SANDIA REPORT
SAND2001-8075
Unlimited Release
Printed November 2000

MPP Direct Numerical Simulation of Diesel Autoignition

H. N. Najm, J. H. Chen, J. F. Grcar, R. C. Armstrong, C. A. Kennedy, J. Ray, W. S. Koegler, A. E. Lutz, M. D. Allendorf, D. Klinke, A. H. McDaniel, N. Nystrom, R. Subramanya, and R. Reddy

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/ordering.htm

Sandia Report SAND2001-8075
Unlimited Release
Printed November 2000

MPP Direct Numerical Simulation of Diesel Autoignition

H.N. Najm, J.H. Chen, J.F. Grcar, R.C. Armstrong, C.A. Kennedy, J. Ray, W.S. Koegler
A.E. Lutz, M.D. Allendorf, D. Klinke, and A.H. McDaniel

Sandia National Laboratories
Livermore, CA 94550

and

N. Nystrom, R. Subramanya, and R. Reddy

Pittsburgh Supercomputing Center
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We discuss developments of massively parallel simulation and post-processing tools for direct numerical simulations of reacting flow, as well as results of flame studies pertaining to diesel conditions. A 3D massively parallel (MP) compressible fixed-mesh reacting flow direct numerical simulation (DNS) code was developed. Machine-generated code for efficient reaction rate evalutions was implemented and benchmarked. Additive implicit-explicit Runge-Kutta time integration methods, targeted at stiff reacting flow problems, were developed and evaluated. Object-oriented code frameworks were utilized and evaluated with regard to their use for adaptive mesh refinement (AMR) computations of reacting flow with detailed kinetics. Common-Component-Architecture (CCA) based data mining tools were developed and used to identify, track, and analyze arbitrary features, in computed AMR results of ignition kernels in 2D reacting flow. The burnout of fuel-rich diffusion flame pockets, expected features of autoignition in diesel engines, was studied using a 1D transient flame code using detailed kinetics. We also modeled the catalytic oxidation of hydrogen over palladium as part of continuing efforts focused on processes relevant in catalytic converters. Overall, this work fostered and advanced collaborations with the Pittsburgh Supercomputer Center (PSC) on massively parallel reacting flow code development and optimization, and with other groups in Sandia-CA on the development and utilization of Common Component Architecture (CCA) object-oriented frameworks for maximizing code reuse.

Acknowledgments

This work was supported by the Laboratory Directed Research and Development (LDRD) program at Sandia National Laboratories (SNL).

Contents

1. Introduction

This project uses collaborations with academic partners and participating Sandia organizations to move the state of the art of large scale Massively Parallel Programming (MPP) reacting flow simulations forward. The effort involves various facets including efficient coding and optimization of existing algorithms for massively parallel (MP) hardware, enhancements to code performance by using machine-coding of cpu-intensive routines, development of efficient implicit-explicit high-order schemes for stiff reacting flow systems, testing and evaluation of object-oriented frameworks for parallel Adaptive Mesh Refinement (AMR) computations of reacting flow, development of Common Component Architecture (CCA) based object-oriented postprocessing tools for AMR data sets, studies of post-ignition flame burnout at diesel engine conditions, and catalytic studies relevant to processes in catalytic converters.

During the course of the project the following was accomplished:

1. A 3D massively parallel (MP) compressible fixed-mesh reacting flow direct numerical simulation (DNS) code was constructed, demonstrated, and profiled on various computational platforms. This code is now being used for large-scale 3D DNS of turbulent flow with detailed kinetics, greatly enhancing prior capabilities.

2. A detailed study was conducted in collaboration with Pittsburgh Supercomputing Center (PSC) to implement and evaluate/benchmark machine-coded (or auto-coded) reaction-rate evaluation routines in *CHEMKIN-III*. The use of auto-coded rates routines was found to offer substantial savings in cpu-time, leading to more efficient code.

3. A detailed study of N-Additive Runge-Kutta (ARK_N) time integration methods was conducted. The construction of these schemes was examined, along with their error and stability analyses. ARK_N schemes allow for operator-selective integration of governing equations, where an optimal RK construction is used for each operator. This is a necessity for the integration of stiff reacting flow equations where implicit time integration is utilized for stiff terms and explicit integration is used for the non-stiff terms.

4. Two potential object oriented frameworks for adaptive mesh refinement (AMR) DNS of reacting flow (OVERTURE[24] and GRACE[7]) were evaluated and tested as regards overall efficiency, overhead costs, and scalability. We have concluded that the GRACE framework is more efficient, lightweight, scalable, and more readily available for distributed parallel computing. As a

result of this investigation, GRACE is now being used as the basis of new Common Component Architecture (CCA) code development in Sandia.

5. A CCA-based set of tools for data mining/feature extraction targeted at general unstructured/structured meshes have been developed. These tools have been demonstrated on specific test problems, and are being targeted at large-scale AMR or fixed-mesh reacting flow data sets.

6. The burnout of fuel-rich diffusion flame pockets, expected features of autoignition in diesel engines, was studied using a 1D transient flame code using detailed kinetics. Results are used to study the transient evolution of flame radicals during such processes.

7. Given the relevance of post-combustion catalytic conversion of pollutant species to automotive applications, we also targeted modeling efforts at catalysis. We present a study of the catalytic oxidation of hydrogen over palladium.

The execution of this project has brought us significantly further along in the formulation and development of advanced efficient and scalable next-generation MP codes for resolved reacting flow computations with detailed kinetics and transport in large-scale flow. Such capabilities are crucial for accurate computations of reacting flow at diesel engine conditions, or in geometries approaching diesel engine scales.

2. DNS Code Development

We have collaborated with Ravi Subramanya and Reddy Raghurama at the Pittsburgh Supercomputing Center at Carnegie Mellon University on developing a massively parallel 2D/3D DNS code based on the compressible formulation of the reacting flow equations using MPI message passing protocols. This collaboration has led to an efficient, scalable code that is portable across multi-processor platforms that support MPI, including Cray T3E, SGI Origin, and IBM SP3 machines. The code has been validated against serial flame-vortex, and turbulent flame DNS runs. See Figure 2.1 for scalability data. Profiling of the code indicted that a significant fraction of the CPU time in performing DNS is expended on computing the reaction rate. Computation of the reaction rate was sped up by 60% for the test case, and overall the code speedup was 13% by generating in-line code for the reaction rate subroutines. A Chemkin API in the DNS code [34, 35] was also developed to facilitate inclusion of different chemical mechanisms. Utilities to morph restart files to different partition sizes were developed and parallel turbulence initialization routines were written.

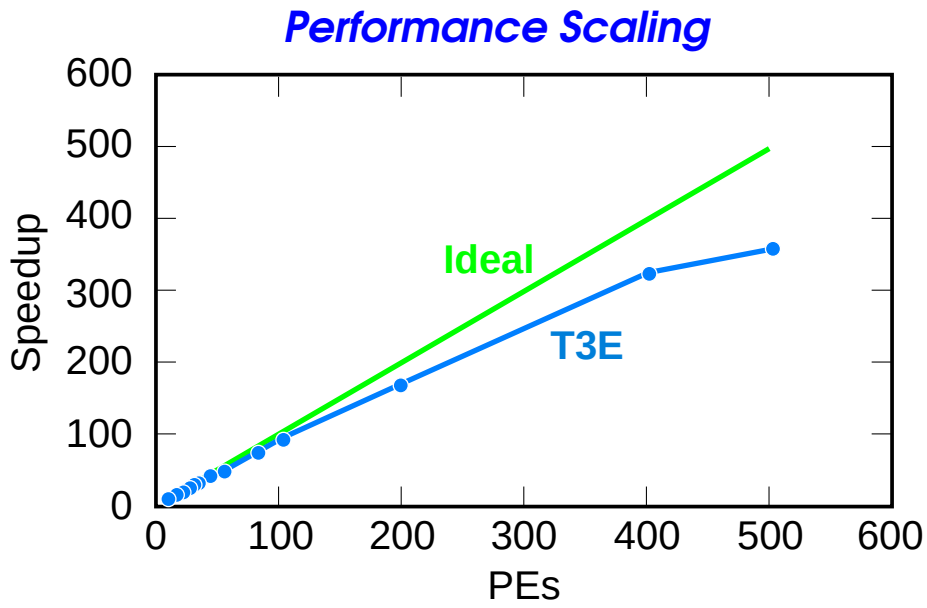

Figure 2.1 3D DNS code scaling curves for the T3E performed with a flame-vortex problem. The straight line corresponds to linear speedup.

3. Auto-CHEMKIN Software Development

3.1 General Description

This project focuses on comparing the performance and design characteristics of the *CHEMKIN-III* subroutine `CKWYP` and its descendents, responsible for returning molar production rates for the chemical species given pressure, temperature(s), and mass fractions, versus that of automatically generated ("autocode") replacement routines [38, 21]. The latter subroutines, generated by *auto-CHEMKIN* for each chemical system and collectively denoted herein as `GETRATES`, have different performance characteristics from `CKWYP` and offer advantages for building a chemical kinetics/combustion problem solving environment (PSE). For a complete description of the project and its findings, see [23].

3.2 Background on CHEMKIN and Auto-CHEMKIN

CHEMKIN-III is a suite of application codes, libraries, and databases. Robert J. Kee and others at Sandia wrote the original software. The libraries are used in fluid dynamics codes to supply values such as chemical source terms and transport properties. The fluid dynamics codes may either use the *CHEMKIN-III* libraries directly, or may use them within ODE modules that propagate chemical composition in time. The libraries compute reaction rates from "first principles" Arrhenius expressions based on Arrhenius and other parameters that are devised in part using the application codes.

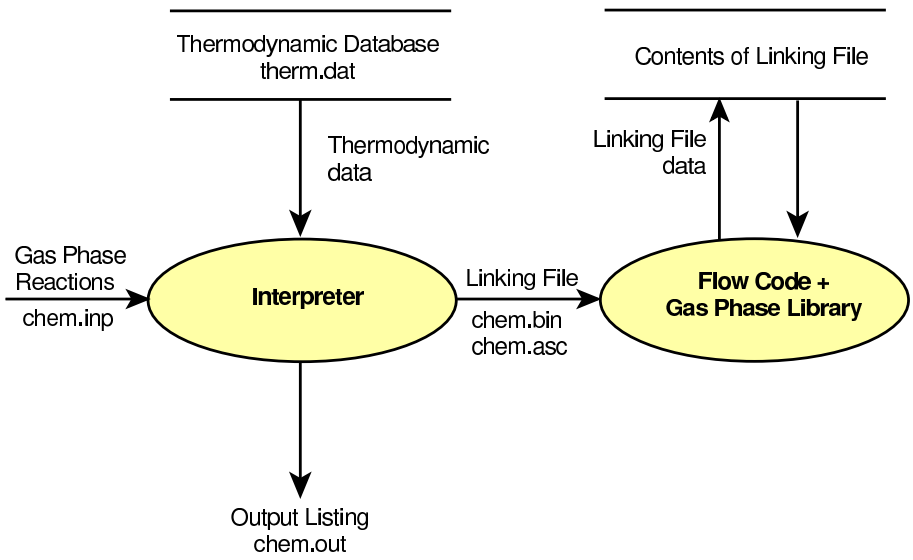

Figure 3.1 High Level Data Flow Diagram for *CHEMKIN-III*. Ovals represent data transformations, arrows represent data flows, and twin horizontal lines represent data stores.

The High Level Data Flow Diagram shown in Figure 3.1 illustrates the general usage of *CHEMKIN-*

III. The interpreter is a program that reads a symbolic description of a reaction and then extracts the needed thermodynamic data for each species involved from the thermodynamic database. The output from the interpreter is a binary/ASCII file called the linking file. This file contains all the required information about the elements, species, and reactions in the user's mechanism. Inside the flow code, the user dimensions three arrays based on the output of the interpreter and then calls `CKINIT` to read the linking file. The Gas Phase Library subroutines can then be called at any time to compute thermodynamic properties, chemical production rates, etc.

Auto-CHEMKIN is an automatic code generator for a subset of the *CHEMKIN-III* functionality. Peter Wyckoff and Habib Najm of Sandia developed and implemented this software. It consists of `getrates`, an automatic code generator written using *lex* and *yacc* to parse a chemical reaction "grammar". That grammar describes the contents of *chem.inp* files, which contain problem-specific reaction and rate data. The result is source code specialized for a particular reaction mechanism.

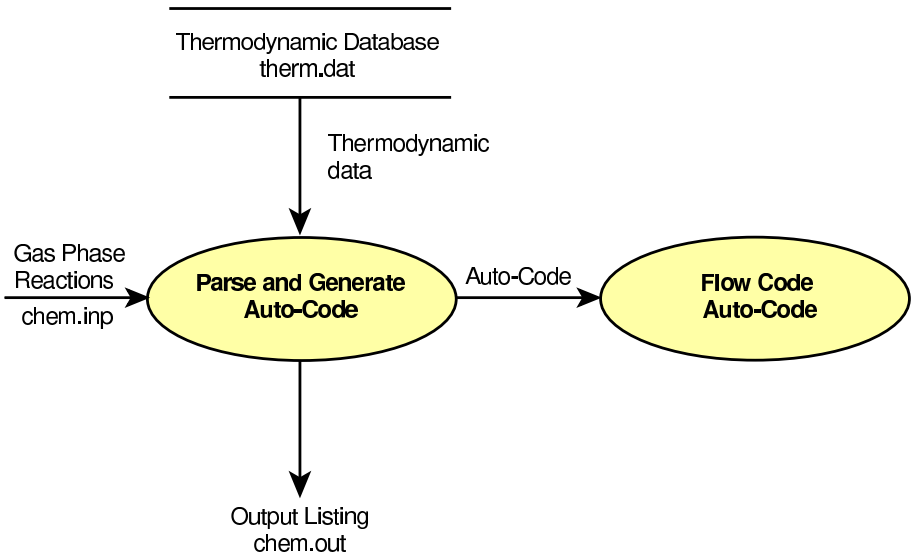

Figure 3.2 High Level Data Flow Diagram for auto-CHEMKIN. Ovals represent data transformations, arrows represent data flows, and twin horizontal lines represent data stores.

The High Level Data Flow Diagram shown in Figure 3.2 illustrates the general usage of *auto-CHEMKIN*. The user provides the same input as when using *CHEMKIN-III*, namely *chem.inp* and *therm.inp* files. In the case of *auto-CHEMKIN*, however, the `getrates` code generator parses the *chem.inp* file to produce Fortran-77 source. The main entry point in the source so generated is subroutine `GETRATES`, which can be called by applications to obtain the rates of generation of all species. The source is available for compiling and linking against applications, thereby offering an automated

process for generating efficient, problem-dependent code.

3.3 Modifications to Auto-CHEMKIN

A modified version of the autocode generator, named `cgetrates` to denote the distinction, was created to produce implementations of the `GETRATES` subroutine that conform to `CKWYP` calling conventions. The Fortran source code produced by `cgetrates` can be compiled and linked into a program that would otherwise use `CKWYP`, with the only change required in the application being to change calls from `CKWYP` to `GETRATES`. The actual calling sequence is identical.

