An Optimizing Compiler for Parallel Chemistry Simulations*

Jun Cao[1], Ayush Goyal[2], Samuel P. Midkiff[1] and James M. Caruthers[2]
[1]Purdue University
School of Electrical and Computer Engineering
West Lafayette, IN 47907
{caoj,smidkiff}@purdue.edu
[2]Purdue University
School of Chemical Engineering
West Lafayette, IN 47907
{ayush,caruther}@purdue.edu

Abstract

Well designed domain specific languages enable the easy expression of problems, the application of domain specific optimizations, and dramatic improvements in productivity for their users. In this paper we describe a compiler for polymer chemistry, and in particular rubber chemistry, that achieves all of these goals. The compiler allows the development of a system of ordinary differential equations describing a complex rubber reaction – a task that used to require months – to be done in days. The generated code, like much machine generated code, is more complex than human written code, and stresses commercial compilers to the point of failure. However, because of knowledge of the form of ODEs generated, the compiler can perform specialized common sub-expression and other algebraic optimizations that simplifies the code sufficiently to allow it to be compiled (eliminating all but 6.9% of the operations in our largest program) and to provide five times faster performance on our largest benchmark codes.

1 Introduction

Rubber chemistry, although of tremendous economic importance, is not well understood. Current chemical formulations have been found by years of trial and error, guided by intuition and incomplete theory. A major research goal in this area is to develop models, based on quantum chemistry, to give a deeper understanding of rubber chemistry and to increase the ability to predict the properties of a particular rubber formulation. Unfortunately, a researcher is forced to manually construct systems of ODEs containing hundreds of equations and thousands to millions of floating point operations and variable references. This process can take months, and because of the detailed solution description required from the chemist, is extremely error prone, requiring additional weeks to verify that the proposed system is the same as the model represented by the system of ODEs. Once a model is proposed, simulating the system and computationally fixing unbound parameters in the system can take additional days.

In an ideal world, the chemist would describe reactions using a high level language in a matter of hours or a few days, a compiler would produce efficient code for the ODEs, and running the simulation and computationally fixing unbound parameters would take minutes or hours. The bottleneck in the process would be the chemist's creativity, not the drudge work of developing systems of equations followed by waiting for days for a computation to finish. This would allow a researcher to turn-around multiple solutions each week, dramatically increasing the productivity of the researcher, and greatly improving the rate of scientific progress. This paper describes a system that brings the chemistry researcher closer to this ideal world.

Our system is a compiler for a chemical reaction language and a runtime that allows researchers to provide a high level description of the reactions they wish to test, and then have the labor intensive and error prone ODE development phase of the cycle performed automatically. Our domain specific language, like many, produces as output a program in another relatively high-level language (C, in our case). As is often the case with machine generated code, the code that is naively generated by our compiler is more complex than that which is ever produced by a human, and sometimes exceeds the limits of the C Compiler, causing the compiler to fail. In particular, in our largest test case we produce basic blocks whose expressions contain approximately 3.3 million floating point operations. By employing

*This work is supported in part by the National Science Foundation under Grant Nos. CCF-0429535, and by the U.S. Department of Energy Catalysis Science Grant No. DE-FG0203ER15466A3. Supercomputing resources were supported by the National Science Foundation under the following NSF programs: Partnerships for Advanced Computational Infrastructure, Distributed Terascale Facility (DTF) and Terascale Extensions: Enhancements to the Extensible Terascalel Facility via Grant No. DMR050018P. Computing resources were also provided by Information Technology at Purdue (ITaP).

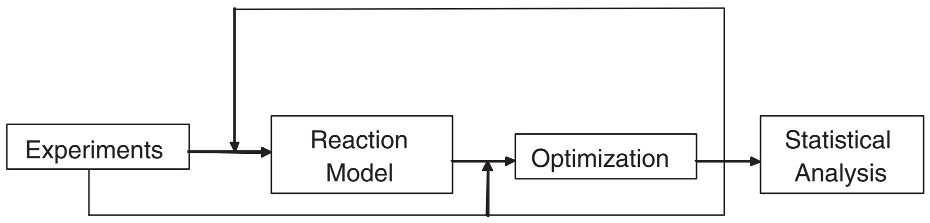

Figure 1. The work flow for understanding the chemistry behind compounds.

specialized algebraic and common sub-expression elimination algorithms we reduce the code size by eliminating all but 6.9% of the original arithmetic operations in the code, increasing performance by over five times. Finally, the code produced by the compiler will allow the resulting system of ODEs to be solved in parallel, allowing the computation time to be reduced to a matter of hours.

Figure 1 shows the workflow, consisting of collecting experimental data, determining the fit against (perhaps newly developed) possible reactions, optimizing the parameters of the model and testing the results, and statistically analyzing the results. Tweaking of the reaction model and optimization might need to be performed repeatedly until a good correlation with the experimental results is obtained. In particular the expression of a reaction model as a system of ODEs, and the linear optimization of the model to determine its correlation with experimental results, has been automated by our system. Because of the enormous size of the ODEs, manually generating and maintaining these equations is not possible for large and realistic reaction systems without help from a system such as ours.

