

12-1-2005

A language and compiler for enabling automatic and parallel chemistry simulations

Jun Cao

Ayush Goyal

Sam P. Midkiff

Y. Charlie Hu

James M. Caruthers

Follow this and additional works at: <http://docs.lib.purdue.edu/ecetr>

Cao, Jun ; Goyal, Ayush ; Midkiff, Sam P. ; Hu, Y. Charlie ; and Caruthers, James M. , "A language and compiler for enabling automatic and parallel chemistry simulations" (2005). *ECE Technical Reports*. Paper 71.
<http://docs.lib.purdue.edu/ecetr/71>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

A LANGUAGE AND COMPILER FOR
ENABLING AUTOMATIC AND
PARALLEL CHEMISTRY
SIMULATIONS

JUN CAO
AYUSH GOYAL
SAM P. MIDKIFF
Y. CHARLIE HU
JAMES M. CARUTHERS

TR-ECE – 05-18
DECEMBER 2005

PURDUE
UNIVERSITY

SCHOOL OF ELECTRICAL
AND COMPUTER ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, IN 47907-2035

A language and compiler for enabling automatic and parallel chemistry simulations

Jun Cao, Ayush Goyal, Sam P. Midkiff, Y. Charlie Hu, and James M. Caruthers
Purdue University, West Lafayette, IN 47907
{caoj, ayush, smidkiff, ychu, caruther}@purdue.edu

Abstract

The modeling of chemical reactions, and the ability to predict the properties of the end-products of a chemical reaction, is of extreme commercial importance. The properties of many compounds have complex dependencies on a variety of additives, in ways that are not well understood. This paper describes a domain specific language, compiler and parallel runtime system that allows chemists to investigate, and understand how, different additives affect these properties. In particular, our system allows the inputs and types of reactions that are possible to be specified in a high level language. It then produces a set of ordinary differential equations (ODEs) that when combined with boundary conditions from quantum chemistry are processed using parallel templates and off-the-shelf solvers to simulate the reaction. This paper describes the complete system, including optimizations to reduce the amount of redundant computation in the ODEs, the parallel templates for simulating the reaction, and experimental data showing the effectiveness of these. Our system saves the chemist from manually developing, testing and debugging systems of hundreds or thousands of ODEs that require weeks or months to develop.

1 Introduction

The modeling of chemical reactions, and the ability to predict the properties of the end-products of a chemical reaction, is of extreme commercial importance. Unfortunately, for many economically important compounds, no “closed form” solution of the outcome of a reaction is possible. A common experimental paradigm is for the chemist to formulate a compound, create a model for the underlying chemistry, and then see how well the properties predicted by the model and properties of the formulated compound match. The model, based on quantum chemistry, is *manually* developed and consists of large complex systems of ordinary differential equations (ODEs). These systems can have hundreds or thousands of differential equations and take days to months to develop and debug. Each iteration over the “model development, ODE development, result evaluation” is labor intensive and fraught with error. As well, simulating the reaction by solving the system of ODEs is computationally intensive.

This paper describes a compiler for a chemical reaction language and a runtime that allows researchers to *automate* the labor-intensive and error prone ODE development phase of the cycle, and to automatically solve the resulting system in parallel, allowing the cycle time to be reduced to a matter of hours. Figure 1 shows this workflow. 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. Our compiler and runtime targets general

chemical reactions, however in our group it is being used primarily to simulate the chemistry of rubber compounds.