In addition to the changes necessary for generating `CKWYP`-compatible code, the `cgetrates` autocode generator contains several extensions to parse more complex mechanisms:

- hyphens are accepted in species names, e.g. C_3H_4-a
- an optional "(+M)" is accepted in species names, e.g. CH_3O(+M)
- commas are accepted in species names, e.g. $C_2H_4O_{1,2}$
- long species names supported (up to 20 characters)
- Fortran output improved for readability and consistency

Validation of the modified autocode generator `cgetrates` was performed by automated comparison of large numbers of rate results against those produced by `CKWYP`. Results were within acceptable tolerances based on what one would expect from applying different sequences of large numbers of floating-point operations to arrive at a given result. In addition, for the smaller two benchmarks, autocode from `getrates` was also found to produce identical results to those from `cgetrates`.

3.4 Performance of Auto-CHEMKIN

The application used to obtain performance data is the *wmain* driver written by Joseph Grcar at Sandia. During its execution, *wmain* calls `CKWYP` and `GETRATES` in sequence, `LOOP`×`VALUES` each. For the purpose of this testing, `LOOP`=100 and `VALUES`=100, and measurements are collected over 11 cycles. The *wmain* driver includes additional validation of results against those generated using `CKWYP`, providing a reasonable level of assurance that the `GETRATES` implementation is generating correct results.

Performance analysis of *CHEMKIN-III* and *auto-CHEMKIN* was carried out on `skipper.psc.edu`, a 4-processor Silicon Graphics Power Challenge. Each 194-MHz processor contains a MIPS R10000

CPU and a MIPS R10010 FPU. `Skipper` has 1 GB of RAM, with 32-kB instruction and primary data caches and a 2-MB secondary data cache. However, every effort has been made to ensure code portability and to consider extensibility of the analyses to other systems.

The performance analysis presented here reflects one particular hardware/software environment. While details will naturally differ between systems in terms of processor architecture, caches, compilers, etc., the conclusions will focus on aspects applicable to a broad range of platforms representing the portion of the market that is dominant today as well as for the foreseeable future. In particular, that includes major offerings based on MIPS, Digital, SPARC, and Intel chipsets. Exceptions include certain proprietary architectures that differ substantially from the superscalar RISC model, including for example, the Cray T90, the Fujitsu VPP5000, the NEC SX-5, and the Tera MTA. Each of those systems is sufficiently specialized to merit individual consideration; however, the represent a relatively small number of installations and have less bearing on anticipated CHEMKIN usage.

For purposes of code development and subsequent performance analysis, four distinct systems of chemical reactions of increasing complexity were considered. They are described in Table 3.1.

Numerous profiling runs were executed using each set of benchmarking data to understand the performance of the *auto-CHEMKIN* approach relative to that of the *CHEMKIN-III* approach, to identify execution bottlenecks, and to reveal opportunities for optimization. Measurements were made on a nearly dedicated system (i.e. other processes were not disallowed, but no other users were active, and each benchmarking job was the only significant active process during the course of its run), leading to good reproducibility.

Calls to *etime* provide a best estimate of execution time in the absence of any overhead introduced by performance tuning tools. These times are divided into user time, the CPU time used while executing instructions in the user space of the calling process, and system time, the CPU time used by the system on behalf of the calling process. Results on skipper are precise to 0.01s. Results for the four benchmarks under consideration are shown in Table 3.2.

Execution times for `getrates` are lower than for `CKWYP` by significant margins. Based on execution times measured using *etime* (i.e. with essentially no overhead), `getrates` proved to be 2.48, 2.84, 2.95, and 4.19 times faster than `CKWYP` for benchmarks A, B, C, and D, respectively. Performance of `getrates` relative to `CKWYP` increases with problem size as illustrated in Figure 3. This recommends `GETRATES` generally and especially for applications where large systems of reactions are important.

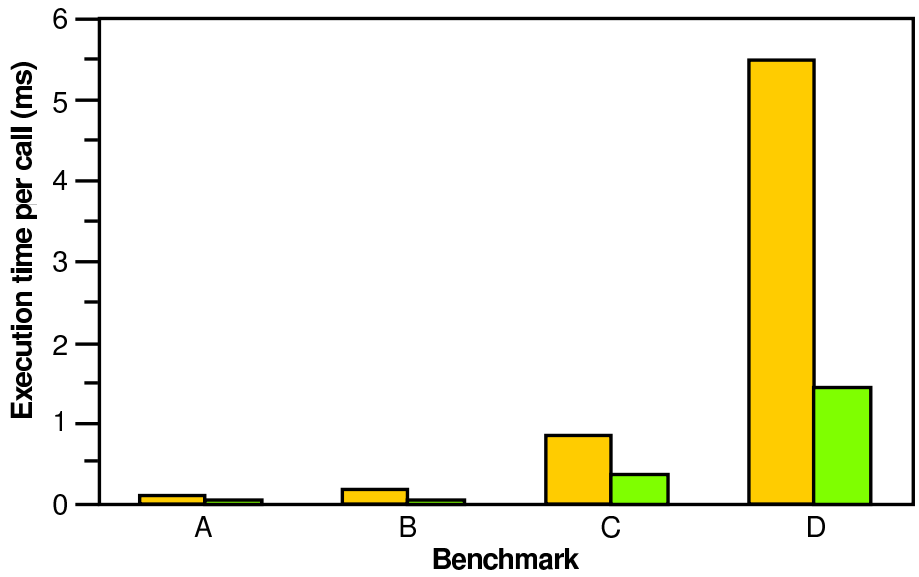

Figure 3.3 Improved execution of `GETRATES` relative to `CKWYP` for four benchmark problems of increasing complexity.

3.5 Analysis of Performance Differences

The factors underlying `GETRATES`' superior performance were analyzed. The *SpeedShop tool* was used to obtain detailed profiling information through access to the 32 hardware counters available on the R10000 processor. A wide range of experiments was run, targeting performance-related quantities including execution time, cache behavior, and instruction counts of individual subroutines. Experiments other than those presented here were also performed. For example, floating-point exceptions were instrumented and found to be absent. Repeat runs confirmed the high reproducibility of the profiles. The data so gathered reveal much about the performance characteristics of *CHEMKIN-III*'s `CKWYP` and *auto-CHEMKIN*'s `GETRATES`, the relation between them, and opportunities for optimization.

Figures 3.5 and 3.6 dissect the total execution time into that consumed by each subroutine and intrinsic function, as measured by sampling the program counter once per millisecond. In both cases, computing exponentials (`exp`) is the dominant operation, consuming up to 30% of the CPU time in the context of `CKWYP` and up to 40% of the CPU time in the context of `GETRATES`. For `GETRATES` the only subroutine with demands close to those of exp is `GETRATES` itself, simply because that is where the body of the rate calculations now reside.

The profiles described by Figures 3.5 and 3.6 imply that realizing additional performance gains through performance tuning would require significant improvements to many subroutines in the case of `CKWYP`, but only to the top-level routine in the case of `GETRATES`. In addition, `CKWYP` spends up

to 3.5 times as long in `exp` as does `GETRATES`, reducing the potential gains realizable through tuning *CHEMKIN-III* for `CKWYP` relative to `GETRATES`.

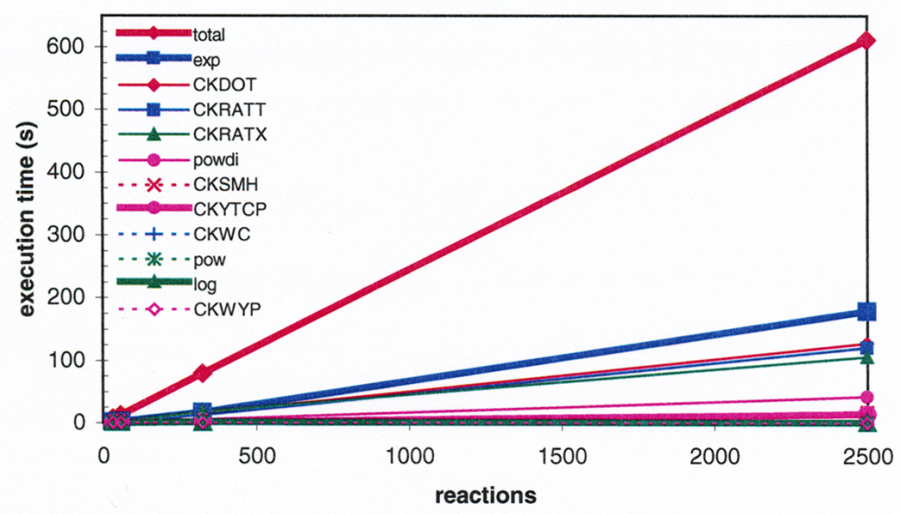

Figure 3.4 Execution times of `CKWYP`, related subroutines, and intrinsic functions as determined by program counter sampling once per millisecond.

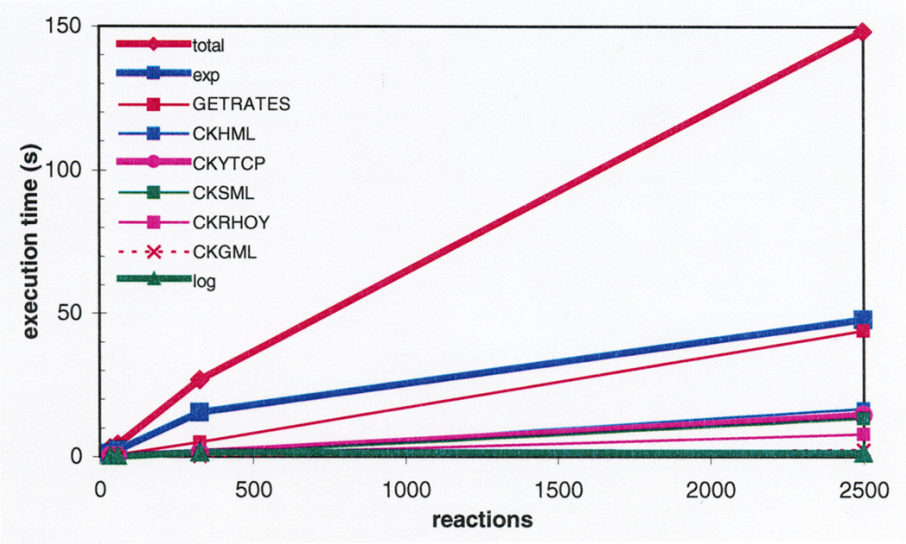

Figure 3.5 Execution times of `GETRATES`, related subroutines, and intrinsic functions as determined by program counter sampling once per millisecond.

Table 3.3 shows the total numbers of instructions required for `GETRATES` and `CKWYP` ($i_{\texttt{CKWYP}}$, $i_{\texttt{getrates}}$). This provides a good first-order predictor for the observed execution rates ($t_{\texttt{CKWYP}}$, $t_{\texttt{getrates}}$). Such a first-order approximation is always high, however, indicating that other factors such as instruction type

and efficiency of instruction and data cache use act to narrow the performance margin.

The execution rate of floating-point instructions, obtained from accurate CPU times (etime) and graduated floating point operation counts, is observed to be invariably higher for `GETRATES` than for `CKWYP` (Table 3.4). In addition to the difference in floating point execution rates, `GETRATES` also performs far fewer floating-point operations, amplifying the disparity in total execution times. For example, considering the large n-heptane benchmark, `CKWYP` consumes 537 seconds performing floating-point operations, whereas `GETRATES` consumes only 128 seconds.

The single operation, which consumes the most time for both *CHEMKIN-III* and *auto-CHEMKIN* and for all four benchmarks, is computing exponentials, i.e. `exp()`. Heavy dependence on computing exponentials is to be expected when calculating reaction rates, and profiling indicates that up to 57% of the execution time goes to this task (Table 3.5). This strong dependence on `exp()` suggests possible approaches to further performance improvement.

3.6 Software-Engineering Considerations

When faced with the choice of optimizing a legacy library versus committing to a new approach, software-engineering considerations are at least as important as performance issues. Sound software engineering facilitates rapid development, reliable code, ease of maintenance, and software reusability and extensibility. Performance will improve with each generation of hardware, but inadequate attention to software design results in ongoing costs in maintenance, bug identification and removal, and enhancement.

The static library approach embodied in *CHEMKIN-III* pervaded several decades of scientific and engineering computing. That approach, however, has various drawbacks, especially when considered in the context of building modern problem solving environments:

1. Flow codes depend on the availability of *CHEMKIN-III* libraries on each target platform rather than being self-contained and portable.

2. Flow codes running on distributed systems, e.g. using MPI for interprocess communications, would have to provide a separate mechanism for communicating common blocks initialized in `CKINIT`.

3. In the current *CHEMKIN-III* structure, library routines add several layers to the calling tree, resulting in a performance penalty. In addition, Fortran compilers on today's computing systems cannot inline (i.e. further optimize) library calls.

4. The Fortran-77 library code should be replaced by modern, extensible, Fortran-95 (the current ANSI/ISO Fortran standard) to improve compatibility with current compilers, to ease maintenance, and to facilitate integration and enhancement.

The autocode generator approach used for *auto-CHEMKIN* provides an elegant solution to the problem of generating problem-specific code. In addition, it offers numerous benefits that would be comparatively expensive to realize using the former approach:

1. Flow code portability would be enhanced because only the autocode generator (itself portable by virtue of its design) need be running on each target platform.

2. Flow code performance would benefit from improved cache utilization, decreased communications requirements, and compiler inlining of code that currently resides in library routines.

3. Integrability, reliability, maintainability, and extensibility would be significantly improved.

3.7 Recommedations for Future Work

Based on the performance and software design of the two approaches, the top-level recommendation is to pursue development of *auto-CHEMKIN*. A prerequisite for the development effort would be item 1. Items 2 and 3 outline desirable features of autocode generators. Item 4 is more platform specific.

1. The customer will compile a prioritized list of *CHEMKIN-III* features. Based on the prioritization and available time, autocode generators for those features will be implemented and tested.

2. Automatically generated code, as well as the autocode generators themselves, should exhibit modern software engineering practices.

   (a) The `cgetrates` code generator should be rewritten to generate ANSI/ISO compliant C and/or C++ source code with appropriate interfaces for Fortran applications.

   (b) Alternatively, and at a minimum, Fortran-95 compliance is required due to the obsolescence of Fortran-77. References to deleted and obsolescent features should be replaced, and pending specification of user requirements, interfaces between applications and automatically generated code should be thoroughly defined, preferably as Fortran MODULEs. The revised `cgetrates` should be used as a model for other autocode generators.