This paper provides a high level overview of our system, and focuses on an extremely effective set of algebraic optimizations, which combined with a domain specific common subexpression elimination algorithm yields over a factor of 5 speedup performance improvement on realistically sized problems when compared to the results of a commercial compiler. Moreover, because of the complexity of the expressions to be optimized, the commercial compiler fails at high optimization levels. These optimizations, and the parallel code generated, provide researchers with rapid turnaround for their models, leading to large improvements in their effectiveness. To summarize the technical contributions of the paper:

- We describe our system for automating the testing of chemistry models;
- We describe the scalar algebraic optimizations and common subexpression elimination algorithms developed and implemented specifically for our system that remove redundant computations from the ODEs produced by our system;
- We describe the parallel runtime, developed and implemented specifically for our system, for quickly finding the optimized ODE solutions;
- We present experimental data, from a research project studying the vulcanization of rubber, that shows the effectiveness of our domain specific compiler, the sequential optimizations, and the parallel templates.

The rest of the paper is organized as follows. Section 2 provides an overview of the system, and describes the ODEs generated by the system before optimization. Section 3 describes the optimizations we perform to remove redundant computation from the system. Section 4 describes the parallel templates and library support that allow the efficient computation of the optimized ODEs' solutions. Section 5 gives our experimental results. Finally, Sections 6 and 7 describe related work and conclusions.

2 System overview

Our tool set, the Reaction Modeling Suite, is shown in Figure 2. The solid-line boxes are the components described in this paper. The first component is the *Chemical Compiler*, which accepts a high-level language specification of the chemical reactions using syntax similar to that of Prickett's Reaction Description Language (RDL) [11, 10, 9]. From the reaction descriptions, the chemical compiler automatically generates the reaction network that describes all possible reactions.

Each molecule specified can have variants that arise because many molecules differ from one another only in the lengths of chains of some atom (typically sulfur in rubbers.) Our input language allows all these variants to be expressed in a compact form which is then expanded by the chemical compiler. The input language is also used to specify the reactions among all of the input compounds. In general, rules can be generated for six reactions: (1) disconnect two atoms; (2) connect two atoms; (3) decrease the bond order between two atoms; (4) increase the bond order between two atoms; (5) remove a hydrogen atom; and (6) add hydrogen atoms. The language is rich enough to allow these rules to be applied with context sensitive knowledge,

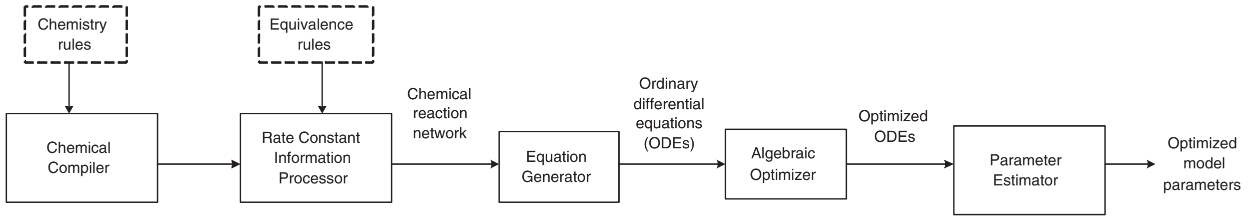
Chemistry rules
Equivalence rules
Chemical Compiler
Rate Constant Information Processor
Chemical reaction network
Equation Generator
Ordinary differential equations (ODEs)
Algebraic Optimizer
Optimized ODEs
Parameter Estimator
Optimized model parameters

Figure 2. The Reaction Modeling Suite.

$$\begin{array}{ll} 1. & -A + B + B \,\backslash\backslash\, [K_A]\,; \\ 2. & -C - D + E \,\backslash\backslash\, [K_{CD}]\,; \\ & \ldots \end{array}$$

Figure 3. The intermediate equations (i.e. the reaction network) generated by the chemical compiler.

e.g. to only break sulfur to sulfur bonds when the bonds are between sulfur atoms a least three atoms from the end of a chain of sulfurs. As well, certain actions and forms can be forbidden. Internally, molecules are stored and manipulated using the SMILES Java classes [1], a symbolic chemistry manipulation library. The output of the frontend is a set of expressions (called a reaction network) that represent the possible reactions, as seen in Figure 3. In each expression, $A, B, C, \ldots$ are molecules and radicals (henceforth simply referred to as molecules) being produced or consumed, where "-" indicates a consumed molecule, "+" indicates a produced molecule, and names beginning with "K" indicate a *kinetic rate constant*, as described below.