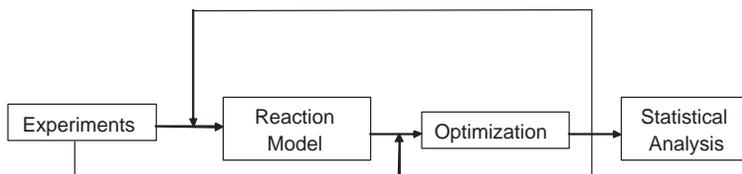


Figure 1: The work flow for understanding the chemistry behind compounds

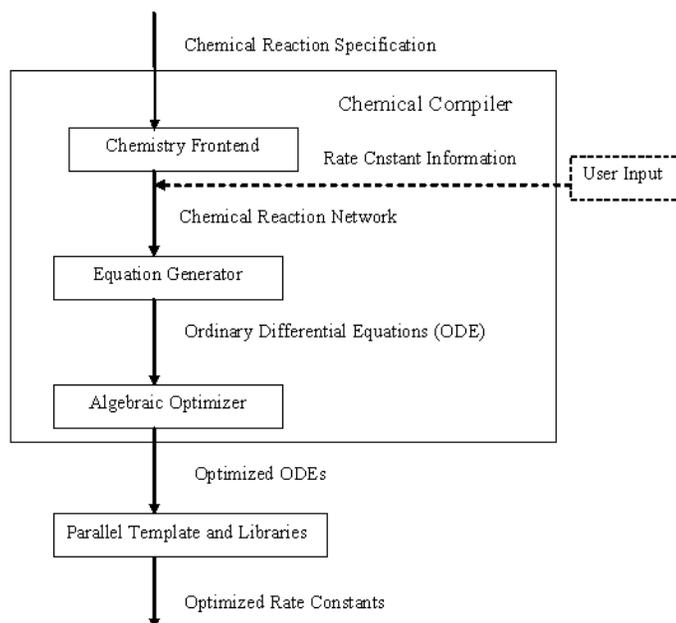


Figure 2: a Reaction Modeling Suite

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 *Chemistry Frontend*, which accepts a high-level language specification of the chemical reactions and from these 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, e.g., chains of sulfur atoms. Our input language allows all these variants to be expressed in a compact form which is then expanded by the chemistry frontend. The input language is also used to specify the reactions among all of the input compounds.

The second component is the *Equation Generator*, which accepts the reaction network created by the Chemistry Frontend and generates ODEs to describe the reactions involving each variant of the molecules. These ODEs contain *kinetic rate constants* which are terms that describe the relative rates at which different reactions occur, and are set by the chemist, aided by Gaussian '03 Quantum Chemistry Package [2]. These rate constants serve as constraints on the possible boundary conditions of the ODEs. The resulting ODEs contain many common terms which lead to

significant amounts of redundant computation. The *Algebraic Optimizer* removes these redundant calculations in the ODEs, and increases the performance of the following computational stages.

The fourth component is *parallel template and library support*. This component uses non-linear optimization of the ODEs’ solutions, by varying the rate constants, to find 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.

Many compounds, including rubbers, have complex sets of reactions that occur at different rates and that lead to the final, economically useful end product. By adjusting the ratios and types of additives the properties of the end product can be dramatically changed. Unfortunately, the chemistry of these compounds is not yet well enough understood to allow properties to be predicted based on the knowledge of the inputs to the reaction. Our compiler is a crucial research tool for allowing chemists to rapidly gain deep insights into the underlying chemical processes by automating the work of generating and solving the system of ODEs representing the reaction. In doing so, it reduces a task that formerly took hundreds to thousands of hours of the time of a skilled chemist to one that can now be done in hours.

This paper makes the following technical contributions:

- We describe our system for automating the testing of chemistry models, and do so in terms understandable to computer scientists;
- We describe scalar algebraic optimizations developed and implemented for our system that remove redundant computations from the ODEs produced by our system;
- We describe the parallel templates, developed and implemented 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 describes the input language and the generation of all possible intermediate compounds given the input compounds and allowed reactions. Section 3 describes the generation of ODEs to describe these reactions, and the optimization of these ODEs to remove redundant computation. 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 The Chemistry Frontend

The chemistry frontend parses a program written in a language that describes sets of molecules and possible reactions involving those molecules. The input program for the compiler provides descriptions of molecules that are inputs to the chemical reactions, and how these compounds can interact. In this section we will describe, using examples, the language and the actions taken by the frontend on programs written in this language.

In Figure 3 different oxygen molecules and their reactions are described, as well as dissociation reactions and combination reactions. The first section describes two input molecules, which have been labeled *Per* and *Hydro*. Atoms are denoted by their atomic symbol in brackets, e.g. “[Au]”

for gold. For the organic subset of the elements, *C*, *N*, *O*, *P*, *S*, *Br*, *Cl* and *I*, the brackets may be omitted. The first molecule described is the molecule $H - O - O - H$ (two hydrogens and two oxygens). Note that hydrogen atoms are not specified, and are assumed to exist wherever necessary to fill electron holes. The second molecule described is $H - O^*$ – a hydroxide radical¹. That this is a radical is indicated by the “*” subscript on the oxygen symbol (“O”). The sequence “1,1” says that there is exactly one of the atom in position 1 – by changing this to, e.g. “5,1”, we could denote a chain of from 1 to 5 of the atom that is given in first position. This allows “families” of molecules to be compactly expressed. The notation also allows different bonds (e.g. single, double, and aromatic) and rings to be described. The notation is general enough to allow any chemical compound to be described.