3. Investigation into optimized ("vectorized") implementations of `exp()` could further decrease execution times.

3.8 Conclusion

The autocode approach of *auto-CHEMKIN* is found to be more efficient by a factor of 2.5 to 4.2 for a range of benchmark problems containing up to 2498 reversible reactions. The increase in efficiency is attributed primarily to a lower instruction count for the `GETRATES` subroutine and its dependents, and to a lesser extent on better data cache reuse. Three main recommendations are advanced: to modernize the autocode generators, to proceed with authoring autocode generators for other aspects of *CHEMKIN-III*, and to provide "vectorized" exponentials for applicable computer platforms.

Table 3.1 Benchmark problems used for performance analysis of CKWYP and getrates.

Benchmark	System	Elements	Species	Reactions	Source
A	SiF_4/NH_3	4	17	$33+0$	`chemkin/test/aurora/test001/chem.inp`
B	CH_4 Combustion	4	17	$38+0$	`chemkin/test/premix/test002/chem.inp`
C	GRI-Mech 3.0	5	53	$309+16$	`http://euler.me.berkeley.edu/gri_mech/`
D	n-heptane	5	570	$2498+0$	`Hong Im (hgim@heptane.ca.sandia.gov)`

Table 3.2 CPU time for `CKWYP` and `getrates` as dtermined by `etime`.

	Benchmark A	Benchmark B	Benchmark C	Benchmark D
`CKWYP` - t_{user}(s)	6.92×10^{-5}	1.12×10^{-4}	6.95×10^{-4}	5.36×10^{-3}
`CKWYP` - t_{system}(s)	2.04×10^{-8}	4.55×10^{-8}	6.25×10^{-8}	1.05×10^{-6}
`getrates` - t_{user}(s)	2.78×10^{-5}	3.94×10^{-5}	2.36×10^{-4}	1.28×10^{-3}
`getrates` - t_{system}(s)	2.09×10^{-8}	1.10×10^{-8}	2.81×10^{-8}	3.73×10^{-7}
speedup (user time)	2.48	2.84	2.95	4.19

Table 3.3 Inclusive instruction count ratios as an initial approximation to rate routine.

	Benchmark A	Benchmark B	Benchmark C	Benchmark D
$t_{\texttt{CKWYP}}/t_{\texttt{getrates}}$	2.48	2.90	2.81	4.19
$i_{\texttt{CKWYP}}/i_{\texttt{getrates}}$	2.67	3.08	3.18	5.34
relative error	7.6%	6.2%	13%	27%

Table 3.4 Floating-point performance of rate subroutines (MFLOPS).

	Benchmark A	Benchmark B	Benchmark C	Benchmark D
`CKWYP`	41.1	41.1	47.7	48.6
`GETRATES`	71.1	82.6	85.2	63.0

Table 3.5 Percentage of execution time devoted to computing exponentials.

	Benchmark A	Benchmark B	Benchmark C	Benchmark D
`CKWYP`	21%	24%	22%	29%
`GETRATES`	38%	50%	57%	32%

4. The Theory of N-Additive Runge-Kutta Methods

4.1 Introduction

It is oftentimes useful to consider the compressible Navier-Stokes equations (NSE) as evolution equations with several driving forces, each having somewhat different characteristics. Typically, one distinguishes between terms such as convection, diffusion, reaction. As such, one often considers the more tractable convection-diffusion-reaction (CDR) equations as a prologue to the full compressible NSE. In the search for ever more efficient integrators, it is intuitively appealing to seek individual integration methods which are ideally suited for specific parts of the governing equations. The individual methods are then rolled into a single composite method which, ideally, would be more efficient than any individual method applied to the full computation. To accomplish this, one may consider partitioned methods. Schemes constructed to take advantage of termwise partitioning of the CDR for integration purposes may be called additive methods. Runge-Kutta methods, with their extensive theoretical foundation, allow for the straightforward design and construction of stable, high-order, partitioned methods composed of arbitrary numbers of elemental Runge-Kutta schemes. In addition, they also allow for the direct control of partitioning (splitting or coupling) errors. Direct numerical simulation (DNS) and large-eddy simulation (LES) of fluid phenomena, with their relatively strict error tolerances, are prime candidates for such methods. The need for these strategies is by no means limited to the compressible NSE and CDR equations.

The goal of this section is to give a general overview of the coupling of N different Runge-Kutta methods for first-order ODEs whose right hand side is the summation of N terms; N-additive Runge-Kutta methods (ARK_N). We do not consider the topics of storage reduction, contractivity, dispersion and dissipation, regularity, or boundary error.

4.2 N-Additive Runge-Kutta Methods

4.2.1 General

Following Araújo, Murua, and Sanz-Serna[1], ARK_N methods are used to solve equations of the form

$$\frac{dU}{dt} \;=\; F(U) \;=\; \sum_{\nu=1}^{N} F^{[\nu]}(U), \tag{1}$$

where $F(U)$ has been additively composed of N terms. They are applied as

$$U^{(i)} \;=\; U^{(n)} + (\Delta t) \sum_{\nu=1}^{N} \sum_{j=1}^{s} a_{ij}^{[\nu]} F^{[\nu]}(U^{(j)}), \tag{2}$$

$$U^{(n+1)} = U^{(n)} + (\Delta t)\sum_{\nu=1}^{N}\sum_{i=1}^{s} b_i^{[\nu]} F^{[\nu]}(U^{(i)}), \tag{3}$$

$$\hat{U}^{(n+1)} = U^{(n)} + (\Delta t)\sum_{\nu=1}^{N}\sum_{i=1}^{s} \hat{b}_i^{[\nu]} F^{[\nu]}(U^{(i)}), \tag{4}$$

where each of the N terms are integrated by their own s-stage Runge-Kutta method. Also, $U^{(n)} = U(t^{(n)})$, $U^{(i)} = U\left(t^{(n)} + c_i\Delta t\right)$ is the value of the $U-$ vector on the i^{th}-stage, and $U^{(n+1)} = U(t^{(n)}+\Delta t)$. Both $U^{(n)}$ and $U^{(n+1)}$ are of order q. The $U-$vector associated with the embedded scheme, $\hat{U}^{(n+1)}$, is of order p. Each of the respective Butcher coefficients $a_{ij}^{[\nu]}$, $b_i^{[\nu]}$, $\hat{b}_i^{[\nu]}$, and $c_i^{[\nu]}$, $\nu = 1, 2, \cdots, N$ are constrained, at a minimum, by certain order of accuracy and stability considerations.

4.2.2 Order Conditions

Order-of-accuracy conditions for ARK_N methods may be derived via N-trees.[1] These N-trees resemble the traditional Butcher 1-trees[6, 14, 15] but each node may be any one of N varieties or colors. Expressions for the equations of condition associated with the q^{th}-order N-trees are of the form

$$\tau_{k[n]}^{(q)} = \frac{1}{\sigma}\Phi_{k[n]}^{(q)} - \frac{\alpha}{q!} = \frac{1}{\sigma}\left(\Phi_{k[n]}^{(q)} - \frac{1}{\gamma}\right), \qquad \text{where} \qquad \Phi_{k[n]}^{(q)} = \sum_{i}^{s} b_i \Phi_{i,k[n]}^{(q)}, \tag{5}$$

where $\Phi_{k[n]}^{(q)}$ and $\Phi_{i,k[n]}^{(q)}$ are scalar sums of Butcher coefficient products and $1 \le n \le N^q$ is used to distinguish between the many possible color variations of the k^{th} 1-tree of order q. Both α and σ are color dependent i.e., for a given 1-tree, the many corresponding N-trees may have different values of α and σ depending on the details of the node colorings. Tree density, γ, is color independent and consequently so is the product of α and σ; $\sigma\alpha = q!/\gamma$. Order conditions, $\tau_2^{(3)}$, $\tau_{2,4}^{(4)}$, $\tau_{6,7,9}^{(5)}$, given below never exhibit color dependence.

$$\begin{array}{llll}
\tau_1^{(1)} = \sum_{i=1}^{s} b_i - \frac{1}{1!} & & \tau_1^{(2)} = \sum_{i=1}^{s} b_i c_i - \frac{1}{2!} & \\
\tau_1^{(3)} = \frac{1}{2}\sum_{i=1}^{s} b_i c_i^2 - \frac{1}{3!} & & \tau_2^{(3)} = \sum_{i,j=1}^{s} b_i a_{ij} c_j - \frac{1}{3!} & \\
\tau_1^{(4)} = \frac{1}{6}\sum_{i=1}^{s} b_i c_i^3 - \frac{1}{4!} & & \tau_2^{(4)} = \sum_{i,j=1}^{s} b_i c_i a_{ij} c_j - \frac{3}{4!} & \\
\tau_3^{(4)} = \frac{1}{2}\sum_{i,j=1}^{s} b_i a_{ij} c_j^2 - \frac{1}{4!} & & \tau_4^{(4)} = \sum_{i,j,k=1}^{s} b_i a_{ij} a_{jk} c_k - \frac{1}{4!} & \\
\tau_1^{(5)} = \frac{1}{24}\sum_{i=1}^{s} b_i c_i^4 - \frac{1}{5!} & & \tau_2^{(5)} = \frac{1}{2}\sum_{i,j=1}^{s} b_i c_i^2 a_{ij} c_j - \frac{6}{5!} & \\
\tau_3^{(5)} = \frac{1}{2}\sum_{i,j=1}^{s} b_i a_{ij} c_j a_{ik} c_k - \frac{3}{5!} & & \tau_4^{(5)} = \frac{1}{2}\sum_{i,j=1}^{s} b_i c_i a_{ij} c_j^2 - \frac{4}{5!} & \\
\tau_5^{(5)} = \frac{1}{6}\sum_{i,j=1}^{s} b_i a_{ij} c_j^3 - \frac{1}{5!} & & \tau_6^{(5)} = \sum_{i,j,k=1}^{s} b_i c_i a_{ij} a_{jk} c_k - \frac{4}{5!} & \\
\tau_7^{(5)} = \sum_{i,j,k=1}^{s} b_i a_{ij} c_j a_{jk} c_k - \frac{3}{5!} & & \tau_8^{(5)} = \frac{1}{2}\sum_{i,j,k=1}^{s} b_i a_{ij} a_{jk} c_k^2 - \frac{1}{5!} & \\
\tau_9^{(5)} = \sum_{i,j,k,l=1}^{s} b_i a_{ij} a_{jk} a_{kl} c_l - \frac{1}{5!} & & &
\end{array}$$

When an equation of condition, $\tau_{k[n]}^{(q)}$, is made to vanish, color dependence is immaterial because $\Phi_{k[n]}^{(q)} = 1/\gamma$. For equations of condition that are not made to vanish, color dependence must be taken

into consideration to accurately assess the leading order error terms. Order conditions for partitioned Runge-Kutta methods have also been derived by Jackiewicz et al.[17] following an approach of Albrecht.

4.2.3 Coupling Conditions

Aside from satisfying the order conditions specific to each of the elemental methods of the $\mathrm{ARK_N}$, one must also satisfy various coupling conditions. One may write the total number of order conditions for a general $\mathrm{ARK_N}$ associated with each particular root node coloring, $\alpha_i^{[N]}$, using the expression[13, 14]

$$\sum_{i=1}^{\infty} \alpha_i^{[N]} x^{(i-1)} = \prod_{i=1}^{\infty} \left(1 - x^i\right)^{-N\alpha_i^{[N]}}. \tag{6}$$

At order i, a general $\mathrm{ARK_N}$ method has $N\alpha_i^{[N]}$ order conditions, where

$$\alpha_i^{[1]} = \{1, 1, 2, 4, 9, 20, \cdots\}, \tag{7}$$

$$\alpha_i^{[2]} = \{1, 2, 7, 26, 107, 458, \cdots\}, \tag{8}$$

$$\alpha_i^{[3]} = \{1, 3, 15, 82, 495, 3144, \cdots\}, \tag{9}$$

$$\alpha_i^{[4]} = \{1, 4, 26, 188, 1499, 12628, \cdots\}, \tag{10}$$

$$\alpha_i^{[5]} = \{1, 5, 40, 360, 3570, 37476, \cdots\}. \tag{11}$$

Some of these, $N\alpha_i^{[1]}$, are order conditions of the elemental methods which compose the $\mathrm{ARK_N}$. This implies that $N(\alpha_i^{[N]} - \alpha_i^{[1]})$ of the order conditions are composed of portions of different elemental methods. These are coupling conditions. As both order of accuracy and N increase, their numbers grow explosively. Table 1 shows their numbers for orders up to five and for each 1-tree listed above with which the coupling conditions are associated. Also contained in the table is the type of order condition; quadrature(Q), subquadrature(SQ), extended subquadrature(ESQ), and nonlinear(NL).[37] It should be noted that there are no more than $N * s(s+1)$ independent Butcher coefficients to satisfy all of these order conditions.

Obviously, without some type of simplifying assumptions, even fourth-order methods appear rather hopeless. The simplist remedy is to require all root or canopy nodes of the N-trees to be the same, or effectively, colorless. In terms of Butcher coefficients, this is done by setting $b_i^{[\nu]} = b_i^{[\mu]}$ (root nodes) or $c_i^{[\nu]} = c_i^{[\mu]}$ (canopy nodes), $\mu, \nu = 1, 2, \cdots, N$. Tables 2 and 3 show the results of these simplifications. Since there is always only one root node for any N-tree yet there may be many canopy nodes, assuming $c_i^{[\nu]} = c_i^{[\mu]}$ generally reduces the number of coupling conditions further than $b_i^{[\nu]} = b_i^{[\mu]}$. Table 4, where all root and canopy nodes are made equal, shows that matters become much more tractable. It may also be seen from this that as long as $b_i^{[\nu]} = b_i^{[\mu]}$ and $c_i^{[\nu]} = c_i^{[\mu]}$, an arbitrary number of independent third-order methods having the same number of stages may be coupled together with no associated coupling error. Tables 2 and 3 show that selecting $b_i^{[\nu]} = b_i^{[\mu]}$ or $c_i^{[\nu]} = c_i^{[\mu]}$ allows second-order error-free coupling of an arbitrary number of schemes. Choosing neither of these assumptions reduces error-free coupling, as seen from Table 1, to first-order.