The second component is the *Rate Constant Information Processor* (RCIP). Kinetic rate constants are terms that describe the relative rates at which different reactions occur, and are set by the chemist, aided by the Gaussian '03 Quantum Chemistry Package [2]. Input data to the RCIP are expressions that define some constants as integer constants, and other constants as expressions of these integer constants. It then associates the rate constants with expressions in the reaction network, as shown in Figure 3.

The third component is the *Equation Generator*, which takes the reaction network created by the chemical compiler and generates ODEs to describe the reactions involving each variant of the molecules (see Figure 4). Equations are formed as follows. For each term T in the right hand side of the intermediate equations (e.g. A and B in Equation 1 of Figure 3) an equation with the left hand side of dT/dt is formed. The right hand side of the equation consists of the product of the rate constant for the intermediate reaction and each *reactant* term in the right hand side of the intermediate equation. The sign of the right hand side is the sign of T in the intermediate equation, thus if T is a reactant, the sign of the hand side is negative. After these equations are formed, the final ODEs (see Figure 5) are formed by summing all of the right hand sides of equations with the same left hand side. The resulting ODEs are shown in Figure 5.

The equation generator maintains an *equation table* that stores all of the ODEs that have been created. Every entry in this table represents one molecule, and consists of a doubly linked list of nodes, each representing one sum-of-products in the equation, broken down into individual terms.

The resulting ODEs contain many common terms which lead to significant amounts of redundant computation. Our optimizations, discussed in Section 3, remove these redundancies. The output from the Equation Generator is a C code function that that evaluates the ODEs.

The task of the fifth component, the *Parallel Parameter Estimator*, is to estimate values for the rate constants that are both consistent with quantum chemistry, and that allow the model to most closely match the experimental data. This component uses a parallel non-linear optimization of the ODEs' solutions, and by varying the rate constants it finds the closest match of the simulated model to experimental results. The closeness of this match serves as the quality metric for the accuracy of the model. This component uses all available processors to find a solution.

3 Optimizations

The output from the rate constant information processor is an exhaustive listing of all possible chemical reactions with regressed rate constants. The equation generator transforms these reactions into ODEs. The resulting ODEs have a large amount of computational redundancy – as we see in the experimental results section the single node running time experiences a five-fold increase in performance by removing these redundancies.

In this section we describe the optimizations performed by our system's algebraic optimizer to remove these redundancies.

3.1 Equation Simplification

The first optimization is to simplify the equations, in particular this optimization changes the equation:

1. $dA/dt = -K_A * A;$
2. $dB/dt = +K_A * A;$
3. $dB/dt = +K_A * A;$
4. $dC/dt = -K_C * D * C;$
5. $dD/dt = -K_C * D * C;$
6. $dE/dt = +K_C * D * C;$

$\ldots$

Figure 4. The initial set of ODEs produced from the reaction network of Figure 3.

1. $dA/dt = -K_A * A;$
2. $dB/dt = +K_A * A + K_A * A;$
3. $dC/dt = -K_C * D * C;$
4. $dD/dt = -K_C * D * C;$
5. $dE/dt = +K_C * D * C;$

$\ldots$

Figure 5. The final set of ODEs produced from the equations of Figure 4

.

$$\frac{dA}{dt} = 2 * k_1 * B * C + ... + 3 * k_1 * B * C + ...$$

into

$$\frac{dA}{dt} = 5 * k_1 * B * C + ...$$

After this optimization, each product in the sum-of-products representation of each molecule differs in at least one non-constant term from every other product. The transformation is performed on-the-fly as the equations are generated. When a sum-of-products is to be added to the linked list representing the equation for a reaction, it is combined, whenever possible, with another term that differs from it only in the constant terms.

3.2 Distributive Optimization

The second optimization is the distributive optimization. This optimization transforms the equation from:

$$\frac{dA}{dt} = k_1 * B * C + k_1 * B * D + k_1 * E * F \quad (1)$$

into

$$\frac{dA}{dt} = k_1 * (B * C + B * D + E * F) \quad (2)$$

and then into

$$\frac{dA}{dt} = k_1 * (B * (C + D) + E * F) \quad (3)$$

Figure 6 gives the algorithm for the transformation. We use the following expression to illustrate the action of algorithm.

$$k * p_1 + k * p_2 + \ldots + k * p_n + p_{n+1} + \ldots + p_m$$

The data structure P in lines 2 and 3 of the algorithm are created by the equation generator. The algorithm `DistOpt` performs the distributive optimization, and is invoked by line 3 for every equation E in the set of ODEs, terminates when all sums-of-products have been examined (the **while** loop condition of line 7), and returns the result of the optimization which is accumulated in P_{res}, and which is initially empty. The multi-set T holds the terms in the equation being optimized.

The algorithm finds the term k that appears in the most products, i.e. the term that appears most in T, and c, the number of times k appears. In the example above, k is that term.