The *dissociation reaction* description tells how the molecules can be broken down during the chemical reaction. The dissociation reaction in Figure 3 says that the “oo” (really $H - O - O - H$) molecule *Per* can be broken into two “ $H - O^*$ ” radicals. The phrase *require r1 reactant(Per)* says that a *Per* molecule must be used, and will be referred to as *r1*. The phrase *(label-site a1 atom(find neutral O belong-to r1))* identifies a neutral (non-radical) oxygen, and labels it *a1*. The phrase *(disconnect a1 a1)* says that the bond between two of these neutral oxygens can be broken. Finally, the phrase *(assign rateconstant(Kd))* says that the rate at which the reaction occurs will be governed by the rate constant *K*, and that this is a dissociation (*d*) reaction.

Similarly, rules for combining molecules can be specified, as seen in lines 11-15. Essentially, a reaction between a methyl and an ethyl molecule is specified (lines 10 and 11), with the radical carbon in the methyl molecule (line 12) and the radical carbon in the ethyl molecule (line 13) binding to one another (line 14.) The bonding of these two forms propane, but that propane is the end result of the reaction is not stated explicitly: it is the task of the chemistry frontend to determine this given the list of all possible molecules (for the reactions under investigation) expressed by the input. Line 15 shows the rate reaction constant for formation of this bond is *K*, and the “*b*” indicates that this is the constant for a combination (or bond forming) reaction.

Other reactions can be specified in the elided code (indicated by the ...). For example a reaction making use of the “ $H - O^*$ ” radicals formed in lines 6-9 have been elided. In general, rules can be generated for six actions:

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;
6. add hydrogen atoms.

As well, certain actions and forms can be forbidden. Phrases also exist to locally assign identifiers to different atoms within a molecule to allow them to be referred to in later rules. Internally, molecules are stored and manipulated using the SMILES Java classes [1] – a symbolic chemistry manipulation library.

¹Intuitively, a *radical* is a charged molecule, i.e. a molecule with empty electron positions in its outer shell.

```

/* molecule representation */
1. molecule_input_temp1.input("oo", "Per", 1, 1, molecule_table);
2. molecule_input_temp1.input("o*", "Hydro", 1, 1, molecule_table);
3. molecule_input_temp1.input("c*", "Methyl", 1, 1, molecule_table);
4. molecule_input_temp1.input("cc*", "Ethyl", 1, 1, molecule_table);
5. molecule_input_temp1.input("ccc", "Propane", 1, 1, molecule_table);

/* dissociation reaction */
6. {(require r1 reactant(Per))
7. (label-site a1 atom(find neutral O belong-to r1))
8. ((disconnect a1 a1)
9. (assign rateconstant(Kd)))}

...

/* combination reaction */
10. {(require r1 reactant(Methyl))
11. (require r2 reactant(Ethyl))
12. (label-site a1 atom(find radical C belong-to r1))
13. (label-site a2 atom(find radical C belong-to r2))
14. ((connect a1 a1)
15. (assign rateconstant(Kb)))}

...

```

Figure 3: The input to the chemistry frontend. Line numbers are not part of the input

```

1.  $-Per + Hydro + Hydro \setminus [KdPer]$ ;
2.  $-Methyl - Ethyl + Propane \setminus [KbMethylEthyl]$ ;
...

```

Figure 4: The intermediate equations (i.e. the reaction network) generated by the chemistry frontend

1. $dper/dt = -KdPer * Per;$
2. $dHydro/dt = +KdPer * Per;$
3. $dHydro/dt = +KdPer * Per;$
4. $dMethyl/dt = -KbMethyl * Ethyl * Methyl;$
5. $dEthyl/dt = -KbMethyl * Ethyl * Methyl;$
6. $dPropane/dt = +KbMethyl * Ethyl * Methyl;$
- ...

Figure 5: The initial set of ODEs produced from the reaction network of Figure 4

Lines 1 and 2 of Figure 4 show the specified reactions expressed as equations. These equations will be used by the *Equation Generator* of the compiler to form the system of ODEs for the reaction. Each equation has a negative term corresponding to the molecules that are consumed in the given reaction and a positive term for the produced molecules. Thus, in equation 2, methyl and ethyl are consumed and propane is produced. The rate constant for the reaction is also specified. Because the rate constant names are overloaded (i.e. in both the combining and dissociation reactions of Figure 3 the reaction constant is named K) they are disambiguated by appending both the *type* of the reaction (b or d) and the required reactants (Methyl and Ethyl in the case of the combining reaction) to the rate constant name.