Table 1: General $\mathrm{ARK_N}$ coupling conditions

Eqn. of cond.	Type of Cond.	Gen. 2-Trees	Gen. 3-Trees	Gen. 4-Trees	Gen. 5-Trees	Gen. N-Trees
$\tau_1^{(1)}$	Q	0	0	0	0	0
$\tau_1^{(2)}$	Q	2	6	12	20	$N(N-1)$
$\tau_1^{(3)}$	Q	4	15	36	70	$N^2(N+1)/2! - N$
$\tau_2^{(3)}$	SQ	6	24	60	120	$N^3 - N$
$q=3$		10	39	96	190	$N(N-1)(3N+4)/2!$
$q\leq 3$		12	45	108	210	$N(N-1)(3N+6)/2!$
$\tau_1^{(4)}$	Q	6	27	76	345	$N^2(N+1)(N+2)/3! - N$
$\tau_2^{(4)}$	ESQ	14	78	252	620	$N^4 - N$
$\tau_3^{(4)}$	SQ	10	51	156	370	$N^3(N+1)/2! - N$
$\tau_4^{(4)}$	SQ	14	78	252	620	$N^4 - N$
$q=4$		44	234	736	1780	$N(N-1)(16N^2+22N+24)/3!$
$q\leq 4$		56	279	884	1990	$N(N-1)(16N^2+31N+42)/3!$
$\tau_1^{(5)}$	Q	8	42	136	345	$N^2(N+1)(N+2)(N+3)/4! - N$
$\tau_2^{(5)}$	ESQ	22	159	636	1870	$N^4(N+1)/2! - N$
$\tau_3^{(5)}$	NL	18	132	540	1620	$N^3(N^2+1)/2! - N$
$\tau_4^{(5)}$	ESQ	22	159	636	1870	$N^4(N+1)/2! - N$
$\tau_5^{(5)}$	SQ	14	87	316	870	$N^3(N+1)(N+2)/3! - N$
$\tau_6^{(5)}$	ESQ	30	240	1020	3120	$N^5 - N$
$\tau_7^{(5)}$	ESQ	30	240	1020	3120	$N^5 - N$
$\tau_8^{(5)}$	SQ	22	159	636	1870	$N^4(N+1)/2! - N$
$\tau_9^{(5)}$	SQ	30	240	1020	3120	$N^5 - N$
$q=5$		196	1458	5960	17805	$N(N-1)(125N^3+179N^2+210N+216)/4!$
$q\leq 5$		252	1737	6804	19795	$N(N-1)(125N^3+243N^2+334N+384)/4!$
$q=6$		876	9372	50432	187280	$N(N-1)(1296N^4+1936N^3+2296N^2+2376N+2400)/5!$
$q\leq 6$		1128	11109	57236	207075	$N(N-1)(1296N^4+2561N^3+3511N^2+4046N+4320)/5!$

Table 2: $\mathrm{ARK_N}$ coupling conditions with $b_i^{[\nu]} = b_i^{[\mu]}$

Eqn. of cond.	Type Cond.	$b_i^{[\nu]} = b_i^{[\mu]}$ 2-Trees	$b_i^{[\nu]} = b_i^{[\mu]}$ 3-Trees	$b_i^{[\nu]} = b_i^{[\mu]}$ 4-Trees	$b_i^{[\nu]} = b_i^{[\mu]}$ 5-Trees	$b_i^{[\nu]} = b_i^{[\mu]}$ N-Trees
$\tau_1^{(1)}$	Q	0	0	0	0	0
$\tau_1^{(2)}$	Q	0	0	0	0	0
$\tau_1^{(3)}$	Q	1	3	6	10	$N(N+1)/2! - N$
$\tau_2^{(3)}$	SQ	2	6	12	20	$N^2 - N$
$q=3$		3	9	18	30	$3N(N-1)/2!$
$\tau_1^{(4)}$	Q	2	7	16	30	$N(N+1)(N+2)/3! - N$
$\tau_2^{(4)}$	ESQ	6	24	60	120	$N^3 - N$
$\tau_3^{(4)}$	SQ	4	15	36	70	$N^2(N+1)/2! - N$
$\tau_4^{(4)}$	SQ	6	24	60	120	$N^3 - N$
$q=4$		18	70	172	340	$N(N-1)(16N+22)/3!$
$q\leq 4$		21	79	190	370	$N(N-1)(16N+31)/3!$
$\tau_1^{(5)}$	Q	3	12	31	65	$N(N+1)(N+2)(N+3)/4! - N$
$\tau_2^{(5)}$	ESQ	10	51	156	370	$N^3(N+1)/2! - N$
$\tau_3^{(5)}$	NL	8	42	132	320	$N^2(N^2+1)/2! - N$
$\tau_4^{(5)}$	ESQ	10	51	156	370	$N^3(N+1)/2! - N$
$\tau_5^{(5)}$	SQ	6	27	76	170	$N^2(N+1)(N+2)/3! - N$
$\tau_6^{(5)}$	ESQ	14	78	252	620	$N^4 - N$
$\tau_7^{(5)}$	ESQ	14	78	252	620	$N^4 - N$
$\tau_8^{(5)}$	SQ	10	51	156	370	$N^3(N+1)/2! - N$
$\tau_9^{(5)}$	SQ	14	78	252	620	$N^4 - N$
$q=5$		89	468	1463	3525	$N(N-1)(125N^2+179N+210)/4!$
$q\leq 5$		110	547	1653	3895	$N(N-1)(125N^2+243N+334)/4!$
$q=6$		418	3084	12548	37376	$N(N-1)(1296N^3+1936N^2+2296N+2376)/5!$
$q\leq 6$		528	3631	14201	41271	$N(N-1)(1296N^3+2561N^2+3511N+4046)/5!$

Table 3: $\mathrm{ARK_N}$ coupling conditions with $c_i^{[\nu]} = c_i^{[\mu]}$

Eqn. of cond.	Type Cond.	$c_i^{[\nu]} = c_i^{[\mu]}$ 2-Trees	$c_i^{[\nu]} = c_i^{[\mu]}$ 3-Trees	$c_i^{[\nu]} = c_i^{[\mu]}$ 4-Trees	$c_i^{[\nu]} = c_i^{[\mu]}$ 5-Trees	$c_i^{[\nu]} = c_i^{[\mu]}$ N-Trees
$\tau_1^{(1)}$	Q	0	0	0	0	0
$\tau_1^{(2)}$	Q	0	0	0	0	0
$\tau_1^{(3)}$	Q	0	0	0	0	0
$\tau_2^{(3)}$	SQ	2	6	12	20	$N(N-1)$
$q=3$		2	6	12	20	$N(N-1)$
$\tau_1^{(4)}$	Q	0	0	0	0	0
$\tau_2^{(4)}$	ESQ	2	6	12	20	$N(N-1)$
$\tau_3^{(4)}$	SQ	2	6	12	20	$N(N-1)$
$\tau_4^{(4)}$	SQ	6	24	60	120	N^3-N
$q=4$		10	36	84	160	$N(N-1)(N+3)$
$q\leq 4$		12	42	96	180	$N(N-1)(N+4)$
$\tau_1^{(5)}$	Q	0	0	0	0	0
$\tau_2^{(5)}$	ESQ	2	6	12	20	$N(N-1)$
$\tau_3^{(5)}$	NL	4	15	36	70	$N^2(N+1)/2!-N$
$\tau_4^{(5)}$	ESQ	2	6	12	20	$N(N-1)$
$\tau_5^{(5)}$	SQ	2	6	12	20	$N(N-1)$
$\tau_6^{(5)}$	ESQ	6	24	60	120	N^3-N
$\tau_7^{(5)}$	ESQ	6	24	60	120	N^3-N
$\tau_8^{(5)}$	SQ	6	24	60	120	N^3-N
$\tau_9^{(5)}$	SQ	14	78	252	620	N^4-N
$q=5$		42	183	504	1110	$N(N-1)(2N^2+9N+16)/2!$
$q\leq 5$		54	225	600	1290	$N(N-1)(2N^2+11N+24)/2!$
$q=6$		164	888	2940	7580	$N(N-1)(6N^3+39N^2+87N+114)/3!$
$q\leq 6$		218	1113	3540	8870	$N(N-1)(6N^3+45N^2+120N+186)/3!$

Table 4: $\mathrm{ARK_N}$ coupling conditions with $b_i^{[\nu]} = b_i^{[\mu]}$ and $c_i^{[\nu]} = c_i^{[\mu]}$

Eqn. of cond.	Type of Cond.	$b_i^{[\nu]} = b_i^{[\mu]}$ $c_i^{[\nu]} = c_i^{[\mu]}$ 2-Trees	$b_i^{[\nu]} = b_i^{[\mu]}$ $c_i^{[\nu]} = c_i^{[\mu]}$ 3-Trees	$b_i^{[\nu]} = b_i^{[\mu]}$ $c_i^{[\nu]} = c_i^{[\mu]}$ 4-Trees	$b_i^{[\nu]} = b_i^{[\mu]}$ $c_i^{[\nu]} = c_i^{[\mu]}$ 5-Trees	$b_i^{[\nu]} = b_i^{[\mu]}$ $c_i^{[\nu]} = c_i^{[\mu]}$ N-Trees
$\tau_1^{(1)}$	Q	0	0	0	0	0
$\tau_1^{(2)}$	Q	0	0	0	0	0
$\tau_1^{(3)}$	Q	0	0	0	0	0
$\tau_2^{(3)}$	SQ	0	0	0	0	0
$q=3$		0	0	0	0	0
$\tau_1^{(4)}$	Q	0	0	0	0	0
$\tau_2^{(4)}$	ESQ	0	0	0	0	0
$\tau_3^{(4)}$	SQ	0	0	0	0	0
$\tau_4^{(4)}$	SQ	2	6	12	20	$N(N-1)$
$q=4$		2	6	12	20	$N(N-1)$
$\tau_1^{(5)}$	Q	0	0	0	0	0
$\tau_2^{(5)}$	ESQ	0	0	0	0	0
$\tau_3^{(5)}$	NL	1	3	6	10	$N(N-1)/2!$
$\tau_4^{(5)}$	ESQ	0	0	0	0	0
$\tau_5^{(5)}$	SQ	0	0	0	0	0
$\tau_6^{(5)}$	ESQ	2	6	12	20	$N(N-1)$
$\tau_7^{(5)}$	ESQ	2	6	12	20	$N(N-1)$
$\tau_8^{(5)}$	SQ	2	6	12	20	$N(N-1)$
$\tau_9^{(5)}$	SQ	6	24	60	120	N^3-N
$q=5$		13	45	102	190	$N(N-1)(2N+9)/2!$
$q\leq 5$		15	51	114	210	$N(N-1)(2N+11)/2!$
$q=6$		63	258	678	1440	$N(N-1)(6N^2+39N+87)/3!$
$q\leq 6$		78	309	792	1650	$N(N-1)(6N^2+45N+120)/3!$
$q=7$		272	1354	4216	10380	$N(N-1)(24N^3+216N^2+616N+976)/4!$
$q\leq 7$		350	1663	5008	12030	$N(N-1)(24N^3+240N^2+796N+1456)/4!$

4.2.4 Error

Error in an elemental q^{th}-order Runge-Kutta scheme contained within an ARK_N method may be quantified in a general way by taking the L_2 principal error norm,[18]

$$A^{(q+1)} = \|\tau^{(q+1)}\|_2 = \sqrt{\sum_{k=1}^{\alpha^{[1]}_{(q+1)}} \left(\tau_k^{(q+1)}\right)^2}, \tag{12}$$

where $\tau_j^{(q+1)}$ are the $\alpha^{[1]}_{q+1}$ error coefficients associated with order of accuracy $q+1$. For embedded schemes, additional definitions are useful such as

$$\hat{\tau}_k^{(p)} = \frac{1}{\sigma}\sum_i^s \hat{b}_i \Phi^{(p)}_{i,k} - \frac{\alpha}{p!}, \quad \hat{A}^{(p+1)} = \|\hat{\tau}^{(p+1)}\|_2, \tag{13}$$

$$B^{(p+2)} = \frac{\hat{A}^{(p+2)}}{\hat{A}^{(p+1)}}, \quad C^{(p+2)} = \frac{\|\hat{\tau}^{(p+2)} - \tau^{(p+2)}\|_2}{\hat{A}^{(p+1)}}, \quad E^{(p+2)} = \frac{A^{(p+2)}}{\hat{A}^{(p+1)}}, \tag{14}$$

where the superscript circumflex denotes the values with respect to the embedded method and the expressions in equation (14) apply for the case $p = q-1$. In the case of ARK_N methods, we generalize the traditional expression for $A^{(q+1)}$ to

$$A^{(q+1)} = \|\tau^{(q+1)}\|_2 = \sqrt{\sum_{k=1}^{\alpha^{[1]}_{(q+1)}} \sum_{n=1}^{n_{N,k,(q+1)}} \left(\tau_{k[n]}^{(q+1)}\right)^2}, \tag{15}$$

where we have used $n_{N,k,(q+1)}$ to denote the total number of ARK_N order conditions arising from all variations of the k^{th} 1-tree at order $(q+1)$. For example, for general N-trees with $N = 5$, $q = 5$, and $k = 15$, Table 1 gives $n_{5,15,6} = 4370 + 5 = 4375$. If root and/or canopy nodes have been set equal then $n_{N,k,(q+1)}$ is given in Tables 2,3, or 4, whichever is appropriate. Since there is generally no reason to assume any particular order condition is more important than another, it is prudent to consider all of them. In certain special cases, one may be able to rule out extended subquadrature, or nonlinear order conditions based on the structure of the ODE at hand. To evaluate the quality of the error controller, one may evaluate $B^{(p+2)}$, $C^{(p+2)}$, and $E^{(p+2)}$ using each of the $\tau_k^{(q+1)}$'s from each elemental method or using all $\tau_{k[n]}^{(q+1)}$'s.

All embedded schemes considered here are applied in local extrapolation mode. For a given order of accuracy, one strives to minimize $A^{(q+1)}$. Following experience with ERK methods having $q = p+1$, $B^{(p+2)}$, $C^{(p+2)}$, and $E^{(p+2)}$ are ideally kept of order unity. Another term,

$$D = \text{Max}\{|a_{ij}^{[\nu]}|, |b_i^{[\nu]}|, |\hat{b}_i^{[\nu]}|, |c_i^{[\nu]}|\}, \tag{16}$$

is usually kept less than 20.

Because of the large number of order conditions associated with the embedded scheme of an ARK_N relative to any one of its elemental methods, we allow for the possibility that the order of the main and embedded methods differ by more than one, as is customary.[36] A priori quality criteria for $q = p+2$ pairs does not appear to have been derived in the context of first-order ODEs.