If $c = 1$, then the most common term only appears once, and no optimization is possible. If $c > 1$, all of the products containing k are found (line 10), and the sum of those products with k factored out are multiplied by k (see line 11). This results in the equation above being rewritten as:

$$k * (p_1 + p_2 + \ldots + p_n) + \Gamma$$

which generates the intermediate result in Equation 2 above.

The algorithm is then applied to $k\ (p_1 + p_2 + \ldots + p_n)$ (the recursive call in line 11), and Γ (later iterations of the **while** loop at line 7). This generates the result in Equation 3 above. Thus the algorithm is applied to each created subterm until no further optimization is possible.

Finally, in line 12, P_k and k are subtracted from their respective sets so that they are not considered again.

Before applying the optimizations Equation 1 contains six multiplications and two additions. After applying the optimizations, as shown in Equation 3, there are only three multiplications and two additions.

3.3 Common Subexpression Elimination

After the distributive optimization, there are many common subexpressions both within and across equations. To remove this redundant computation we implement the third optimization, a form of common subexpression elimination (CSE). Our CSE optimization changes the equations from:

$$\begin{aligned}\frac{dA}{dt} &= ... + (A + B + C + D) * k_1 * E + \ldots \\ \frac{dB}{dt} &= ... + (A + B + C + D) * k_2 * F + \ldots \\ \frac{dC}{dt} &= ... + (A + B + C) * k_3 * G + \ldots\end{aligned}$$

```
for each ODE $E$
    $P$ = set of sum of products in $E$
    $P_{opt} = DistOpt(P)$

DistOpt($P$)
    $P_{res} = \emptyset$
    Let $T$ = terms($P$) /* $T$ is a multiset */
    while $P \neq \emptyset$
        $(k, c) = mostFrequent(T)$
        if $c > 1$
            $P_k = \{p : p$ a product $\in E, k \in p\}$
            $P_{res} = P_{res} \cup k \cdot DistOpt(\{\Sigma_{p \in P_k}(p/k)\})$
            $P = P - P_k$; $T = T - k$
        else
            $P_{res} = P_{res} \cup P$
            $P = \emptyset$
    return $P_{res}$
```

Figure 6. The Distributive Optimization Algorithm

into:

$$
\begin{aligned}
temp[0] &= A + B + C \ldots \\
temp[1] &= temp[0] + D \ldots \\
\frac{dA}{dt} &= \ldots + temp[0] * k_3 * G + \ldots \\
\frac{dB}{dt} &= \ldots + temp[1] * k_2 * F + \ldots \\
\frac{dC}{dt} &= \ldots + temp[1] * k_3 * G + \ldots
\end{aligned}
$$

The optimization, as described in the algorithm of Figure 7, proceeds as follows. First, the sub-expressions that form equations are stored in an indexed structure keyed on the sub-expression length (`exprList` in the algorithm.) The terms of each sub-expression are stored in a canonical lexicographical order – this allows an easy matching of expressions. The algorithm considers in turn each e_{long}, the longest sub-expression not yet processed (or one of the longest, if there are several sub-expressions whose length is equal to the length of e_{long}). This is done by the **foreach** loop of line 3 of the algorithm. The algorithm then examines all sub-expressions e_{leq} whose length is equal to the length of e_{long} (lines 4-6) or less than than the length of e_{long} (lines 7-11). When matching equal length expressions, any subexpression that matches e_{long} is replaced by e_{long}'s temporary. The replacement is done by the `replacePrefix` function. When matching a shorter expression to a prefix of a longer expression, the search is done from longest to shortest strings, since finding the longest matching prefix of e_{long} corresponds to finding the most redundancy in the computation. The matching is facilitated by the canonical order imposed on the terms making up each temporary. When a match is found, the prefix of e_{long} is replaced by the shorter expression's (E_{leq}) temporary, and the search stops. We note that the `replacePrefix` function also marks the `genTemp` bit of the subexpression whose temporary is to be used, ensuring that an assignment into that temporary of the subexpression is performed.

An *Expression* is a structure with three fields

expr:	the expression
lhs-list:	list of left-hand sides this expression is assigned to
temp:	a temporary name the *expr* can be assigned to
genTemp:	a boolean – initially **false**, set to **true** if *expr* should be assigned to temp in final code.
exprList[1:maxLen]:	an array of lists of expressions
maxLen:	the maximum number of terms in any *expr*.
exprList[i]:	list of all *expr* with i terms

```
1.  $len = maxLen$
2.  while ($len > 1$)
3.      foreach $E_{long} \in exprList[len]$
4.          foreach $E_{leq}$ in $exprList[len]$that is not $E_{long}$
5.              if matchesPrefix($E_{leq}.expr, E_{long}.expr$)
6.                  replacePrefix($E_{leq}, E_{long}$)
7.          for ($i = 1; i < len; i++$)
8.              foreach $E_{leq}$ in $exprList[len]$that is not $E_{long}$
9.                  if matchesPrefix($E_{leq}.expr, E_{long}.expr$)
10.                     replacePrefix($E_{long}, E_{leq}$)
11.                     break
12. for ($i = 1; i < maxLen; i++$)
13.     foreach $e \in exprList[i]$ such that $e.genTemp$)
14.         genAssign($e.temp, e.expr$)
```

Figure 7. Common Sub-Expression Optimization Algorithm.