3 Equation Generator and Algebraic Optimizations

The output from the chemistry frontend is an exhaustive listing of all possible chemical reactions. The equation generator transforms these reactions into ODEs. The resulting ODEs have a large amount of redundancy – as we see in the experimental results section up to 19% of the single node running time can be eliminated 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 Generator

The set of equations (or reaction network) produced by the chemistry frontend, and describing all chemical reactions possible with the original input, is the input to the equation generator. The equation generator transforms these chemical reactions into ODEs, and groups ODEs belonging to the identical molecules together. We will now explain this process by means of an example.

The equation generator transforms the chemical reactions described in Figure 4 into the following set of ODEs found in Figure 5.

From the first equation of Figure 4 we get equations 1, 2 and 3 of Figure 5. These equations are formed as follows. For each reaction specified in the input, there are a set of equations in the reaction network. Lines 6-9 and 10-15 in Figure 3 specify two reactions, and lines 1 and 2 of Figure 4 are the equations for each reaction, respectively. Each reaction specifies a reactant in the first lines: (Per and $Methyl, Ethyl$) respectively for the two reactions. The right hand side of the equation for dm/dt , where m is some molecule m involved in the reaction, is the product of every reactant term in the equation for the reaction, times the rate constant for the reaction. The sign of the right hand side terms comes from the sign in the reaction network equation. Thus equation

1. $dPer/dt = -KdPer * Per;$
2. $dHydro/dt = +KdPer * Per + KdPer * Per;$
3. $dMethyl/dt = -KbMethyl * Ethyl * Methyl;$
4. $dEthyl/dt = -KbMethyl * Ethyl * Methyl;$
5. $dPropane/dt = +KbMethyl * Ethyl * Methyl;$
- ...

Figure 6: The final set of ODEs produced from the equations of Figure 5

```

struct equation_item {
    int number_reactant; // is 1, for the 1 reactant Per
    string symbol; // is "-", for the leading term
    double rate_constant_coeff; // is 1, the coefficient of KdPer
    string rate_constant; // is KdPer
    string reactant[MAX_NUMBER_REACTANT]; // is ["Per"], the reactant name(s)
    struct equation_item *prenode; // ptr to the previous term
    struct equation_item *sucnode; // ptr to the next term
};
struct equation_item *equation_table[number_species];

```

Figure 7: The data structure that holds a term in the ODE description

1 in Figure 4 gives rise to equations with left hand sides $dPer/dt$ and $dHydro/dt$, with right hand sides that are the product of all reactions involved in equation 1. These equations are shown in lines 1–3 of Figure 5. Note that because *Hydro* appears twice in the reaction network equation, there are two equations with it on the left hand side in Figure 5.

After these equations are formed, the final ODEs 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 6.

3.2 Equation Simplification

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 a term, as shown in Figure 7. We describe the value of the fields for the term “ $-KdPer * Per$ ” from either the first, second or third equation in Figure 5. This table facilitates the four optimizations described next by keeping, for each term in the equation the molecule name, the rate constant and its coefficient and the reactants.

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

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

into

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

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 new term is added to the linked list representing the molecule, it is combined with other terms where possible.

3.3 Distributive Optimization

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

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

into

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

and then into

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

Before the optimization, there are six multiplications and two additions in the equation, while afterwards there are only three multiplications and two additions. To perform this optimization on the equation:

$$k * \Pi_1 + k * \Pi_2 + \dots + k * \Pi_n + \pi_{n+1} + \dots + \pi_m$$

the algorithm first finds the term k that appears in the most products, where k may itself be composed of several terms. In the above, terms $\pi_i, n + 1 \leq i \leq m$ do not contain k . Rewriting $\pi_i, n + 1 \leq i \leq m$ as Γ , the expression can be rewritten as

$$k * (\Pi_1 + \Pi_2 + \dots + \Pi_n) + \Gamma$$

which generates the intermediate result in Equation 2 above. The algorithm is then applied to k , $(\Pi_1 + \Pi_2 + \dots + \Pi_n)$ and Γ (which generates the result in Equation 3 above). The algorithm is applied to each created sub-term until no further optimization is possible.

