity of operators and the rearrangements of equations) which creates a pathway for the reuse of algorithms and concepts already researched in the field of computer algebra. Thus, the parallelization and speed up of a computer algebra system is accomplished through information readily available in the field of computer algebra.

Runtime unknowns

One such limitation on rearrangement is that of runtime unknowns. Compile-time optimization is limited by the information present before execution. For instance, unrolling the GCD of a set of number by hand is tedious and quickly grows into an unsolveable problem.

```
return gcd([11,75,34,87,45,23])
```

Becomes:

```
a = send(1,"gcd(11,75)")
b = send(2,"gcd(34,87)")
c = send(3,"gcd(45,23)")
d = send(4,"gcd("+a+","+b+")") -- depends on a and b?
e = send(5,"gcd("+d+","+c+")") -- depends on d (a nd b) and c?
return e
```

And if the literal set is replaced with a variable, the loop cannot be unrolled at compile-time.

Dependencies

Next, if *any* iteration is dependent upon a previous iteration, the process of parallelization

breaks down because subsequent calculations in the data path require results form earlier steps.

Sometimes this dependency relationship, when appropriately mapped out, can be simplified. For instance, if a single invocation is always dependent on the previous invocation (making it implicitly sequential), but a secondary function fuction is pure, then:

```
result := y
max_val := set.1
for x in 1..n repeat
  result := gcd(set.x,result)
  inv_set.x := (set.x)^-1
```

Can be transformed to:

```
result = y
for x in 1..n repeat
  result:= gcd(set.x,result)
for x in 1..n repeat
  inv_set.x := (set.x)^-1
```

The pure function loop can be trivially parallelized and the footprint of the non-parallelizable loop is reduced.

But this is only possible if we can be certain that the output of gcd() in a single iteration does not affect the input or execution of the inverse immediately afterwards. In effect, the result can only be guaranteed functionally equivalent if we can provide specific properties on the functions involved.

Latency thresholds

If the latency of the network dominates the time of execution for the loop body, then the process of parallelizing the loop may actually increase execution time. While complete compile-time estimation of the time required to complete a sequence of instructions degenerates to the halting problem, simple heuristics might suffice when paired with an intelligent run-time system. Serializing and transmitting data structures brings its own problems (and expenses), and finally typing the transaction is a nightmare.

Present solutions

The present manner of dealing with these difficulties is to have the programmer perform the tasks manually. But bookkeeping and micromanagement are generally fields that humans make mistakes often, whereas computers shine. The most modern approaches ask the programmer to instruct the compiler as to how the code can be parallelized. Typically, these annotations are little more than macros that expand the code into a series of delegations. For instance:

```
result = 1
for x in 1..n repeat -- %ASSOCIATIVE_PARALLELIZE
  result := gcd(set.x,set.(x+1))
```

Might produce code for the parent node like:

```
for x in 1..n repeat
  push(stack,set.x)
runstackop(stack,set,gcd)
```

And code for each node that resembles:

```
if size(stack)>2 then
  lock(stack)
  a := pop(stack)
  b := pop(stack)
  unlock(stack)
  temp := gcd(a,b)
  lock(stack)
  push(stack,temp)
  unlock(stack)
```

Macro systems such as these are available for some modern computer algebra systems (like Schreiner (2009)). These systems significantly reduce the amount of code required by the programmer, but fail to add any understanding of the underlying program by the compiler. Our approach differs in what is annotated: while currently the emphasis is on instructing the compiler as the the location of potential improvement, our approach focuses on describing the mathematical structure of functions such that the compiler can programmatically determine the locations for improvement. This involves specifiying the properties of functions, rather than places where their invocations are portable to other machines or cores. If a function is pure (that is: it has no side effects), then the compiler can assume independence of iterations and choose to trivially parallelize the loop without intervention from the programmer. This presents a number of benefits, one obvious reason being that old code doesn't need to be rewritten to annotate parallelizable loops. Additionally, when writing code the knowledge of how a function works and what it does is clearer during its creation than its invocation.