4.2.5 Simplifying Assumptions

Butcher[6, 15] row and column simplifying assumptions will be helpful in designing methods because they can reduce and simplify the order conditions. By comparing Tables 1 - 4, one quickly surmises that

higher-order methods effectively require the use of the assumptions $b_i^{[\nu]} = b_i^{[\mu]} = b_i$ and $c_i^{[\nu]} = c_i^{[\mu]} = c_i$. Also, without identical root or canopy nodes, application of Butcher simplifying assumptions would become very awkward. Therefore simplifying assumptions are considered in the form

$$C^{[\nu]}(\eta, i): \qquad \sum_{j=1}^{s} a_{ij}^{[\nu]} c_j^{q-1} = \frac{c_i^q}{q}, \quad i = 1, ..., s, \quad q = 1, ..., \eta, \tag{17}$$

$$D^{[\nu]}(\zeta, j): \qquad \sum_{i=1}^{s} b_i c_i^{q-1} a_{ij}^{[\nu]} = \frac{b_j}{q}(1 - c_j^q), \quad j = 1, ..., s, \quad q = 1, ..., \zeta. \tag{18}$$

4.2.6 Stability

The linear stability function for N-additive methods is considered using the equation

$$F(U) \;=\; \sum_{\nu=1}^{N} \lambda^{[\nu]} U, \tag{19}$$

from which it is determined that the stability function is[9]

$$R(z^{[1]}, z^{[2]}, \cdots, z^{[N]}) \;=\; \frac{P(z^{[1]}, z^{[2]}, \cdots, z^{[N]})}{Q(z^{[1]}, z^{[2]}, \cdots, z^{[N]})} \tag{20}$$

$$\;=\; \frac{\mathrm{Det}\left[\mathbf{I} - \sum_{\nu=1}^{N}\left(z^{[\nu]}\mathbf{A}^{[\nu]}\right) + \sum_{\nu=1}^{N}\left(z^{[\nu]}\mathbf{e} \otimes \mathbf{b}^{[\nu]^T}\right)\right]}{\mathrm{Det}\left[\mathbf{I} - \sum_{\nu=1}^{N}\left(z^{[\nu]}\mathbf{A}^{[\nu]}\right)\right]}, \tag{21}$$

where $\mathbf{A}^{[\nu]} = a_{ij}^{[\nu]}$, $\mathbf{b}^{[\nu]} = b_i^{[\nu]}$, $\mathbf{I} = \delta_{ij}$, $z^{[\nu]} = \lambda^{[\nu]}\Delta t$, and $\mathbf{e} = \{1, 1, \cdots, 1\}$. Stability for the embedded method is considered using

$$\hat{R}(z^{[1]}, z^{[2]}, \cdots, z^{[N]}) \;=\; \frac{\mathrm{Det}\left[\mathbf{I} - \sum_{\nu=1}^{N}\left(z^{[\nu]}\mathbf{A}^{[\nu]}\right) + \sum_{\nu=1}^{N}\left(z^{[\nu]}\mathbf{e} \otimes \hat{\mathbf{b}}^{[\nu]^T}\right)\right]}{\mathrm{Det}\left[\mathbf{I} - \sum_{\nu=1}^{N}\left(z^{[\nu]}\mathbf{A}^{[\nu]}\right)\right]}, \tag{22}$$

where $\hat{\mathbf{b}}^{[\nu]} = \hat{b}_i^{[\nu]}$.

For nonlinear stability, which we do not pursue, one may consider

$$F(U) \;=\; \sum_{\nu=1}^{N} \lambda^{[\nu]}(t) U. \tag{23}$$

Defining $\mathbf{Z}^{[\nu]} \;=\; \left\{z_1^{[\nu]}, z_2^{[\nu]}, \cdots, z_s^{[\nu]}\right\}$, the Runge-Kutta K-function is given by

$$K(\mathbf{Z}^{[1]}, \mathbf{Z}^{[2]}, \cdots, \mathbf{Z}^{[N]}) \;=\; \frac{\mathrm{Det}\left[\mathbf{I} - \sum_{\nu=1}^{N}\left\{\mathbf{Z}^{[\nu]}\mathbf{A}^{[\nu]}\right\} + \sum_{\nu=1}^{N}\left\{\mathbf{Z}^{[\nu]}\mathbf{e} \otimes \mathbf{b}^{[\nu]^T}\right\}\right]}{\mathrm{Det}\left[\mathbf{I} - \sum_{\nu=1}^{N}\left\{\mathbf{Z}^{[\nu]}\mathbf{A}^{[\nu]}\right\}\right]}. \tag{24}$$

4.2.7 Conservation

Conservation of certain integrals or invariants may also be of interest in additive Runge-Kutta methods.[1, 16] Similar to the algebraic stability matrix, one may define

$$M_{ij}^{[\nu,\mu]} \;=\; b_i^{[\nu]} a_{ij}^{[\mu]} + b_j^{[\mu]} a_{ji}^{[\nu]} - b_i^{[\nu]} b_j^{[\mu]}. \tag{25}$$

ARK_N methods conserve linear first integrals, in general, only if

$$b_i^{[\nu]} - b_i^{[\mu]} \;=\; 0, \tag{26}$$

and conserve certain quadratic first integrals, in general, only if

$$b_i^{[\nu]} - b_i^{[\mu]} \;=\; 0, \qquad M_{ij}^{[\nu,\mu]} \;=\; O, \tag{27}$$

where $i, j = 1, 2, \cdots, s, \;\; \nu, \mu = 1, 2, \cdots, N, \nu \neq \mu$. Conservation of cubic invariants with Runge-Kutta methods is not possible.[8, 16]

4.2.8 Dense Output

The purpose of dense output[14, 26] has traditionally been to allow high-order interpolation of the integration variables at any point, $\theta\Delta t$, inside the current integration step where $0 \leq \theta \leq 1$. It may also be used, albeit more cautiously, for extrapolating integration variable values to enable better stage value guesses when one or more of the elemental methods is implicit.[22, 33] For an ARK_N method, it is accomplished as

$$U(t^n + \theta\Delta t) \;=\; U^{(n)} + (\Delta t)\sum_{\nu=1}^{N}\sum_{i=1}^{s} b_i^{*[\nu]}(\theta)F^{[\nu]}(U^{(i)}), \quad b_i^{*[\nu]}(\theta) = \sum_{j=1}^{p^*} b_{ij}^{*[\nu]}\theta^j, \quad b_i^{*[\nu]}(\theta = 1) = b_i^{[\nu]} \tag{28}$$

where p^* is the lowest order of the interpolant on the interval $0 \leq \theta \leq 1$. By construction, $b_i^{*[\nu]}(\theta = 0) = 0$. Order conditions, at order m, for the dense output method are given by

$$\tau_{k[n]}^{*(m)} \;=\; \frac{1}{\sigma}\sum_{i}^{s} b_i^* \Phi_{i,k[n]}^{(m)} - \frac{\alpha\theta^m}{m!}. \tag{29}$$

As with the main and embedded formulae, one may write terms like $A^{*(p^*+1)} = A^{*(p^*+1)}(\theta)$ to access the accuracy of the dense output method. When used as an extrapolation device ($\theta > 1$), the stability function, $R^*(z^{[1]}, z^{[2]}, \cdots, z^{[N]}, \theta)$ must be considered,[4]

$$R^*(z^{[1]}, z^{[2]}, \cdots, z^{[N]}, \theta) \;=\; \frac{\text{Det}\left[\mathbf{I} - \sum_{\nu=1}^{N}\left(z^{[\nu]}\mathbf{A}^{[\nu]}\right) + \sum_{\nu=1}^{N}\left(z^{[\nu]}\mathbf{e}\otimes\mathbf{b}^{*[\nu]^T}(\theta)\right)\right]}{\text{Det}\left[\mathbf{I} - \sum_{\nu=1}^{N}\left(z^{[\nu]}\mathbf{A}^{[\nu]}\right)\right]}, \tag{30}$$

where $\mathbf{b}_i^* = b_i^*$. We also assume $b_{ij}^{*[\nu]} = b_{ij}^{*[\mu]} = b_{ij}^*$.

5. Object-Oriented Mesh-Management Frameworks

5.1 Introduction

Multi-physics simulations are often characterised by large extremes of characteristic time and length scales. Such simulation can only be rendered tractable by adopting sophisticated techniques which resolve scales locally. This is true of both time and length scales.

Most fluid dynamical problems are solved in an Eulerian formulation. Such formulations usually resolve disparity of time-scales by adopting an implicit integration scheme and a disparity of length scales by locally refining the grid in regions of high spatial gradients. This can be done by clustering mesh points or by adding "new" mesh points, leading to variable resolution meshes.

In the absence of complex geometrical shapes (i.e. if the domain is rectangular or cuboidal), the method of hierarchial mesh refinement [5] provides a conceptually elegant solution. Mesh cells requiring refinement (determined by some metric) are corralled into rectangular/ cuboidal patches. The sub-domain defined by the bounding box of the patch is regridded with another mesh with a finer resolution (usually a factor of 2). The patches can also be further refined if needed. This recursive refinement strategy establishes a hierarchy of grids whose resolutions differ by some exponent of K, K being the refinement factor. Care is taken that each G_j^{l+1} is completely surrounded by G_i^l, where G is a grid on a "patch", the subscript denotes the patch number and the superscript the grid level. Thus, a boundary cell of a grid can have a neighbor which difers from it by at most 1 level of refinement. Usually a halo of cells (with their properties prolongated from neighboring coarse cells) is kept around a refined patch to simplify the evaluation of derivatives/differences. As the solution field evolves, meshes are refined/coarsened, leading to adaptive mesh refinement.

Implementing such a mesh refinement infrastructure in a large simulation code is a non-trivial task. Apart from introducing complexity, a badly designed mesh-management infrastructure can dramatically slow down a code. These issues are further complicated in a parallel computing environment, since the distributed subdomains frequently undergo very different degrees of refinement, leading to load imbalance. Dynamic load balancing in an adapative mesh environment is a difficult problem and very prone to errors. The implementation of such an infrastructure presents few aspects of interest to computational mathematicans or scientists and can, in most cases, be separated from the actual implementation of numerical schemes or physical model. A number of attempts have been made to provide mesh-management softwares/libraries. In this chapter, we will compare two of the more recent infrastructures, Overture[24] and GrACE[7].

5.2 Characteristics of mesh-management packages

GrACE and Overture represent opposite ends of the spectrum of mesh-management packages. GrACE provides nothing more than mesh-management and load-balancing. The distribution is supplied with some simple (default) bilinear interpolation methods for prolongation and restriction operations and a simple algorithm for corralling "dirty" points (mesh cells needing refinement) into patches. Overture, on the other hand, provides a complete (numerical) environment for developing fluid dynamical simulation software and has a very intuitive syntax.

5.2.1 GrACE

GrACE is a lightweight mesh-management framework. It provides a means of defining a rectangular/cuboidal domain and a callback function to calculate the refinement metric. The domain is first divided equally into P equal parts (P being the number of processors), while minimizing the sub-domain perimeters. These sub-domains are then further refined, as per the metric, leading to a 2D/3D distribution of mini-domains, each defined by its bounding box. These boxes are then numbered in Peano-Hilbert or Morton order, thus achieving a $N^3 \longrightarrow N^1$ mapping while preserving spatial locality. The boxes are then distributed amongst the P processors. The boxes (component meshes) are stored as a Directed Acyclic Graph, where levels in the graph correspond to levels of refienement. Data is defined on these meshes as dynamic, distributed arrays. These arrays are stored in a Directed Acyclic Grid Hierarchy. Load-balancing is determined based on a "weight" function, which, by default, is the number of mesh cells in a component grid. Details of the design and implementation of GrACE can be found in [28, 27, 29, 30, 31]

5.2.2 Overture

Overture is a numerical environment for prototyping fluid dynamical solvers. It is built on the A++ array class library[25] which provides Fortran90–like array operations. Overture comes with sophisticated prolongation/restriction routines, higher-order discretization templates, boundary conditions and the facility to deal with *overset grids* i.e refined grid which can be at an angle to the underlying coarser or overlying finer grid. It can also handle complicated geometries. It is being parallelized, with the parallel–array functionality supplied by P++[25], another array class library. Overture also provides methods, defined on A++ array classes to construct Jacobians in implicit numerical solutions of the Navier-Stokes equations.

5.3 Performance characteristics

In this section, we analyse the performance characteristics of both GrACE and Overture. Two physical problems, Marshak and RDiff (described later) will be solved with both GrACE and Overture and compared. Performance will be characterised under the following heads:

1. Overhead added by the library itself: This is essentially the single-processor performance of the package with load-balancing and adaptiveness turned off. The Marshak problem will be solved using GrACE, Overture and a plain C-code and their performance compared.

2. Parallel scalability of the library. This will be tested using the Marshak problem. P++ will be used instead of Overture.

3. Performance with mesh adaptation turned on. Adaptive mesh refinement is expected to add significant overhead since it exercises the entire mesh-management algorithm. This will be done only for GrACE, since Overture does not suport adaptive mesh refinement yet.

4. Performance with both adaptation and load-balancing turned on. This is the strictest test of the package and will be done using RDiff and GrACE.

5.3.1 Marshak

The Marshak problem solves a 2D scalar diffusion problem

$$\phi_t = \nabla \cdot (D \nabla \phi) \tag{31}$$

where $D = \phi^4/Z^3$, Z being a material constant. $Z = 100$ in the entire domain except in the Ω shaped pipe as shown in Fig. 5.1, where it is $Z = 10$. This creates a large difference in D and a diffusion front is seen to move along the pipe. The right hand side of the equation is approximated by central differences. The time integration is done explicitly using a seond-order Runge-Kutta scheme. $\partial\phi/\partial\mathbf{n} = 0$ is imposed on the right, top and bottom boundaries and $\phi = 10$ on the left boundary. At time $t = 0$, $\phi = 1$ over the entire domain.

5.3.2 RDiff

The RDiff problem is a reaction-diffusion problem and was conceived so as to provide a heavy computational load for the parallel (load-balanced) scaling test.

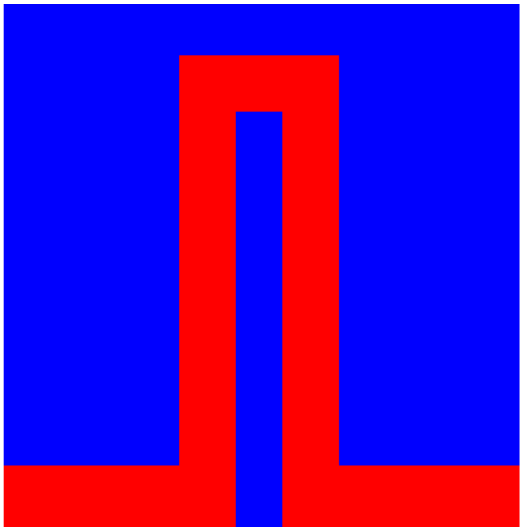

Figure 5.1 A picture of the 2D domain, showing the Ω shaped region of high thermal conductivity.

5.3.2.1 Governing Equations

The physical system can be thought of as a bi-material plate surrounded by a homogeneous mixture of two gases **A** and **B**. The plate consists mainly of a material with very low conductivity, with a "channel" of another material of larger conductivity. Furthermore, the heat conductivity is a function of temperature. Three "dollops" of heat (implemented as Gaussian temperature distributions at $t = 0$) are deposited on the plate, two at the interface between the two materials and one completely within the high-conductivity material. As the heat diffuses through the plate, the (**A** + **B**) gas mixture starts reacting to form **AB**. In doing so, heat is liberated and the temperature increases further. Physically, we expect to see a heat front move along the channel, triggering off the **A** + **B** reactions. Additionally, this front should be accompanied by a visible change in the concentration of **AB**.