3.4 Common Subexpression Elimination

We perform two specialized common subexpression elimination algorithms. 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 first CSE optimization changes the equations from:

$$\frac{dA}{dt} = \dots + (A + B + C + D) * k_1 * E + \dots$$

$$\frac{dB}{dt} = \dots + (A + B + C + D) * k_2 * F + \dots$$

into:

$$\begin{aligned} temp[0] &= A + B + C; \\ \frac{dA}{dt} &= \dots + temp[0] * k_1 * E + \dots \\ \frac{dB}{dt} &= \dots + temp[0] * k_2 * F + \dots \end{aligned}$$

After performing the first CSE optimization, there are still redundant expressions in the right hand sides of the temporary variable assignments. To eliminate these we perform another CSE optimization, e.g. the equations:

$$\begin{aligned} temp[4] &= A + B + C + D; \\ temp[6] &= A + B + C + D + F; \end{aligned}$$

are transformed into

$$\begin{aligned} temp[4] &= A + B + C + D; \\ temp[6] &= temp[4] + F; \end{aligned}$$

The optimization proceeds as follows. First, the sub-expressions that form equations are stored in a hash table keyed on the expression length. The terms of each temporary are stored in a canonical lexicographical order – this allows an easy matching of expressions. The algorithm considers in turn each e_l , the longest sub-expression not yet processed (or one of the longest, if there are several sub-expressions whose length is equal to l .) It then examines all shorter sub-expressions e_i of length $0 < i < l$ looking for a sub-expression that matches the leading i terms of e_l . The search is done from longest to shortest strings e_i and terminates when the first such string is found. The match is facilitated by the canonical order imposed on the terms making up each temporary. If such an expression is found, e_l is rewritten in terms of e_i and the remaining terms of e_l , as shown in the example above. Because a shared common sub-expression is always shorter than the (original length) of the expression that reuses its value, it is sufficient to emit the expressions in length order to ensure that all data dependences are satisfied.

Our CSE algorithm differs from the standard algorithms in the literature (e.g. [6]) 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 are less likely to have the same value than are pairs of integer variables. Therefore, we can use the *name* of a variable to approximate 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 “a+b+c” and “a+c+d” and “b+c+e”, the compiler must choose whether to make the CSEs “a+b”, “a+c”, “b+c”, or some combination. maintaining all combinations of these sub-expressions to determine the candidates for the CSE optimization is expensive, particularly if there are thousands of expressions, and tens or hundreds of thousands of sub-expressions, as in our system. By exploiting the structure of our

computation we can efficiently search the space of expressions that are likely to be common, and that have a high payoff when recognized as common.

4 Parallel Templates and Library Support

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. The non-linear solver manipulates the relative ratios of the kinetic parameters based on the model and experimental data. Using the model and experimental data the number of kinetic parameters can also be reduced, e.g. it is often the case that a kinetic parameter k_i is expressed as a ratio of other kinetic parameters, thus k_i can be written in terms of the other parameters. 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: `imsl_f_ode_runge_kutta()` and `imsl_f_ode_adams_gear()`. 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. 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 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 and upper bounds and initial values for the kinetic rates are set, and the optimization function is called to optimize the problem as we describe 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.

4.3 Objective Function

The objective function, shown in Figure 10, 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

```

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 = imslf_bounded_least_squares(void objective_fcn(),int m,
        int n, float xlb[], float xub[], float xguess[] ...);
    return 0;
}

```

Figure 8: the Non-linear Optimization Routine

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 of more than 3000 records of the form $\langle t_i, \text{property value} \rangle$, where t_i is a time step and **property value** is a measure of the property we desire to learn how to predict. 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 a 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 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 Figures 9 and 10. 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 “block distributed” across the processors (one or more whole files constituting a block) – 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.