CHAPTER III

RESULTS

FFI

One of OpenAxiom's strengths is its ability to work across many different Lisp systems. Supported Lisp systems include SBCL, ECL, GCL, and most versions of CLisp (specifically those that have an FFI interface built in). The challenge of getting Lisp to emit the assembly instructions required to take advantage of SIMD (notably SSE) had a number of solutions, but the foremost aspect we were looking for was speed in implementation. OpenAxiom already had in place an incomplete solution for interfacing Boot code with C code that was compiled with the rest of the Axiom libraries. This was primarily used as a Lisp system portable manner of interacting with the filesystem. It was extended to work with Spad code through an addition to the grammar. The syntax allows for the import of functions from shared libraries compiled with C (but can easily be extended to support other languages). By implementing a SIMD library in C, we were able to mitigate the problem of having the Lisp system emit assembly instructions for the SSE instructions.

The FFI in OpenAxiom allows the passing of arrays represented by the `PrimitiveArray` domain as well as all of the primitive data types of the C language as their OpenAxiom domain equivalents. Meanwhile, many domains in OpenAxiom do not make use of `PrimitiveArray`, instead opting for `List` or `Vector`. While these domains can be coerced into `PrimitiveArray` and vice-versa, the computational overhead associated with the conversion is expensive. This is not uncommon in SIMD conversions, and limits the scope of the SIMD functionality until transformations can take place that might alter the intermediate data structures of existing algorithms and routines.

SIMD

For the SIMD implementation, benchmarks were performed with both 16-byte aligned arrays (Fig. 5 and 7) and unaligned arrays (Fig. 6 and 8), both with optimization on (`gcc -O3`) and off against our test benchmark of multiplying arrays. Each benchmark was done for applicable expressions, like inline assembly or Intel intrinsics.

Of particular note for the results: GCC optimizes C pretty well by emitting instructions that use the floating point stack in a manner such that the processor is able to pipeline the double multiplications across the floating point units. Its speed in aligned instructions very nearly approaches the hand-worked assembly. While aligned operations have a much more significant variety in the way you can produce code, unaligned operations are not quite so resilient with many producing segfaults when fed unaligned input. These benchmarks were left out of the final data sets. Running sample code that makes an FFI call with a simple `PrimitiveArray` in OpenAxiom shows that FFI calls are not guaranteed to be 16-byte aligned. Furthermore, the IA-32 architecture actually dictates[3] that SBCL's implementation of vector is not 16-byte aligned, forcing our SIMD library to assume unaligned arrays.

After running these benchmarks, several SIMD functions were put in the library each with an SSE implementation and a 'fallback' implementation (Fig. 9). This design allows us to tightly integrate OpenAxiom code with the library without fear of incompatibility in systems where SSE or a similar architecture is not available.

Parallelization server

In order to avoid potentially hazardous modifications to the Axiom kernel, the choice was made to develop a standalone server to support multi-core parallelism and multi-node distribution. Presently, OpenAxiom exists as a set of processes governed by a single "sman"

[3] In structs, doubles are aligned on 4 byte boundaries rather than 8 byte boundaries as elsewhere.