When emitting code, assignments to temporaries are generated (lines 13 and 14) before reads of that temporary. Within common sub-expressions of the same length, the temporary assignments are generated before any other assignments, including assignments using the temporary. For common sub-expressions of different lengths, the shorter common sub-expression is always generated first, thus ensuring the value is written before being used in a longer sub-expression.

Because the terms in all expressions are in a canonical lexicographical order, and all like terms have been gathered in the distributive optimization, the number of comparisons between two expressions is the number of expression terms of the longer expression in the worst case. As a result, the asymptotic time complexity of this algorithm is $O(m^2n)$, where m is the number of expressions, and n is the average number of expression terms of each expression. We note that this is better than other string matching algorithms which cannot make these assumptions because of the nature of their input. These algorithms require, for example, time proportional to the terms in both expressions [6].

Our CSE algorithm differs from the standard algorithms in the literature (e.g. [13]) in several ways. First, because we control the generation of our code, we know that uses of variables are not aliased, are only written once (per iteration of the ODE solver loop), and because the values are floating point numbers, pairs of variables with different names are less likely to have the same value than are integer variables. Moreover, those variables with different names most likely to have the same value, i.e. the rate constants, have been renamed based on common values by the rate constant information processor discussed earlier. Therefore, we can use the *name* of a variable to label its *value*, simplifying our algorithm. In the final code for the system of ODEs the left and right hand sides of the ODEs could appear to be aliased to the target C compiler, preventing the target C compiler from optimizing these expressions. Finally, our CSE optimization exploits the structure of the code in optimizing expressions. With any CSE optimization it is necessary to choose which, of many possible expressions, to optimize. For example, given the expressions $d(a + b + c)$, $d * a + d * c$ and $a + c$ and $a + b$, general CSE algorithms will catch some of the redundancy, but not all, since forming all possible representations of these values will be very time consuming. Because of the structure of our equations and constituent expressions, we know that a canonical fully non-distributed representation is best, and represent all expressions this way. Our CSE optimization only uses $O(mn)$ storage for bookkeeping, where m is the number of expressions, and n is the average number of terms in each expression. Just as importantly, our techniques greatly reduce the size of the code going into the more general optimizing compiler, allowing much less storage to be needed for the richer, general IR used by these compilers, and allowing them to compile the C code representation of a larger set of ODEs used in our simulations. As shown in the experimental results section, we can compile programs at least 10 times larger using our optimizations that when not using them.

4 Parallel Parameter Estimator

The optimized ordinary differential equations are passed to the *ODE Solver* and *non-linear optimizer* routines where the system solves for the kinetic parameters. Before passing the ODEs to the runtime, it is necessary for the chemist using the system to set bounds on the different kinetic parameters. These bounds are used to constrain the possible solutions found by the non-linear solver. The ODE solver is used to find the solutions to the set of ODEs, and the resulting solution is passed to the non-linear solver to find the tightest fit to the experimental results. If a tight correlation exists between the runtime result and the experimental results, the model expressed in the original equations and kinetic bounds is assumed to be a good estimator of the chemical reactions for the set of inputs under investigation.

4.1 The ODE Solver

For our ODE solver we use the IMSL libraries for AIX. The IMSL C library provides two ODE solvers, a Runge-Kutta solver and an Adams-Gear solver. The Runge-Kutta solver (`imsl_f_ode_runge_kutta`) solves an initial value problem for ordinary differential equations using the Runge Kutta Verner fifth order and sixth order method. This function is efficient for non-stiff systems. The Adams-Gear solver (`imsl_f_ode_adams_gear`) solves a stiff initial value problem for ordinary differential equations. Because chemical reactions proceed to equilibrium, where molecules and their variants effectively complete their reactions in different epochs, the differential equations modeling the behavior of such systems are stiff. Therefore we use the Adams-Gear solver.

4.2 The Non-linear Optimizer

The IMSL C library also provides constrained minimization functions. We use the non-linear least squares with simple variable bounds algorithm (`imsl_f_bounded_least_squares`). This function uses a modified Levenberg-Marquardt [7, 8] method and an active set strategy to solve the non-linear least squares problems subject to simple bounds on the variables. Figure 8 shows the set-up of arguments and the call to this routine used in our system. The lower bounds, upper bounds and initial values for the kinetic rates are set, and the optimization function is called to optimize the problem, as described next. The values returned by the non-linear optimizer will be the optimized values of the kinetic rate constants: they will both be within the constraints set by the chemist, and the values that provide the closest match to the experimental results.

```
int main(int argc, char *argv[ ]) {
    /* initialize */
    set xlb[ ] lower bounds of kinetic parameters;
    set xub[ ] upper bounds of kinetic parameters;
    set xguess[ ] initial guess values of kinetic parameters;

    /* call optimization */
    optimized_results =
        imsl_f_bounded_least_squares(void objective_fcn(),int m,
            int n, float xlb[ ], float xub[ ], float xguess[ ] ...);
    return 0;
}
```

Figure 8. Code that Invokes the Nonlinear Optimizer.