The physical system will be characterized by a temperature evolution equation and three equations for the evolution of the **A**, **B** and **AB** concentrations. A number of simplifications have been made - e.g. equal molecular masses for the three gases, equal diffusivities, etc. Convective terms have been ignored. The equations are :

$$\frac{\partial T}{\partial t} = \nabla \cdot (D \nabla T) + S_T \tag{32}$$

$$\frac{\partial \chi_A}{\partial t} = \nabla^2 \chi_A + S_A \tag{33}$$

$$\frac{\partial \chi_B}{\partial t} = \nabla^2 \chi_B + S_B \tag{34}$$

$$\frac{\partial \chi_{AB}}{\partial t} = \nabla^2 \chi_{AB} + S_{AB} \tag{35}$$

where $D = T^4/Z^3$, Z being a material constant. $Z = 10$ in the surrounding material and 100.0 in the

channel. χ_i is the mole fraction of species i, where $i = \{A, B, AB\}$. The terms S_T, S_A, S_B, S_{AB} are source terms due to reaction and are described below.

The system is modeled to have two chemical reactions, each governed by a Arrhenius reaction rate. The system is

$$\begin{array}{lllll} A + B & \xrightarrow{k_1} & AB, & H_1 = 100, & E_1 = 1.0 \\ AB + B & \xrightarrow{k_2} & A + 2B, & H_2 = -10.0, & E_2 = 7.0 \end{array}$$

Here, $k_i = 1.0 \exp(-E_i/T)$, where E_i is a surrogate for the activation energy and H_i for the heat of reaction. Under these conditions, the source terms become

$$\begin{aligned} S_T &= H_1 \chi_A \chi_B k_1 + H_2 \chi_B \chi_{AB} k_2 \\ S_A &= -\chi_A \chi_B k_1 + \chi_{AB} \chi_B k_2 \\ S_B &= -\chi_A \chi_B k_1 + 2\chi_{AB} \chi_B k_2 \\ S_{AB} &= \chi_A \chi_B k_1 - \chi_{AB} \chi_B k_2 \end{aligned} \tag{36}$$

5.3.2.2 Initial and Boundary Conditions

The plate is adiabatic i.e. $\nabla T = 0$ on $\partial\mathcal{D}$, where $\mathcal{D}$ is the domain of integration. $\nabla \chi_i = 0$ is imposed for $i = A, B$ and AB on $\partial\mathcal{D}$ for simplicity.

At time $t = 0$, $\chi_A = \chi_B = 0.5$, $\chi_{AB} = 0$, and a temperature field with three Gaussian hot-spots is initialized as

$$\begin{aligned} T(\vec{r}) &= 10 \sum_{i=0}^{2} \exp(-\eta(\vec{r})_i^2) \\ \eta(\vec{r})_i &= |\vec{r} - \vec{r}_i| \end{aligned} \tag{37}$$

where $\vec{r}_0 = (0.25, 0.1)$, $\vec{r}_1 = (0.5, 0.85)$ and $\vec{r}_2 = (0.75, 0.1)$.

5.3.2.3 Numerical Method

The set of equations Eq. 35 can be cast in the following form

$$\frac{\partial \mathbf{Y}}{\partial t} = \mathbf{R} + \mathbf{S} \tag{38}$$

where $\mathbf{R}$ represents the diffusive terms and $\mathbf{S}$ the reactive terms. The diffusive terms are approximated by 2^{nd}-order finite differences. Thus in 1D,

$$\begin{aligned} \nabla \cdot (D \nabla \phi) &= \frac{1}{(\Delta x)^2} \delta^2 \phi \\ \delta \phi &= \phi_{i+1/2} - \phi_{i-1/2} \end{aligned}$$

The system is integrated following an operator-split RK2 scheme [21, 20]. The predictor-phase of the integration scheme goes as such :

$$\begin{aligned} \mathbf{Y^I} &= \mathbf{Y^n} + \frac{\Delta t}{2}\mathbf{R}(\mathbf{Y^n}) \\ \mathbf{Y^{II}} &= \mathbf{Y^I} + \Delta t \mathbf{S}(\mathbf{Y^I}) \\ \tilde{\mathbf{Y}} &= \mathbf{Y^{II}} + \frac{\Delta t}{2}\mathbf{R}(\mathbf{Y^{II}}) \end{aligned} \tag{39}$$

The corrector phase consists of:

$$\begin{aligned} \overline{\mathbf{Y^I}} &= \mathbf{Y^n} + \frac{\Delta t}{4}\left(\mathbf{R}(\mathbf{Y^n}) + \mathbf{R}(\tilde{\mathbf{Y}})\right) \\ \overline{\mathbf{Y^{II}}} &= \mathbf{Y^n} + \frac{\Delta t}{2}\left(\mathbf{S}(\overline{\mathbf{Y^I}}) + \mathbf{S}(\tilde{\mathbf{Y}})\right) \\ \mathbf{Y^{n+1}} &= \mathbf{Y^n} + \frac{\Delta t}{4}\left(\mathbf{S}(\overline{\mathbf{Y^{II}}}) + \mathbf{S}(\tilde{\mathbf{Y}})\right) \end{aligned} \tag{40}$$

5.3.3 Machines and compilation details

The performance tests were conducted on three different machines

1. Pentium/Pentium cluster : Most of the runs were done on an Intel Pentium cluster. The individual CPUs were Pentium II at 450 MHz, 512 kB combined full-speed cache, running Red Hat 6.0 (Linux kernel 2.2.14). Each machine was equipped with 256 MB of RAM. The cluster of 16 CPUs were connected by a 100 Mbps Fast Ethernet switch. The compilations were done using gcc, with -O2 optimzation. Message passing was done by MPICH, version 1.1.12.

2. Sun UltraSparc : This is a 1GB RAM UltraSparc II running at 300 MHz. This was used for serial Overture runs (and the C-code runs for comparison). The Overture and raw C-code were compiled with the native Sun C++ compiler with -fast -cg92 -PIC as options. The Overture library was compiled with the debugging -g option.

3. LLNL Compass Cluster : These are Compaq AlphaServer 8400 Model 5/440 SMPs using 440 MHz EV6 processors. P++ scaling runs were done using MPICH where message-passing was done using MPICH's shared communicator. The code was compiled using Compaq's own compilers with -fast -tune host options. The P++ library was compiled with the debugging option on.

5.4 Results

5.4.1 Test 1

In this test we will attempt to quantify the serial performance of Overture and GrACE.

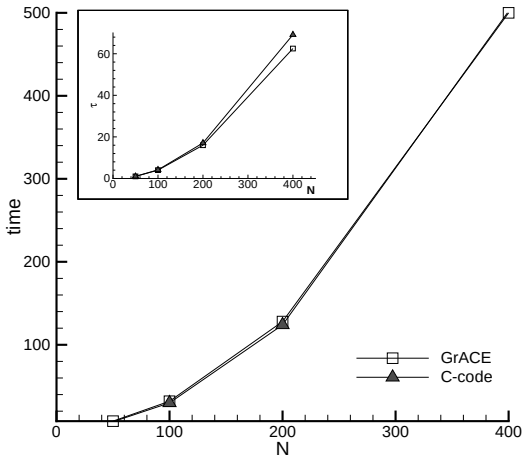

Figure 5.2 Non-adaptive single processor performance for GrACE versus the raw C-code. The "N" on the horizontal axis shows the grid size ($N \times N$) while the vertical axis is the time. Inset: we plot the execution time for a $N \times N$ problem divided by the time for N_0 $times N_0$ grid . Ideally, the line should scale as $(N/N_0)^2$, where $N_0 = 50$.

We ran the Marshak problem on 50×50, 100×100, 200×200 and 400×400 grids using GrACE, Overture and the plain C-code. The time integration was done for 300 timesteps with Δt set with Fourier number of 0.2. GrACE was run on the Pentium cluster (see above) and Overture on the Sun. In Fig. 5.2 we compare GrACE's single-processor performance versus the C-code. The time integration was done for 300 timesteps with Δt set with Fourier number of 0.2. It is clear that the GrACE overhead is small. In the figure inset, $\tau = t_{N \times N}/t_{N_0 \times N_0}$ is around 69 as opposed to the ideal 64 ($(400 \times 400)/(50 \times 50)$).

In Fig. 5.3 the same test (though done on a Sun UltraSparc) shows that Overture takes significantly more time than the optimised C-code. In fact, for a 400×400 grid, the difference is around a factor of 2.

5.4.2 Test 2

In this test we measure the scalability of GrACE, without any adaptiveness of the grid. The test was conducted on the Pentium cluster.

We ran the Marshak problem for 300 timesteps, as above, but on 400×400, 800×800 and 1200×1200 grids on a number of processors. The single processor performance for GrACE, as seen in Fig. 5.2, is very close to a C-code. In Fig. 5.4 we summarise the scalability test results. As the number of processors increases, the computational time becomes comparable to the communications time and we see a divergence from the ideal scaling curve. This occurs at larger numbers of processors for the bigger

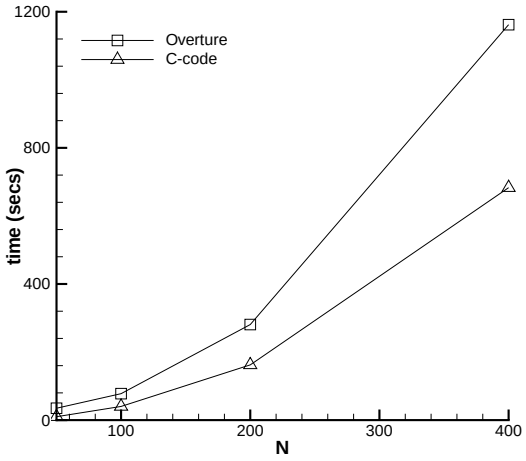

Figure 5.3 Non-adaptive single processor performance for Overture versus the raw C-code. The "N" on the horizontal axis shows the grid size ($N \times N$) while the vertical axis is the time. Overture is seen to take significantly more time.

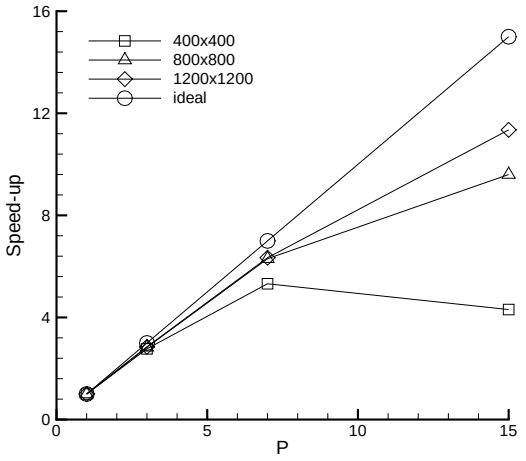

Figure 5.4 Scalability test for GrACE on 400×400, 800×800 and 1200×1200 meshes. P is the number of processors. We see that even for the largest meshes, there is a divergence from ideal scaling as P increases.

meshes. For the smallest mesh, we see a downturn i.e. the communication costs start dominating the computational cost. It is also evident that for larger meshes (or more compute), GrACE could achieve near-ideal scaling.

5.4.3 Test 3

In this test, we compare the costs due to adaptation on a single processor for GrACE. The test was conducted on the Pentium cluster.

We ran the marshak problem on a 100×100 grid for 4000 timesteps, with 0, 1 and 2 levels of refinement. As the simulation proceeds, the number of points on different meshes changes. Further, finer meshes are integrated at a small Δt. Consequently, we calculate a *load* for the entire simulation where 1 unit of load is 1 time-integration of a cell. Thus finer mesh cells contribute more to the load than the coarser meshes. The results are summarised in Fig. 5.5.

The graph shows that the adaptive run takes substantially ($\approx 20\%$) more than the ideal case. This can the directly attributed to the adaptive mesh-management machinery in GrACE.

5.4.4 Test 4

In this test, we exercise GrACE entirely. We solve RDiff on a 400×400 mesh with 2 levels of refinement for 250 timesteps, Δt corresponding to Fourier Number of 0.2. The test was done on the Pentium cluster.

In Fig 5.6 we plot speed-up (scaling) of the problem as the number of processors are increased. We also plot the efficiency of the speed-up. It was clear from Fig. 5.4 that when using a larger number (> 6) of processors, the CPUs were starved for computational work, and parallel speedup dropped off. The scenario seems to be same for the present problem. Poor scaling and efficiency were obtained for $P > 4$.

5.4.5 Test 5

In this test, conducted on LLNL's Compass cluster[11], we tested the scalability and single-processor performance of P++, the parallel array library on which a future parallel version of Overture will be based. The Marshak problem was solved on 200×200, 400×400, 800×800 and 1200×1200 meshes for 100 timesteps on multiple processors. Single processor performance was compared to that of the plain C-code.

The results are in Fig. 5.7. A gradual departure from ideal speed-up as the number of processors increases was seen, as was observed with GrACE. However, the single-processor performance of P++,

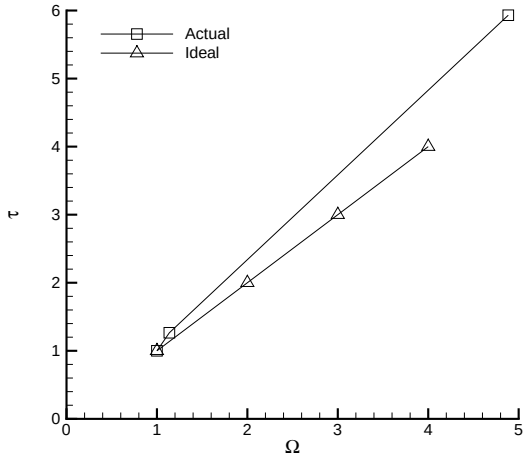

Figure 5.5 Estimation of overhead with GrACE running serially, with adaptation turned on. The results are for 0, 1 and 2 levels of refinement. The load is calculated as 1 time-update of any given mesh cell. The load Ω has been normalised by the load for no adaptation. Thus finer mesh cells contribute proportionately more to the load than coarse meshe cells.

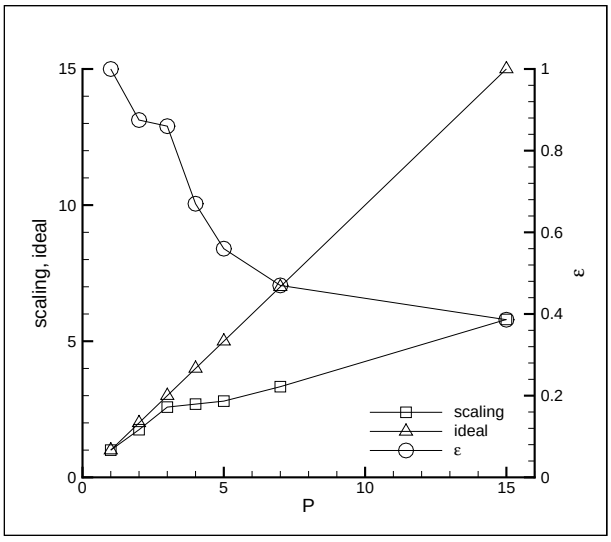

Figure 5.6 Speed-up of the RDiff problem on a 400×400 grid with 2 levels of refinement. We see that the speed-up is much less than ideal and hovers around 40 % when 15 processors are used. We deduce that the bad performance is due to lack of computational work.