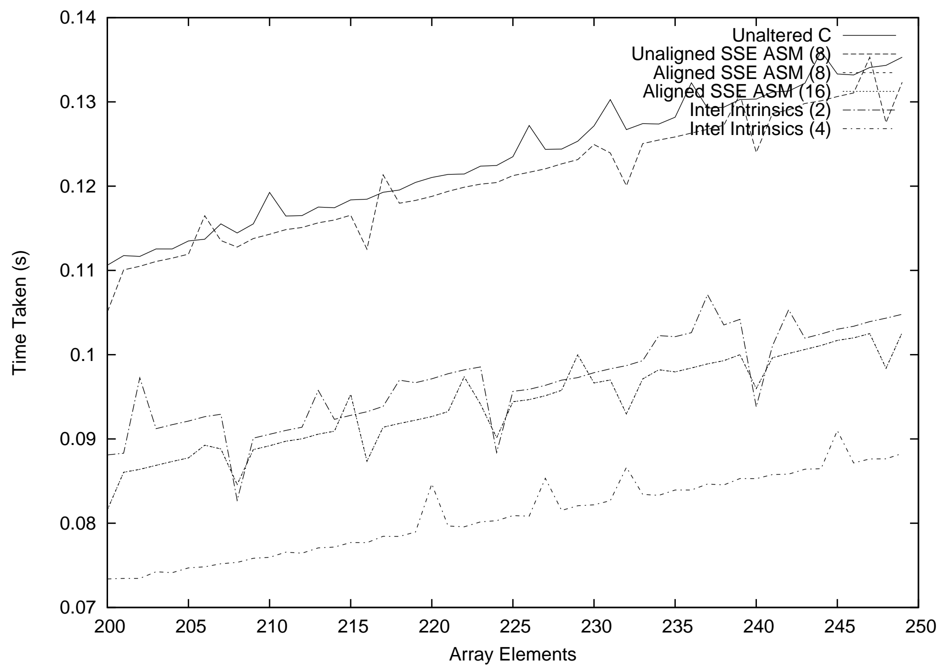

Fig. 5. Optimized Aligned SIMD Operation Benchmark

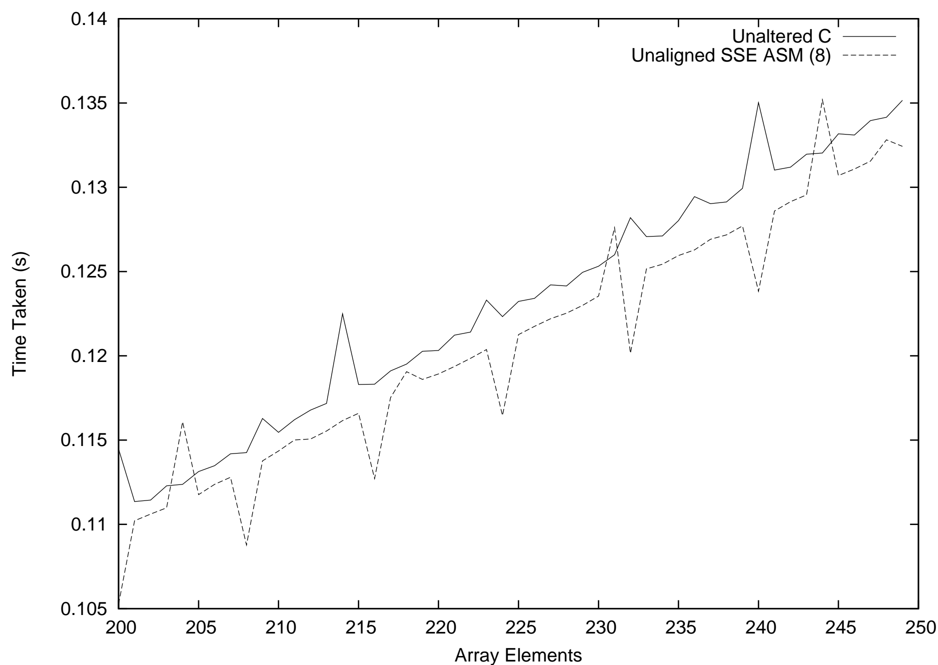

Fig. 6. Optimized Unaligned SIMD Operation Benchmark

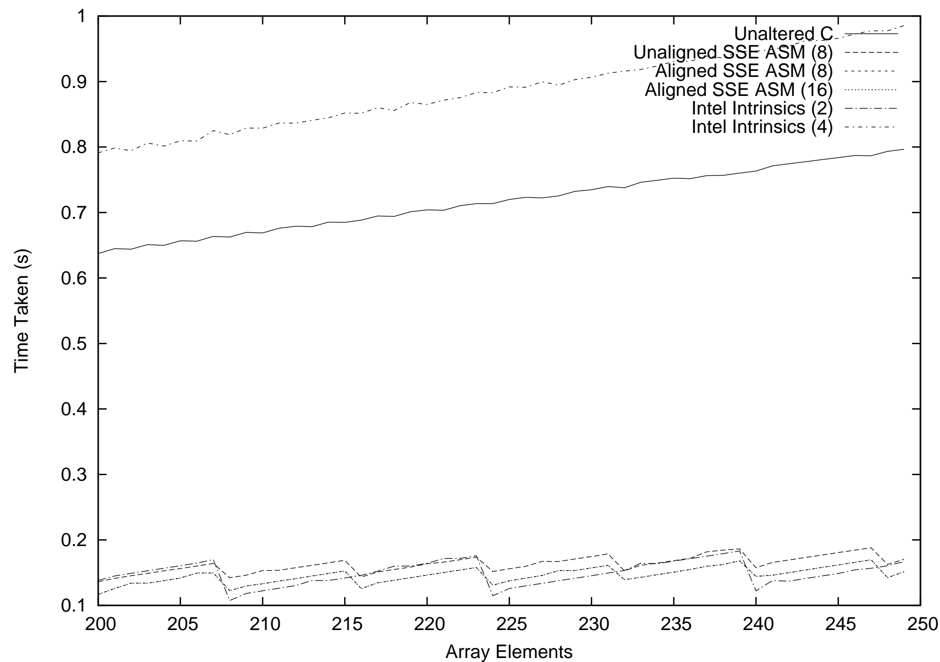

Fig. 7. Unoptimized Aligned SIMD Operation Benchmark

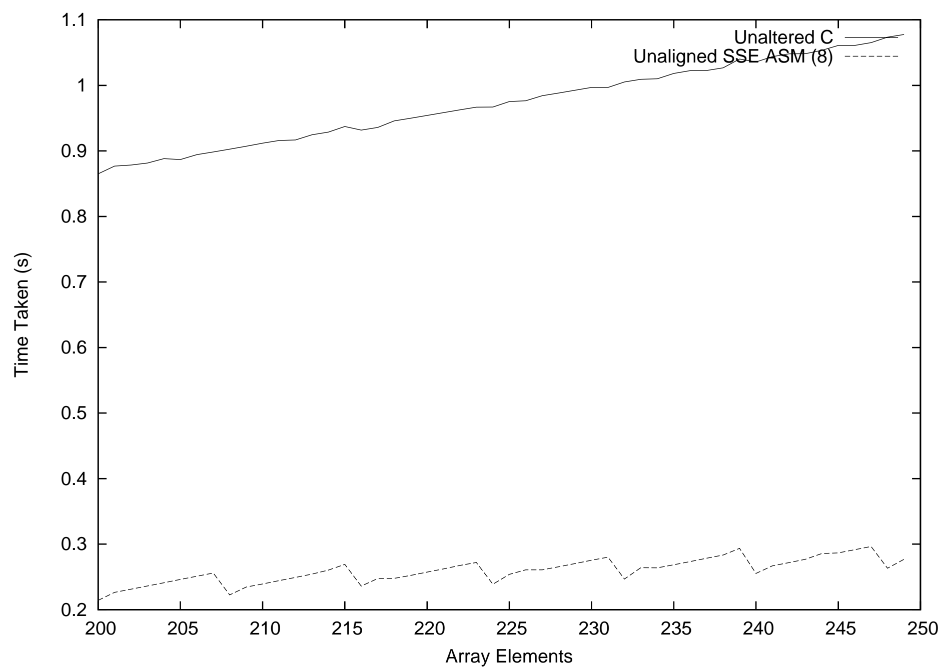

Fig. 8. Unoptimized Unaligned SIMD Operation Benchmark

```
int simdMultiplyDoubleArrays(int len, double* a, double* b, double* dest) {
#ifdef __SSE2__
  int x, end = len-len%4;
  //We're unrolling the operation into sets of 4 doubles
  for(x=0;x<end;x+=4){
    asm volatile(\
        "movupd   (%0), %%xmm0\n" //Move 0:(a+x) into register
        "movupd 16(%0), %%xmm1\n"
        "mulpd    (%1), %%xmm0\n" //Multiply 1:(b+x) and register (a+x)
        "mulpd  16(%1), %%xmm1\n"
        "movupd    %%xmm0,   (%2)\n" //Store back out to 2:(c+x)
        "movupd    %%xmm1, 16(%2)\n"
        :
        : "r" (a+x), "r" (b+x), "r" (dest+x)
        : "memory"
    );
  }
  //We can't use simd for these
  for(x=1;x<len%4+1;++x)
    dest[len-x] = a[len-x]*b[len-x];
#else
  int x; //Fallback
  for(x=0;x<len;++x)
    dest[x] = a[x]*b[x];
#endif
}
```