4.3 Objective Function

The objective function, shown in Figure 9, is the input to the optimization function, and allows the optimality, or closeness, of the ODE solution for a set of kinetic values to the experimental result to be determined. There are three major inputs to the optimization function. The first is the kinetic rate constants mentioned above. The second is the set of ODEs generated by the compiler. The third is a set of files that contain experimental data. Each file contains more than 3000 records of the form $< t_i, \textit{property value} >$, where t_i is a time step and *property value* is a measure of the property that is to be predicted by the chemical model (e.g. elasticity or stiffness of the rubber compound). The ODE solver function is called to calculate the simulated property values. The difference between simulated data values and experimental data values is stored into `error_vector[]`. Multiple files are used at runtime to provide the results of multiple experiments. Each file produces a *local* `error_vector[]`, with a sum reduction being done overall of these vectors to produce a *global* `error_vector[]`.

4.4 Parallel Computation of the Optimized Solution

Because of the complexity of these reactions (real systems have thousands of equations), solving the ODEs and optimizing the kinetic parameters is computationally expensive. As seen above, the objective function opens multiple experimental data files and reads them one by one. The solver consumes most of the computation time in the system (99% of the time in a 16 data file run) and an even larger percentage with more files. Because our goal is to allow a chemist to quickly try out many different models (e.g. kinetic rate constant bounds) for a calculation, it

```
#include <mpi.h>
object_function(int m, int n, float rate_constant[],
        float global_error_vector[]) {
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);

    calculate the number of data files;
    size = BLOCK_SIZE();
    for(i=0; i<size; i++) {
        /*get information from experimental data files*/
        open this data file;
        get number of time steps from this data file;
        get data values at each time steps from this data file;

        /* initialize ODE solver */
        imsl_f_ode_adams_gear_mgr(IMSL_ODE_INITIALIZE,
                &state,...);
        for(j=0; j<number_timestep; j++) {
            /* integrate from t to tend */
            simulated_value = imsl_f_ode_adams_gear(&t, tend,
                    state,ode_fcn, . . . );

            /*calculate the difference between two results*/
            error_vector[j] += function(simulated_value,
                    experimental_value);
        }

        /* end ODE solver function*/
        imsl_f_ode_adams_gear_mgr(IMSL_ODE_RESET, &state, . . . );
        record time for solving this data file;
    }
    MPI_AllReduce(error_vector, global_error_vector, max_steps,
            MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
    MPI_AllReduce(local_time, global_time, number_datafile,
            MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
    apply dynamic load balancing algorithm;
}
```

Figure 9. The Parallel Objective Function using MPI.

is essential for the utility of the system that it provides a solution as quickly as possible. We therefore parallelized our system using MPI, as shown in Figure 9. The number of concurrent processes is specified when the program begins, and remains constant throughout the execution of the program. Each process potentially opens several different data files – the data files are replicated across the processors – and computes the error (of the ODE solution relative to the experimental data solution) and places it into `error_vector[]`. Next, `MPI_AllReduce()` is used to reduce the errors across the different data files into `global_error_vector[]`, which is then sent to every process.

In order to reduce the idle time and improve the efficiency of processors, we also implement a dynamic load balancing algorithm. The aim of load balancing is to optimize the workload distribution, so every processor get equal amount of workload. In our dynamic load balancing algorithm, the time to solve each data file is recorded and put into a priority queue built out of a non-increasing sorted time list. The next item, which corresponds to the data file with the largest solving time among remaining data file in the priority queue, is allocated to the processor with least total allocated time so far. As a result, at the next objective function call, every processor will receive the balanced workload calculated by the current objective function call. Because the data files are replicated, each processor has the data necessary to compute the workload it is assigned.

5 Experimental Results

5.1 Benchmark programs

The system is being actively used within our group to model the vulcanization process of rubber. Therefore, the input for our experiments are the kinetic models for the vulcanization process implemented for the vulcanization of natural rubber by the benzothiazolesulfenamide class of accelerators. There are five test cases with different sets of chemical equations, but the same 10 distinct kinetic parameters. Experimental data is used to determine the kinetic parameters and validate the predictions of the reaction model. Measurements of different rubber formulations have been employed to determine the time evolution of the concentration of crosslinks – strands of sulfur that link rubber molecules and therefore affect the stiffness of the rubber – for different formulations cured at different temperatures. We tested this reaction model using 16 experimental data files, which contain the time evolution of the crosslink concentrations for different formulations at the same temperature. The total execution time of the runtime phase is recorded to evaluate the performance. The models used were developed as part of an ongoing project in predicting the properties of rubber compounds.