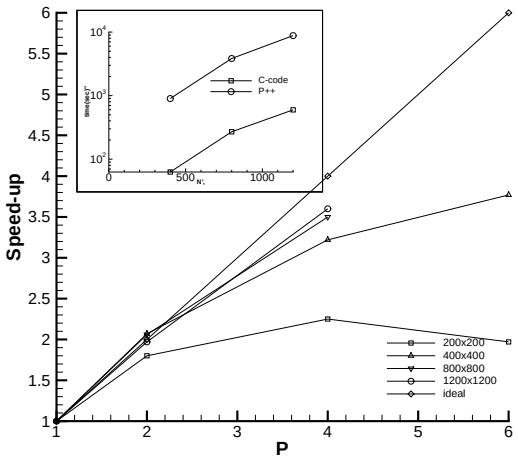

Figure 5.7 Speed-up of the Marshak problem (implemented via P++) on a variety of meshes. We see divergence of the speedup from ideal as the per-processor computational load falls off. Inset: We plot the single processor times for the P++ implementation versus the plain C-code. The C-code is seen to be an order of magnitude faster (the time axis is logarithmic).

shown in the figure insert, was very poor.

5.5 Conclusions

The two adaptive-mesh frameworks, representing the opposite ends of the spectrum provide us with some valuable insights. Light-weight mesh-management frameworks can be used to provide good performance and scalability. However, they suffer from the drawback of providing very little in the way of advanced numerical facilities. A lot of numerical grunge-work (interpolations, boundary conditions etc) will be required when implementing a simulation code within GrACE.

Overture can be best described as a rapid prototying tool for developing simulation codes. Overture's syntax is simple, provides interfaces to many linear solvers and eliminates much of the routine numerical coding involved in writing simulation programs. However, its ease of use and elegant syntax exacts a heavy price on single processor performance. Ongoing efforts to improve the aforesaid include the ROSE preprocessor, which is essentially a source-level loop transformation tool.

6. CCA Post-Processing Tools

6.1 Introduction

The results presented here are the initial steps in the design of a set of codes for post-processing of large data sets. These codes are being developed within the Common Component Architecture (CCA) specification, a standard for component programming currently being developed by the CCA Forum[2, 10]. Component based programming can help reduce complexity, accelerate development of applications, and promote the reuse of code. Other examples of component architectures include CORBA, COM, and JavaBeans. However, the CCA specification is being specifically developed for high performance computing applications. Within CCA there are two types of entities, components and frameworks. Components are independent units of software and the component architecture is a specification of interfaces and rules for combining those components. A framework is a specific implementation of an architecture. CCA specifies a standard way that components and frameworks are built so that components and frameworks from different sources will be compatible.

The primary purpose of the post-processing codes to be assembled in a CCA component will be the identification and analysis of regions of interest, or features, of a data set. In large scale simulations, very often the interesting regions of the data are small subset of that data. These regions may be sparse and difficult to find by casual inspection, especially in 3D data sets. In other cases, there may be numerous regions of interest and their identification and analysis can be very time consuming. Reduction of the amount of the raw data subjected to further analysis is an very important step in post-processing and is the major focus of codes in this component. Additional analysis techniques can be added later by extending this component and/or coupling it with other CCA components.

The starting point for the design was to consider the post-processing CCA component to be a CCA framework populated with CCA components which will each perform the different post-processing tasks of detection, tracking, representation, and simple analysis/statistics of features. The framework would also provide supports, such as interfaces for visualization and other post-processing codes. This structuring allows for a great amount of reuse; the framework will provide support for combining the individual components to perform complicated post-processing tasks, but these components can also be used as simple post-processing tools independent of the framework. And also, because the framework is itself a CCA component, it can be combined with other CCA components.

As a first step in the design of the post-processing framework, a few task components were written in C++ and connected using the the CCA Working Group's test CCA framework, CCAFFEINE[32],

with the goal of exploring how the components should interface and what support the framework will need to provide. Because this was a prototype to focus on component design and interaction, simpler algorithms were implemented for data types existing on a single processor. Once the design is more developed, the algorithms will be rewritten for parallel data structures.

6.2 Results

Two groups of CCA components were created. The first group was a simple test case on a uniform mesh and performed real-time feature detection and tracking of a running simulation based on "The Game of Life". Only two components were used: the simulation and a feature detector/tracker. A second group of components was then developed as a more rigorous test case. These components were used to find and track regions of high heat release rate in a subset of data from a completed combustion simulation which used adaptive mesh refinement (AMR). For this case, the post-processing consisted of three components: a feature manager/track, a feature detector, and a visualization component. An analysis task (a C++ class) was also added but has not yet been made into a CCA component. After completion of these test cases, a preliminary design of the post-processing component framework was developed.

6.3 Game of Life Simulation/Feature Detection

Two CCA components, shown in Figure 6.1, were implemented to develop a few simple post-processing algorithms. The component `LaVie` is a simulation providing data to the `FeatureDetection` component. `LaVie` was developed based on John Conway's game "Life" and Bays' three dimesional adaptation [12, 3]. `LaVie` generates data on a uniform mesh where each cell is either alive (1) or dead (0) and updates the cells based on the state of their neighbors. The features of interest in the data are 'lifeforms' which are distinct groups of 1's. This simulation provided an easy test bed for implementing simple feature detection and tracking routines. The simulation could be initialized with random cells states or a file containing lifeforms with known behaviors. Visualization of the data was performed simply by outputting 1's and 0's in tables of text.

The feature detection component performed two post-processing tasks, feature detection and feature tracking, which were provided by two separate C++ classes (see Figure 1). The class `FeatureManager` manages a list of all features seen during it's lifetime and a list of the features that existed during the last time step examined. `Feature` is a data object that maintains a list of a feature's space-time locations in the data (stored as `FeatureLocation` data objects), as well as it's current existence state,

parents, and children. To obtain features, `FeatureManager` passes a pointer to a time step of the simulation data to the feature detector, `FD_Threshold`, which performs a threshold algorithm on the data and returns a list of `FeatureLocations`. `FeatureManager` performs tracking of the features by comparing this list to the list of locations of existing features, and then updates the `Features` and the existing feature list.

`FD_Threshold` steps over every cell checking if its values are between the two limits `lower_threshold` and `upper_threshold`. If a cell passes this test, a feature has been detected and it's extent is found by first testing it's neighbors, and if they pass the test, their neighbors, and so on. `FD_Threshold` keeps track of cells that have been visited during this testing, so that they will not be tested again. This ensures that every cell is tested only once and that all features found are unique and distinct.

The tracking method implemented by `FeatureManager` considers the following cases when comparing new locations provided by the feature detector: the location may be a continuation of an existing feature, an entirely new feature, a single feature resulting from the merging of multiple features, or one of multiple children splitting from a single feature. Decisions are made based on the overlap in space of the previous and the new locations; overlap in space is assumed to indicate contingency. If a new location does not overlap a previous location, it is a new feature. If a previous and a new location overlap with only each other, the location is a continuing feature, and that feature's location list will be updated. If a new location overlaps with multiple previous locations, it is a child of merging features, and if multiple new locations overlap with a single previous location, they are children of a splitting feature. Children of splits or merges are created as new features and their parent `Features` are appropriately updated and terminated. Finally, if any previously existing features do not overlap a new feature, they are terminated.

The detection and tracking algorithms behaved as expected on both two and three dimensional data sets, as evaluated by visual inspection of output lists of the features compared to text visualization of the life meshes.

6.4 Feature Detection in AMR Combustion Simulation Data

A more developed set of post-processing components was assembled to find and track regions of high heat release rate in a subset of data from a completed combustion simulation which used adaptive mesh refinement (AMR). The two tasks, feature managing and feature detection, were split into separate components, and the algorithms used in the previous case were adapted to work on the refined meshes. The post-processing tasks of visualization and analysis were also added, but only the viz task has been

made into a CCA component.

A diagram of this system of components is shown in Figure 6.2. A driver component read in the data from files and sent this data to the `FeatureManager`. Using the same algorithms as in the previous test case, but adapted for an AMR mesh, `FeatureManger` sends the data to the `FD_Threshold` component which thresholds the data, and returns a list of new `FeatureLocations`. `FeatureManager` tracks the features by matching up these locations with the previous set of existing features. The driver then sends a request to `FeatureManager` to visualize the features, which delegates this task by sending the `Features` to the MatViz component which uses Matlab to visualize the features overlayed on the data. After all of the time-steps have been processed, the `Analysis` class is called to generate the feature statistics of feature size and velocity, and plot them.

The data consisted of the temperature and heat release rates of 16 time-steps of a combustion simulation. Mesh sizes ranged from 3000 nodes and 5 mesh levels to 18,000 nodes and 6 mesh levels stored in 2M files. When thresholded at 30% of the maximum heat release rate for each time step, the system detected a single feature that split into two features at the 10th time step, and these two features existed through the last time-step. Visualization of the features for time-steps 1, 7, 8, 9 10, and 16 are shown in Figures 6.3 through 6.8. Feature area vs. time and velocity magnitude vs. time is shown in Figure 6.9. Velocity magnitude was calculated for each pair of time points by dividing the distance the center of the feature had moved by the different in time. Velocity magnitude was plotted midway between the time steps.

6.5 Preliminary Design of Post-Processing Framework Component

Based on the completed work, a proposed design for the post-processing framework is shown in Figure 6.10. The framework's structure can be divided into four elements: (1) interfaces, (2) the `Director`, (3) the `FeatureManager`, and (4) the post-processing tasks. The Director, `FeatureManger`, and the post-processing tasks will be implemented as CCA components. The `FeatureManager` will still manage a list of the `Features` and a list of existing features, but will no longer contain tracking algorithms. The post-processing tasks will be divided up into four areas: (1) feature detection, (2) feature tracking, (3) feature representation, and (4) feature statistics & analysis. Because a variety of algorithms will be available for each task area, factory components will serve as an interface between the framework and the different algorithms, each implemented as CCA components. This will also allow extension of the set of algorithms implemented without having to rewrite the entire framework. The director will coordinate the different post-processing tasks with each other and the `FeatureManager`, and will receive

commands via an user interface. The `Director` will also communicate with components outside of the post-processing framework through the data, visualization, and analysis interfaces. The visualization interface will allow visualization of the results of the framework's post-processing tasks. The analysis interface will allow export of features to other post-processing tools.

A large variety of algorithms can be implemented within this framework. `FD_Threshold` was just one example of a possible detection algorithm; other algorithms might include finding local minimums or maximums in the data, or regions of high gradients. The tracking algorithms needed to be separate from the `FeatureManager` so that different ones are available to deal with different types of data. For instance, the overlap method used in the implemented `FeatureDectection` Component depends on the contingency of the feature in space between time-steps. If features are moving fast relative to the time-steps, another method will need to be used. For instance, velocity of the feature can be estimated from the previous two time-steps and the expected location of the feature can be estimated and then used in the overlap method. The `FeatureRepresenter` will return a simplified model of the `Feature` useful for performing further analysis and statistics. In the example systems discussed above, the features were represented as bounding boxes. Other useful representations can be obtained by fitting line segments or a Gaussian model to the data. Statistical algorithms can either operate on the representation of the feature, such as plotting the area of the bounding box with time, or can operate on the data in the feature, such as plotting the maximum temperature of each feature at each time step.

6.6 Conclusion

The algorithms shown in Figure 6.12 are just a small sample of the possible post-processing tasks that can be incorporated into the proposed framework. And the power of the entire framework rests not just on the individual tasks it can perform, but on being able to combine these tasks into more complicated post-processing routines. And also, because framework is itself a CCA component, it can be enhanced further by combining it with other CCA components.

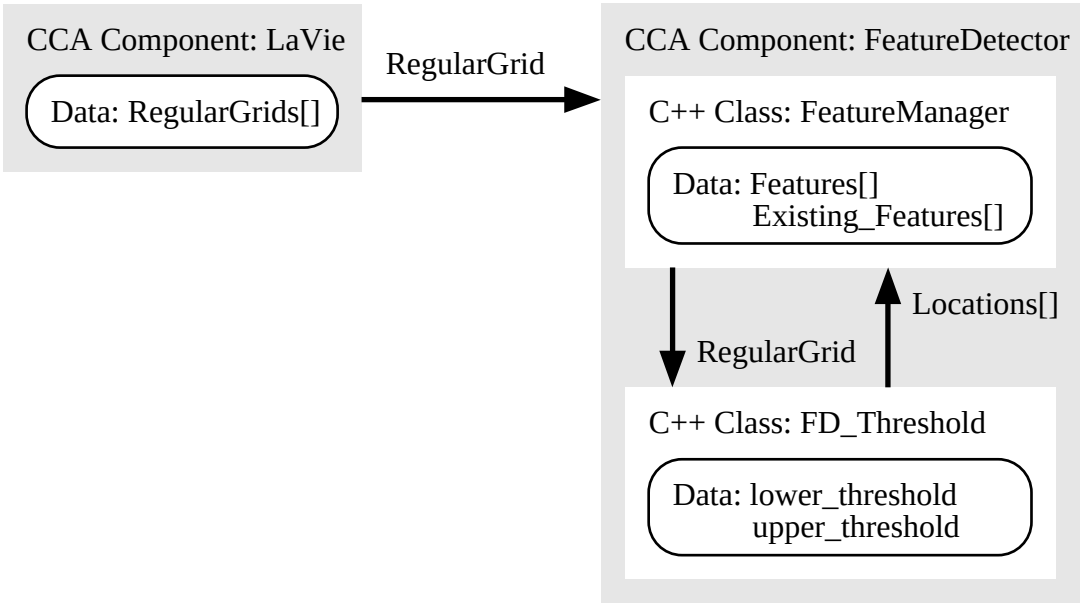

Figure 6.1. Game of Life simulation and feature detection CCA components. Brackets ([]) indicate an array of data objects.