Fig. 9. SIMD Function Example with Fallback

or superman process (see Fig. 10). The child processes include hyperdoc (a graphical X-window process for documentation and examples), clef (a process for gathering user input and offering rudimentary history operations), and AXIOMsys (the actual Axiom kernel for computation).

OA-Serv architecture

Most of the overhead associated with hyperdoc, sman, and clef did not need to be loaded for our parallelization server. Thus, we created a server which loads AXIOMsys processes on sockets for use as calculation 'kernels'. These servers, dubbed OA-Servs, each can load a theoretically infinite number of kernels (though practically the number is best left at the number of cores available). Additionally, OA-Servs are able to connect together to form cluster-like networks (Fig. 11). This obviates the problems with firewalls as only one machine needs to be network-facing and it can serve as the server for all the machines behind a firewall.

Each OA-Serv accepts commands (Fig. 12) delimited by newlines and returns results in the order in which they finish. For most purposes, the clients use commands (like single) that can associate the result with the command that was sent through the use of IDs.

The server can be extended easily to implement new execution plans or command types. For instance, the AssociativeCommandBucket creates a stack which kernels can pull from (if the stack has more than one element) and deposit the result. At start, all the elements of the set are pushed onto the stack. As each kernel frees up and looks for a job, two elements are popped off and when the job finishes a single result is pushed back on. With the associativity property, this ensures correct results (and assuming uniform execution time, $O(log(n))$ execution time, but could later be extended to select pairs of elements that are most likely to run quickly together.