5 Experimental Results

5.1 Rubber Chemistry

The system is being actively used within our group to model the vulcanization process of rubber. Therefore, the input for our experiments is the kinetic model for the vulcanization process implemented for the experiments models the vulcanization of natural rubber by the benzothiazolesulfenamide class of accelerators. In the final ODEs for this model, there are a total of 148

```

int main(int argc, char *argv[]) {
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);

    /* 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 9: the Parallel Non-linear Optimization Routine Using MPI

molecules, and 17 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 affecting 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.

A sample of the generated ODEs for this reaction can be found in Appendix A. By automatically generating these ODEs, our compiler now does in a few minutes what used to take graduate students days 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 -switch connect the nodes. For our experiments we used the proprietary IBM switch and thin nodes, and used one processor per node.

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 the different optimization combination. *Opt I* is the combination optimization, *Opt II* is the

Distributive optimization, *Opt III* is the first CSE optimization and *Opt IV* is the second CSE optimization. These optimizations reduce the single-node execution time by almost 19%. As can be seen, the total of the four optimizations reduces the number of multiplies to 23% of the original number, and reduces the number of add-ops to 43% of the original number.

optimization combination	Execution Time(seconds)	perf. gain (%)	total mulops (*)	total add-ops (+,-)
without optimization	1394.50804	1	2668	1769
Opt I	1241.938752	1.123	2289	1444
Opt I+II	1224.896532	1.138	629	1606
Opt I+II+III	1204.987973	1.157	629	1379
Opt I+II+III+IV	1174.855434	1.187	629	761

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

5.4 Results of Parallel Computation Using MPI

Table 2 demonstrates the performance using different number of nodes. Only one processor is used in each node. As the number of nodes increases, the execution time decreases, and the speedup increases. The speedup does not increase linearly, but slows down gradually, as more nodes are in use. The reason for this is that there is a load imbalance caused by different amounts of work involved in processing each file. We are currently investigating ways to distribute this work more evenly across the processors.

CPU Combination	Total Execution Time(seconds)	Speedup
1 node	1224.896532	1
2 nodes	613.341418	1.997
4 nodes	314.494057	3.895
8 nodes	172.893382	7.085
16 nodes	95.802051	12.786

Table 2: Results in IBM/SP Using MPI

6 Related Work

With improvements in computational power, interest in exploiting computers to design new materials increased dramatically. Venkat et al.[5] [4] 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.

Prichett et al. [7] [8] [9] designed a new 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.

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. Katareand, 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, Vol.76, No.3, pp.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*. Vol.43, No.14, pp.3484-3512, Jul 7, 2004.
- [5] S. Katareand, P. Patkar, P. Ghosh, J. M. Caruthers, W. N. Delgass, and V. Venkatasubramanian. *A systematic framework for rational formulation and design of materials*. Materials Design Approaches and Experience, pp.321-332, 2001.
- [6] 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.
- [7] S.E.Prickett and M.L.Mavrovouniotis. *Construction of complex reaction syatems-I. Reaction description language*. Computers chem Engng, Vol.21, No.11, pp.1219-1235, 1997.
- [8] S.E.Prickett and M.L.Mavrovouniotis. *Construction of complex reaction syatems-II. Molecule manipulation and reaction application algorithms*. Computers chem Engng, Vol.21, No.11, pp.1237-1254, 1997.
- [9] S.E.Prickett and M.L.Mavrovouniotis. *Construction of complex reaction syatems-II. An example: alkylation of olefins*. Computers chem Engng, Vol.21, No.11, pp.1325-1337, 1997.