By automatically generating these ODEs, our compiler now does in minutes or hours what used to take researchers days or months to accomplish.

5.2 Experiment Environment

We performed our experiments on a 320 processor IBM RISC System/6000 POWER Parallel System (SP) with 375 mhz processors located at Purdue University. The 320 processors are divided among 64 *thin nodes* (quad-processor systems with 4.5 GB of memory) and 4 *high nodes* (16-processor systems with 64 GB of memory). Both Ethernet and IBM's proprietary switches connect the nodes. For our experiments we used the faster IBM proprietary switch and thin nodes, and used one processor per node. The compiler is mpCC_r, which uses the AIX xlc compiler, version 6.0.0.0. The options used were "-O4 -qmaxmem=-1" to allow maximum memory to be used by the compiler during the compilation. If compiler failed at this level of optimization, we reduced the optimization level from "O4" to "O3", and on down to the default optimization level until the compilation succeeded, or failed at all optimization levels.

5.3 Results of Algebraic Optimization

We first measure the single node performance of our compiler optimizations using one processor of one node on the IBM/SP. The results shown in Table 1 demonstrate the performance improvements using different optimization combinations. Note that we cannot run the CSE optimization without first running the algebraic optimizations. As can be seen, the program encounters a compiler error when running test case 5, which has the largest set of chemical equations. The error message given is "Compilation ended due to lack of space", which results from more than 4.5GB of memory being needed by the compiler. When we turn on the optimization options (-O4) in the compiler, the program encounters the same compiler error after test case 3. The compiler can compile test case 2 with the highest optimization level, and the resulting code, without any of our optimizations, runs in 82% of the time of the unoptimized program.

As Table 1 shows, our algebraic and CSE optimizations greatly reduce the size of chemical equations. In test case 5 (the largest), the number of multiplies is reduced to 1.35% of the original number, and the number of addition and subtraction operations is reduced to 20.6% of the original number, with overall arithmetic operations being reduced to 6.9% of the original number. Test case 4 achieved a speedup of 5.26, and we could not measure the speedup for test case 5 because the program would not compile (because of a lack of space) without our algebraic optimizations. The significance of the Algebraic Optimizer and Common Subexpression Elimination algorithms is both that the execution time is reduced greatly, and users can create and test the very complex reaction models necessary to simulate real chemical reaction systems like (like test case 5), which could otherwise not be compiled and run.

5.4 Results of Parallel Computation Using MPI

Table 2 demonstrates the performance using different numbers of nodes. Only one processor is used in each node. As the number of nodes increases, the execution time decreases, and the speedup increases. This speedup is over and above the speedups that were obtained from the se-

	Test case 1	Test case 2	Test case 3	Test case 4	Test case 5
Number of Equations	450	10000	24500	125000	250000
Number of * without algebraic/CSE optimizations	2670	85500	229000	1320000	2400000
Number of (+ and -) without algebraic/CSE optimizations	1770	36600	94800	520000	974000
Execution Time (seconds) without algebraic/CSE optimizations	924	4290	7480	42800	compiler error
Execution Time (seconds) with C compiler optimizations only	920	3530	compiler error	compiler error	compiler error
Number of * (with algebraic/CSE optimizations, no C compiler optimizations)	629	7450	11800	22000	32400
Number of (+ and -) (with algebraic/CSE optimizations, no C compiler optimizations)	761	22800	56800	125000	201000
Execution Time (seconds) (with algebraic/CSE optimizations, no C compiler optimizations)	824	2500	4240	8130	15459

Table 1. Results in IBM/SP Using Different Optimization Combination

Number of nodes	Total Time (seconds) Without Dynamic load	speedup Without Dynamic load	Total Time (seconds) With Dynamical load	speedup With Dynamical load
1	15459	1	15459	1
2	7619	1.99	7784	2.03
4	3874	3.91	3598	3.99
8	1935	7.08	2183	7.99
16	1210	12.78	1210	12.78

Table 2. Results in IBM/SP Using MPI

quential optimizations. Since 16 data files are used in all test cases, each processor handles one data file when using 16 nodes, it cannot be further parallelized. Speedup in 16 nodes is only 12.78, the result of a load imbalance caused by different amounts of work involved in processing each file. After applying our dynamic load balancing algorithm, the speed up is nearly linear. We note a small super linear speedup at two nodes which we attribute to either measurement noise or memory effects. At 16 nodes, there is only one task to schedule per processor, so the load balancing algorithm has no affect, causing the performance of the load balanced and non-load balanced runs to be identical.

6 Related Work

With improvements in computational power, interest in exploiting computers to design new materials increased dramatically. Katare et al. [4, 5] applied a reaction modeling suite to material design applications in catalysis, polymers and fuel additives. They employed hybrid optimization heuristic techniques like genetic algorithms to design better additives for fuels. Ghosh et al. [3] showed the feasibility of computer aided design of rubber formulation through reaction kinetic modeling.