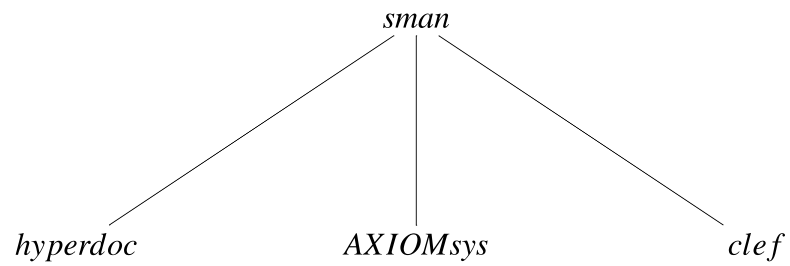

Fig. 10. OpenAxiom Process Layout

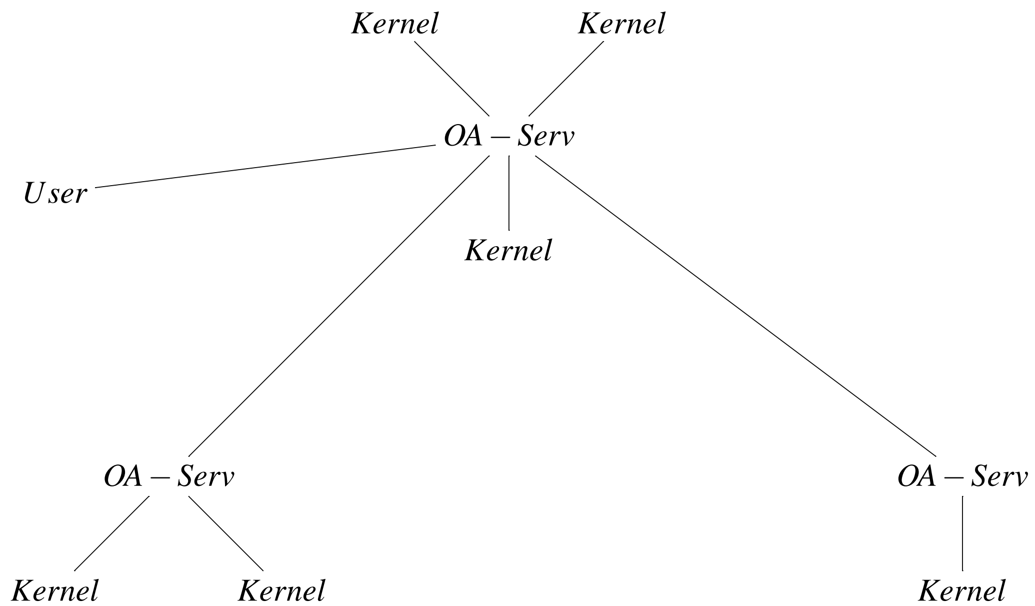

Fig. 11. OA-Serv Topology

Each server runs sequentially, using an event-based paradigm with the socket-multiplexing select statement to handle incoming connections and network writes. In order to add distribution capability without additional threads, a simple handshake was created. When a server saturates all its own local nodes with calculations, it broadcasts a FREE command to all the connected nodes. Each node will accept the FREE command, examine its command buffer and local node states, and respond with a GIMME if there is room for more commands. The source node responds to GIMME by pushing out to the requesting node a SINGLE command, that specifies a command to run and an ID to associate with it. When this command finishes, it returns a SINGLERESPONSE command to the source node which is then associated with the command sent and checked off of the queue.

Transport

One of the primary hurdles associated with parallelization is the task of transporting data between processes. There are many options, with various tradeoffs associated with network latency, throughput, and the potential for missing data. By limiting our parallelization to the exploitation of algebraic properties, all of the shipments can be characterized by defining closures. For instance, in the example of Berlekamp's algorithm, the factors can be found by taking the GCD of the polynomial with a set of potential factors created from the Q matrix's null space basis. If this operation is described by

```
for v in basis
 for s in 0..p-1
  map(test+->GCD(polynomial,test),v-s)
```

Then each shipment can be described in two parts: the operation to be performed and the variables required to perform it. In this case, if the function called by map is pure and doesn't depend on external environmental variables then all the necessary variables can be

found in the parameters to the function, GCD.

For our tests, the easiest way to create these shipments without extensive modifications to OpenAxiom was to create OpenAxiom commands from within OpenAxiom to ship to each server. Thus, our example might be written:

```
[GCD(polynomial,v.1),GCD(polynomial,v.2),...GCD(polynomial,v.n)]
```

By substituting the actual values for each element of v (since v will not exist on the remote servers to evaluate) and attaching type information, a simple protocol is established for shipping calculations. Each server will receieve:

```
GCD((x^25+x+1)::Polynomial PrimeField 5,
 (x^4+2x^3+4x+2)::Polynomial PrimeField 5)
```

The result is:

```
(x^4+2x^3+4x+2)::Polynomial PrimeField 5
```

By multiplying together all the results of this mapping, we've solved for all the factors of the polynomial.

Berlekamp's benchmark

In order to test the efficacy of annotations as a way to semi-automatically parallelize programs, we considered a generic implementation of Berlekamp's algorithm for the factorization of polynomials over finite fields.

The generation of the Q matrix can be done through binary powering, a process which creates a tree of dependencies but can generate the matrix in O_{logn} time rather than O_n.

Unfortunately, this does not take advantage of annotations and thus isn't automatically parallized.

Once the Q matrix is generated and the null space basis generated, the task degenerates into a problem of testing by division all the possible factors from the basis. This process can be accomplished by mapping the pure function that takes the GCD of an input and the original polynomial to the basis set. This does not take advantage of the speedup associated with using the reduced polynomial as factors are found, nor can it finish early once the polynomial's factors are all discovered. For this, extra information will have to be sent to the server to signal an end-of-calculation condition.

list	Prints the current status of the server and connected servers.
connect	Connects one server unidirectionally to another.
seq	Runs a group of commands sequentially across as many servers as possible.
assoc	Runs an associative function across a set.
single	Runs a single command with an identifier.
quit	Kills the server and its kernels.