```

#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,...);
    }
    MPI_AllReduce(error_vector, global_error_vector, max_steps,
        MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
}

```

Figure 10: the Parallel Objective Function Using MPI

Appendix: ODEs Generated for the Rubber Chemistry Problem

Figure 11 shows a list of the ODEs automatically generated by our compiler for the experimental results. The reaction being studied is part of a long-term study into understanding the vulcanization of rubber.

$dS_{ring}/dt = -k[0]*S_{ring} - k[1]*S_{ring}*S_{dirad}[8] - k[2]*S_{ring}*BtS_{xrad}[1] - k[2]*S_{ring}*BtS_{xrad}[2] - k[2]*S_{ring}*BtS_{xrad}[3] - k[2]*S_{ring}*BtS_{xrad}[4] - k[2]*S_{ring}*BtS_{xrad}[5] - k[2]*S_{ring}*BtS_{xrad}[6] - k[2]*S_{ring}*BtS_{xrad}[7] - k[2]*S_{ring}*BtS_{xrad}[8] - k[3]*S_{ring}*HS_{xrad}[1] - k[3]*S_{ring}*HS_{xrad}[2] - k[3]*S_{ring}*y[68] - k[3]*S_{ring}*HS_{xrad}[4] - k[3]*S_{ring}*y[70] - k[3]*S_{ring}*HS_{xrad}[6] - k[3]*S_{ring}*HS_{xrad}[7] - k[3]*S_{ring}*y[73];$
 $dS_{dirad}[1]/dt = 0; dS_{dirad}[2]/dt = 0; dS_{dirad}[3]/dt = 0; dS_{dirad}[4]/dt = 0; dS_{dirad}[5]/dt = 0; dS_{dirad}[6]/dt = 0; dS_{dirad}[7]/dt = 0;$
 $dS_{dirad}[8]/dt = +k[0]*S_{ring} - k[1]*S_{ring}*S_{dirad}[8] - k[4]*S_{dirad}[8] - k[5]*S_{dirad}[8]*MBT;$
 $dS_{dirad}[9]/dt = 0; dS_{dirad}[10]/dt = 0; dS_{dirad}[11]/dt = 0; dS_{dirad}[12]/dt = 0; dS_{dirad}[13]/dt = 0; dS_{dirad}[14]/dt = 0; dS_{dirad}[15]/dt = 0;$
 $dS_{dirad}[16]/dt = +k[1]*S_{ring}*S_{dirad}[8] - k[4]*S_{dirad}[16] - k[5]*S_{dirad}[16]*MBT; dMBT/dt = -k[5]*S_{dirad}[16]*MBT - k[5]*S_{dirad}[8]*MBT + k[6]*RSxBt[3] + k[6]*RSxBt[4] + k[6]*RSxBt[5] + k[6]*RSxBt[6] + k[6]*y[106] + k[6]*y[107] + k[6]*y[108] + k[6]*RSxBt[10] + k[6]*RSxBt[11] + k[6]*RSxBt[12] + k[6]*RSxBt[13] + k[6]*RSxBt[14] + k[6]*RSxBt[15] + k[6]*RSxBt[16];$
 $dA[1]/dt = 0;$
 $dA[2]/dt = -k[7]*A[2] - k[8]*BtS_{xrad}[2]*A[2] - k[8]*BtS_{xrad}[3]*A[2] - k[8]*BtS_{xrad}[4]*A[2] - k[8]*BtS_{xrad}[5]*A[2] - k[8]*BtS_{xrad}[6]*A[2] - k[8]*BtS_{xrad}[7]*A[2] - k[8]*BtS_{xrad}[8]*A[2] - k[8]*BtS_{xrad}[9]*A[2] - k[8]*BtS_{xrad}[10]*A[2] - k[8]*BtS_{xrad}[11]*A[2] - k[8]*BtS_{xrad}[12]*A[2] - k[8]*BtS_{xrad}[13]*A[2] - k[8]*BtS_{xrad}[14]*A[2] - k[8]*BtS_{xrad}[15]*A[2]; dA[3]/dt = -2.0*k[7]*A[3] + k[8]*BtS_{xrad}[2]*A[2] - k[8]*BtS_{xrad}[2]*A[3] - k[8]*BtS_{xrad}[3]*A[3] - k[8]*BtS_{xrad}[4]*A[3] - k[8]*BtS_{xrad}[5]*A[3] - k[8]*BtS_{xrad}[6]*A[3] - k[8]*BtS_{xrad}[7]*A[3] - k[8]*BtS_{xrad}[8]*A[3] - k[8]*BtS_{xrad}[9]*A[3] - k[8]*BtS_{xrad}[10]*A[3] - k[8]*BtS_{xrad}[11]*A[3] - k[8]*BtS_{xrad}[12]*A[3] - k[8]*BtS_{xrad}[13]*A[3] - k[8]*BtS_{xrad}[14]*A[3];$