Prickett et al. [11, 10, 9] designed a computer language to describe general types of reactions. Their *Reaction Descriptive Language* (RDL) can be used to describe various reaction networks, with reaction classes being defined by sequence of commands to locate the reaction site and manipulate the reactants to form the product. The syntax of RDL has been adopted in our reaction compiler.

The Chemistry Development Kit (CDK) is a freely available open-source Java library for Structural Chemo- and Bioinformatics. It is now supported by more than 20 developers world-wide. The CDK provides methods for many common tasks in molecular informatics, including 2D and 3D rendering of chemical structures, I/O routines, SMILES parsing and generation, ring searches, isomorphism checking, structure diagram generation, etc. [1] We use the CDK library to handle molecule manipulation operations in our chemical compiler.

There are two broad categories of CSE elimination techniques: partial redundancy elimination and value numbering. Partial redundancy elimination does not consider the possibility of reordering expressions. Instead, they work with expressions available in 3-address code [14]. Value numbering accomplishes a form of symbolic execution and is capable of finding a much larger class of redundancies. however, the problem of obtaining optimal results (even if restricted to basic blocks) is NP-complete [13].

In [15], Xiong et al. describe the SPL component of the Spiral [12] library generator. SPL is a domain specific language for developing signal processing codes. They show that common subexpression elimination is important in this context for good performance. In SPL a value numbering algorithm is used, but unlike our algorithm does not appear to be tuned specifically for the programs being compiled.

7 Conclusions

Our domain specific chemistry compiler allows a chemist to generate a high-level and easily debugged description of a reaction that can be quickly converted into a set of ODEs. This saves weeks of time in creating this set of equations. Because we exploit existing library support and a parallel template we are able to quickly provide feedback as to how well the model proposed by the chemist matches existing experimental data, allowing the chemist to quickly explore alternate models if necessary. Moreover, because of domain specific information about the structure of the computation, efficient node code and effective parallelization is straightforward, and guarantees good parallel performance. The framework should dramatically increase the productivity of chemists, and aid in the rapid advances in the production of new materials.

References

[1] CDK. http://cdk.sourceforge.net/api/, 2004.

[2] Gaussian '03 web page. http://www.gaussian.com/g03.htm, last checked Oct. 13, 2005.

[3] P. Ghosh, S. Katare, P. Patkar, J. M. Caruthers, V. Venkatasubramanian, and K. A. Walker. Sulfur vulcanization of natural rubber for benzothiazole accelerated formulations: From reaction mechanisms to a rational kinetic model. *Rubber Chemistry and Technology*, 76(3):592–693, July-August 2003.

[4] S. Katare, J. M. Caruthers, W. N. Delgass, and V. Venkatasubramanian. An intelligent system for reaction kinetic modeling and catalyst design. *Industrial and Engineering Chemistry Research*, 43(14):3484–3512, July 7, 2004.

[5] S. Katare, P. Patkar, P. Ghosh, J. M. Caruthers, W. N. Delgass, and V. Venkatasubramanian. A systematic framework for rational formulation and design of materials. In J.-C. Zhao, M. Fahrmann, and T. Pollock, editors, *Materials Design Approaches and Experience*, pages 321–332. The Minerals, Metals & Materials Society, 2001.

[6] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):322–350, 1977.

[7] K. Levenberg. *A Method for the Solution of Certain Problems in Least Squares*. Quart. Appl. Math., 1944.

[8] D. Marquardt. *An Algorithm for Least Squares Estimation on Nonlinear Parameters*. J. APPL. MATH., 1963.

[9] S. Prickett and M. Mavrovouniotis. Construction of complex reaction syatems-ii. an example:alkylation of olefins. *Computers and Chemical Engineering*, 21(11):1325–1337, 1997.

[10] S. Prickett and M. Mavrovouniotis. Construction of complex reaction syatems-ii. molecule manipulation and reaction application algorithms. *Computers and Chemical Engineering*, 21(11):1237–1254, 1997.

[11] S. Prickett and M. Mavrovouniotis. Construction of complex reaction systems-i. reaction description language. *Computers and Chemical Engineering*, 21(11):1219–1235, 1997.

[12] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Int. J. High Perform. Comput. Appl.*, 18(1):21–45, 2004.

[13] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, New York, NY, USA, 1988. ACM Press.

[14] A. Sorkin. Some comments on "A solution to a problem with Morel and Renvoise's 'Global optimization by suppression of partial redundancies' ". *ACM Trans. Prog. Lang. Syst.*, 11(4):666–668, October 1989.

[15] J. Xiong, J. Johnson, R. Johnson, and D. Padua. Spl: a language and compiler for DSP algorithms. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 298–308, New York, NY, USA, 2001. ACM Press.