Fig. 12. OA-Serv Commands

CHAPTER IV

CONCLUSION

Our results show that performance gains are possible both through the use of SIMD instructions and by parallelizing code and distributing it among cores and nodes. While hand-worked parallelization is not new, the potential for parallelization through static analysis and annotations remains a viable step in the march toward programmer-aided compiler-driven parallel transformations.

FFI

The FFI facilities of OpenAxiom were extended and hardened by testing with regard to SIMD. By enabling developers to serparate their native code from the source of OpenAxiom, it should speed development of extensions that require native functionality (like SIMD) by removing the compilation of the kernel from the build process. The FFI of OpenAxiom can easily be extended to support Fortran so that the vast library of scientific libraries available in Fortan can be used, which will go a great way in progressing OpenAxiom toward becoming both a symbolic and numeric computation system. As a computer algebra system that already operates in a heterogeny of languages, adding a FFI was a natural, yet necessary extension.

SIMD

SIMD shows potential in the narrow example of Berlekamp's algorithm and offers the promise of performance gains for most modern computers without extensive configuration of parallelization servers or access to clusters of servers. There may still be minor performance gains to be had in writing more efficient assembly routines or by tweaking the underlying Lisp system's manner of allocating arrays, but most of these wins are dwarfed

by the time spent in arranging bytes and coercing data structures to be used by SIMD.

Future work

A lot of work remains with respect to static analysis and creating a protocol for the transmission of OpenAxiom data structures in between servers and CPU cores. A means of efficiently converting more complex structures like cyclical lists in such a way that they are expressed fully without being overly verbose remains a challenge. Extensions must be added to the OpenAxiom compiler to support annotation of functions before any static analysis can start, but the Syntax domain provides a jump-start on the process by skipping the lexing and parsing steps. When the static analysis is in place, many of the features such as moving averages and execution profiling can be added.

REFERENCES

Berlekamp, E. R., 1967. Factoring polynomials over finite fields. Bell Systems Tech 46, 1853–1859.

Bogong, S., Grishman, R., 1982. Emulating an mimd architecture. In: MICRO 15: Proceedings of the 15th annual workshop on Microprogramming. IEEE Press, Piscataway, NJ, USA, pp. 197–199.

Dos Reis, G., 2009. Openaxiom. Website, http://www.open-axiom.org/.

Dos Reis, G., Mai, S., April 2009. Foreign Function Interface for OpenAxiom. Tech. Rep. TAMU-CSE-2009-4-2, Department of Computer Science & Engineering, Texas A&M University, College Station.

Fateman, R. J., April 1972. Essays in Algebraic Simplification. Master's thesis, Massachusetts Institute of Technology.

Geddes, K. O., Czapor, S. R., Labahn, G., 1992. Algorithms for Computer Algebra. Kluwer Academic Publishers, Norwell, MA.

Herring, D., 2008. Sb-simd simd support for sbcl. Website, http://common-lisp.net/project/sb-simd/.

Knuth, D. E., 10 Jan. 1981. Seminumerical Algorithms, 2nd Edition. Vol. 2 of The Art of Computer Programming. Addison-Wesley, Reading, MA.

Rabhi, F., Gorlatch, S., 2002. Patterns and Skeletons for Parallel and Distributed Computing. Springer-Verlag, London, UK.

Schreiner, W., July 2001. Parallelizing the Big Prime Berlekamp Algorithm with Distributed Maple. Tech. Rep. 01-15, RISC Report Series, University of Linz, Austria.

Schreiner, W., 2009. Distributed maple. Website, http://www.risc.uni-linz.ac.at/software/distmaple.

CONTACT INFORMATION

Name: Stefan Mai

Address: Parasol Lab
301 Harvey R. Bright Bldg, 3112 TAMU
College Station, TX

Email Address: stefan.mai@iamnafets.com

Education: B.S. Computer Engineering, Texas A & M University, May 2010