 $dA[4]/dt = -2.0*k[7]*A[4] - k[7]*A[4] + k[8]*BtS_{xrad}[2]*A[3] - k[8]*BtS_{xrad}[2]*A[4] + k[8]*BtS_{xrad}[3]*A[2] - k[8]*BtS_{xrad}[3]*A[4] - k[8]*BtS_{xrad}[4]*A[4] - k[8]*BtS_{xrad}[5]*A[4] - k[8]*BtS_{xrad}[6]*A[4] - k[8]*BtS_{xrad}[7]*A[4] - k[8]*BtS_{xrad}[8]*A[4] - k[8]*BtS_{xrad}[9]*A[4] - k[8]*BtS_{xrad}[10]*A[4] - k[8]*BtS_{xrad}[11]*A[4] - k[8]*BtS_{xrad}[12]*A[4] - k[8]*BtS_{xrad}[13]*A[4];$
 $dA[5]/dt = -2.0*k[7]*A[5] - 2.0*k[7]*A[5] + k[8]*BtS_{xrad}[2]*A[4] - k[8]*BtS_{xrad}[2]*A[5] + k[8]*BtS_{xrad}[3]*A[3] - k[8]*BtS_{xrad}[3]*A[5] + k[8]*BtS_{xrad}[4]*A[2] - k[8]*BtS_{xrad}[4]*A[5] - k[8]*BtS_{xrad}[5]*A[5] - k[8]*BtS_{xrad}[6]*A[5] - k[8]*BtS_{xrad}[7]*A[5] - k[8]*BtS_{xrad}[8]*A[5] - k[8]*BtS_{xrad}[9]*A[5] - k[8]*BtS_{xrad}[10]*A[5] - k[8]*BtS_{xrad}[11]*A[5] - k[8]*BtS_{xrad}[12]*A[5];$
 $dA[6]/dt = -2.0*k[7]*A[6] - 2.0*k[7]*A[6] - k[7]*A[6] + k[8]*BtS_{xrad}[2]*A[5] - k[8]*BtS_{xrad}[2]*A[6] + k[8]*BtS_{xrad}[3]*A[4] - k[8]*BtS_{xrad}[3]*A[6] + k[8]*BtS_{xrad}[4]*A[3] - k[8]*BtS_{xrad}[4]*A[6] + k[8]*BtS_{xrad}[5]*A[2] - k[8]*BtS_{xrad}[5]*A[6] - k[8]*BtS_{xrad}[6]*A[6] - k[8]*BtS_{xrad}[7]*A[6] - k[8]*BtS_{xrad}[8]*A[6] - k[8]*BtS_{xrad}[9]*A[6] - k[8]*BtS_{xrad}[10]*A[6] - k[8]*BtS_{xrad}[11]*A[6];$
 $dA[7]/dt = -2.0*k[7]*A[7] - 2.0*k[7]*A[7] - 2.0*k[7]*A[7] + k[8]*BtS_{xrad}[2]*A[6] - k[8]*BtS_{xrad}[2]*A[7] + k[8]*BtS_{xrad}[3]*A[5] - k[8]*BtS_{xrad}[3]*A[7] + k[8]*BtS_{xrad}[4]*A[4] - k[8]*BtS_{xrad}[4]*A[7] + k[8]*BtS_{xrad}[5]*A[3] - k[8]*BtS_{xrad}[5]*A[7] + k[8]*BtS_{xrad}[6]*A[2] - k[8]*BtS_{xrad}[6]*A[7] - k[8]*BtS_{xrad}[7]*A[7] - k[8]*BtS_{xrad}[8]*A[7] - k[8]*BtS_{xrad}[9]*A[7] - k[8]*BtS_{xrad}[10]*A[7];$
 $dA[8]/dt = -2.0*k[7]*A[8] - 2.0*k[7]*A[8] - 2.0*k[7]*A[8] - k[7]*A[8] + k[8]*BtS_{xrad}[2]*A[7] - k[8]*BtS_{xrad}[2]*A[8] + k[8]*BtS_{xrad}[3]*A[6] - k[8]*BtS_{xrad}[3]*A[8] + k[8]*BtS_{xrad}[4]*A[5] - k[8]*BtS_{xrad}[4]*A[8] + k[8]*BtS_{xrad}[5]*A[4] - k[8]*BtS_{xrad}[5]*A[8] + k[8]*BtS_{xrad}[6]*A[3] - k[8]*BtS_{xrad}[6]*A[8] + k[8]*BtS_{xrad}[7]*A[2] - k[8]*BtS_{xrad}[7]*A[8] - k[8]*BtS_{xrad}[8]*A[8] - k[8]*BtS_{xrad}[9]*A[8];$
 $dA[9]/dt = -2.0*k[7]*A[9] - 2.0*k[7]*A[9] - 2.0*k[7]*A[9] - 2.0*k[7]*A[9] + k[8]*BtS_{xrad}[2]*A[8] - k[8]*BtS_{xrad}[2]*A[9] + k[8]*BtS_{xrad}[3]*A[7] - k[8]*BtS_{xrad}[3]*A[9] + k[8]*BtS_{xrad}[4]*A[6] - k[8]*BtS_{xrad}[4]*A[9] + k[8]*BtS_{xrad}[5]*A[5] - k[8]*BtS_{xrad}[5]*A[9] + k[8]*BtS_{xrad}[6]*A[4] - k[8]*BtS_{xrad}[6]*A[9] + k[8]*BtS_{xrad}[7]*A[3] - k[8]*BtS_{xrad}[7]*A[9] \dots$

Figure 11: Partial ODEs Generated By the Chemical Compiler