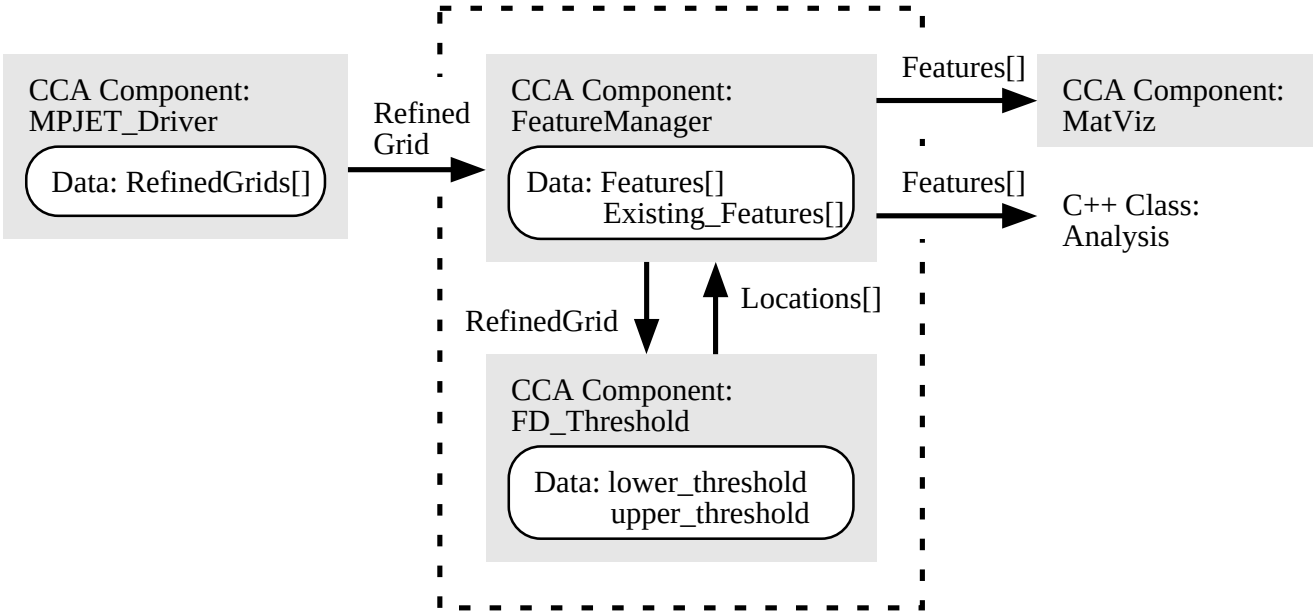

Figure 6.2. Feature detection system for AMR data from a combustion simulation. Brackets ([]) indicate an array of data objects. The componets that are candidates for inclusion in the post-processing framework are located within the dashed box.

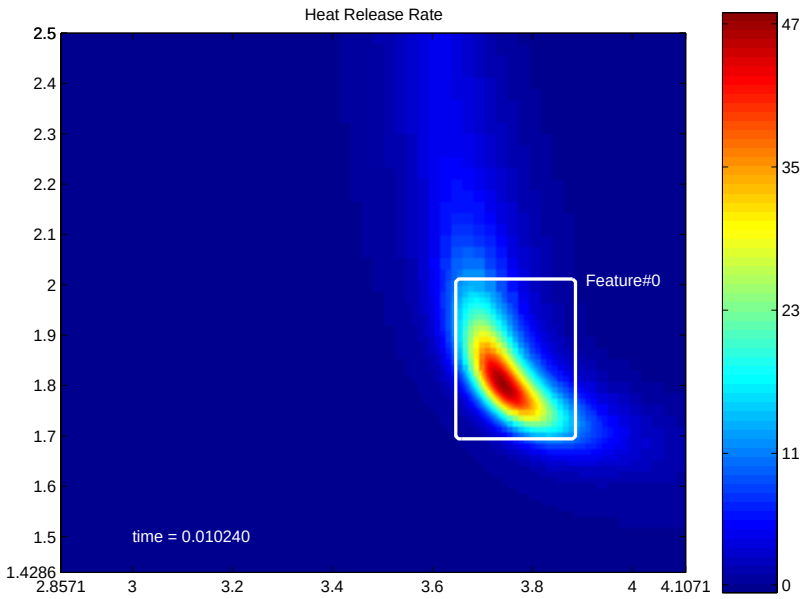

Figure 6.3. Time frame #1 of the combustion simulation. The bounding box of a single feature found by the feature detector by thresholding at 30% of the heat release rate is shown superimposed upon a plot of the heat release rate. The detected feature is a premixed propagating flame front resulting from an ignition at earlier time. The front is propagating down an equivalence ratio gradient, normal to a mixing layer, from a region where the mixture is rich in fuel towards stoichiometric and lean regions.

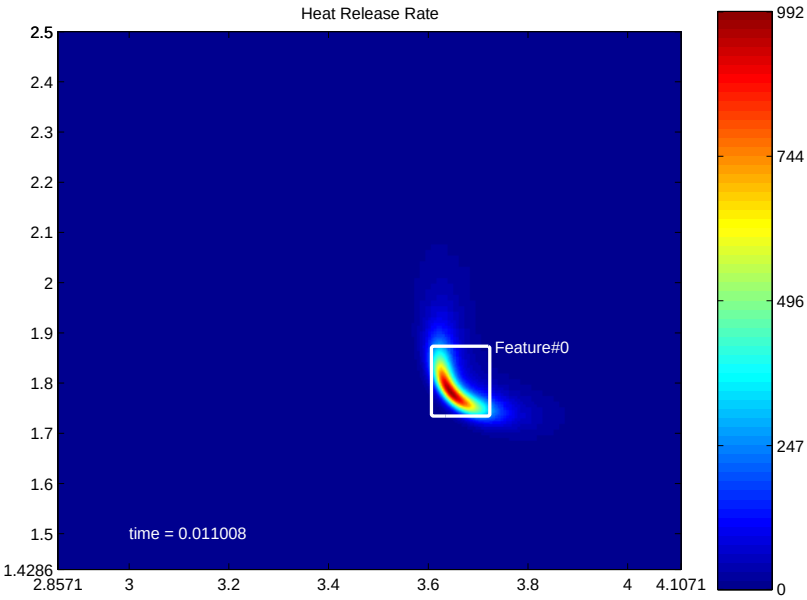

Figure 6.4. Time frame #7 of the combustion simulation. The feature tracker classifies the single feature found by the feature detector to be the same feature as in the first 6 time frames. Note the great increase in peak heat release rate relative to Fig. 6.3. The flame kernel is now in close proximity to the stoichiometric mixture region, and burning and heat release rates are greatly enhanced. The reduction in feature size relative to Fig. 6.3 reflects the higher gradients of heat release rate at the present time.

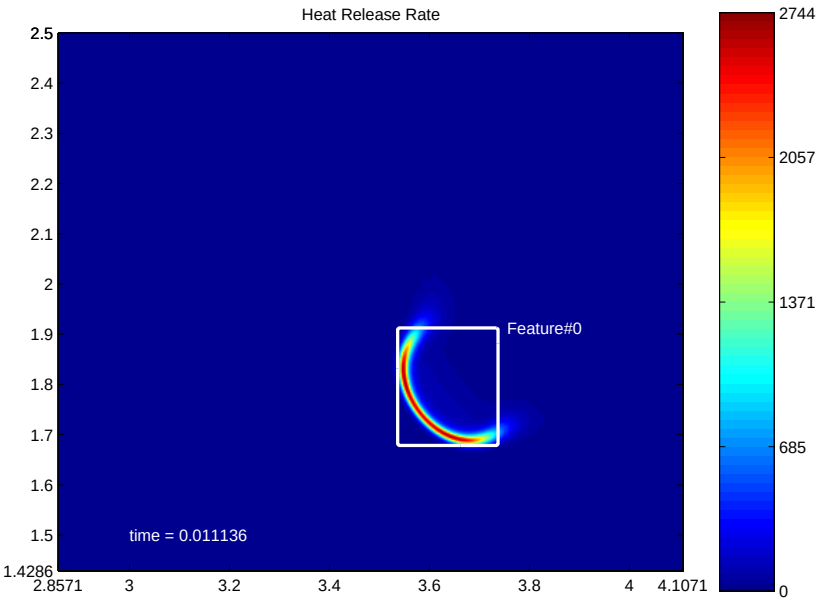

Figure 6.5. Time frame #8 of the combustion simulation. The feature tracker continues to classify the single feature found by the feature detector to be the same feature as is the previous time frame. The front is now expanding rapidly, crossing through the stochiometric mixture region.

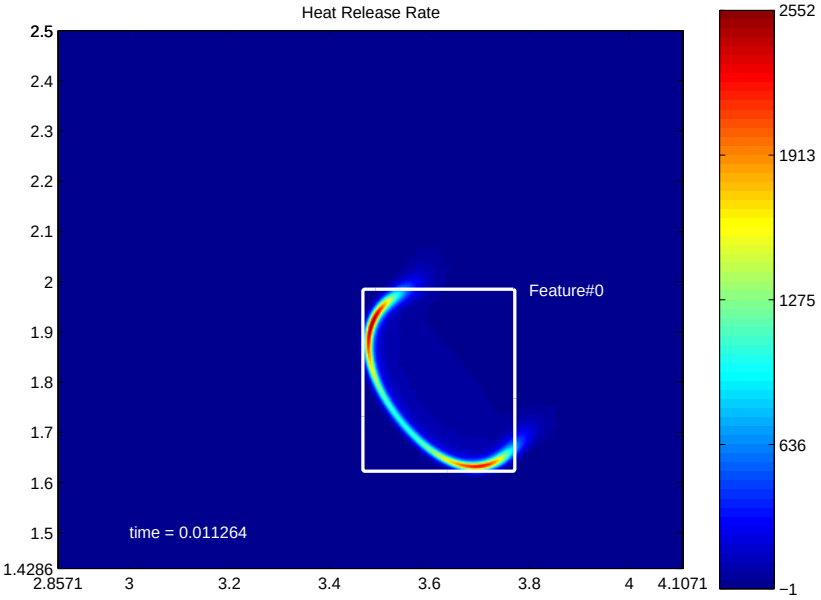

Figure 6.6. Time frame #9 of the combustion simulation. The feature tracker continues to classify the single feature found by the feature detector to be the same feature as is the previous time frame. As the front burns through the stoichiometric mixture, its central region reaches the lean mixture region and exhibits lower burning and heat release rates, while the two edges of the front move sideways along the stoichiometric mixture fraction line in the mixing layer.

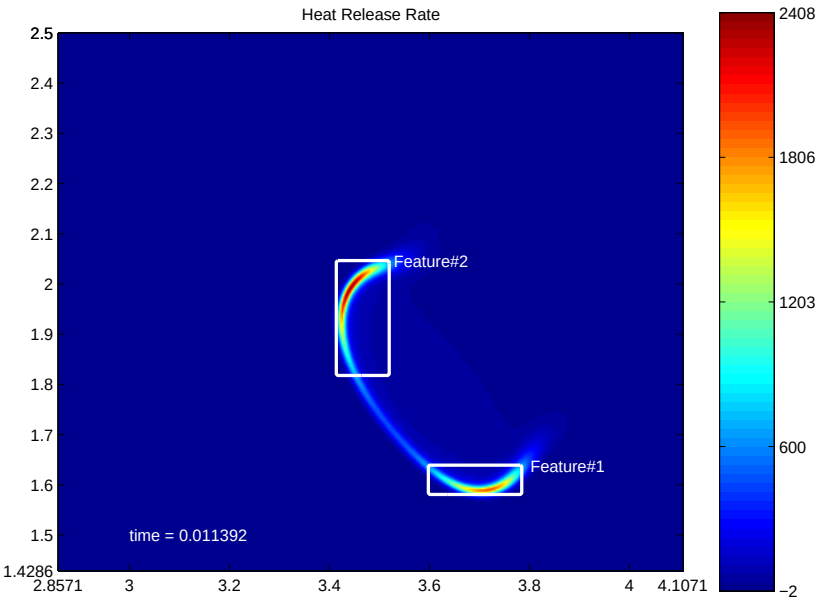

Figure 6.7. Time frame #10 of the combustion simulation. At this point the feature detector finds two separate features and the feature tracker classifies each as a child of `Feature` #0. At this point, the central region of the front is almost extinguished, such that the identification of each edge of the original front as a separate feature is physically meaningful.

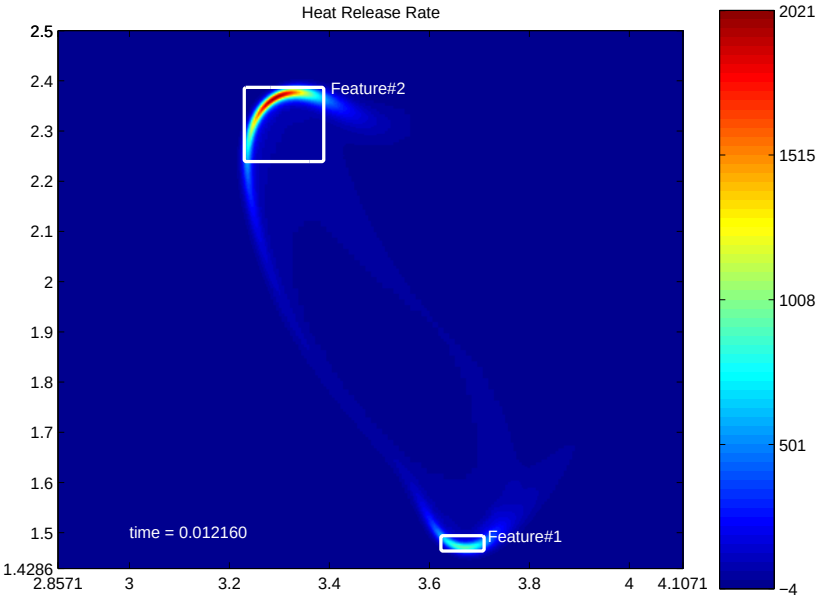

Figure 6.8. Time frame #16 of the combustion simulation. The freature tracker continues to classify the features found by the feature detector to be the same features as is the previous time frame. The two flame fronts are clearly separate entities now, each exhibiting characteristic edge-flame structure, propagating up and down the mixing layer.

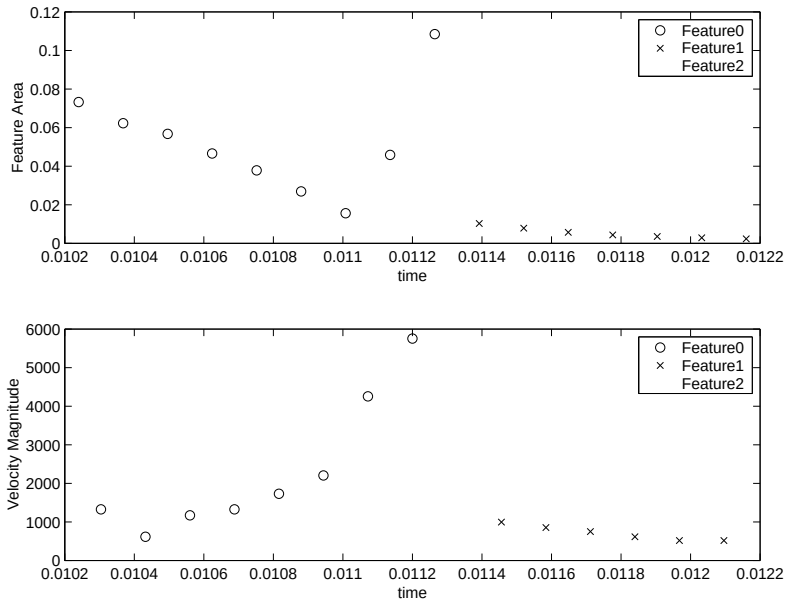

Figure 6.9. Feature statistics. Bounding box area and velocity magintude of the features are plotted vs. time. Velocity magnitude was calculated for successive pairs of timeframes by dividing the distance the center of the feature bounding box moved by the difference in time. Velocity values are plotted at the midpoints of the time frames.

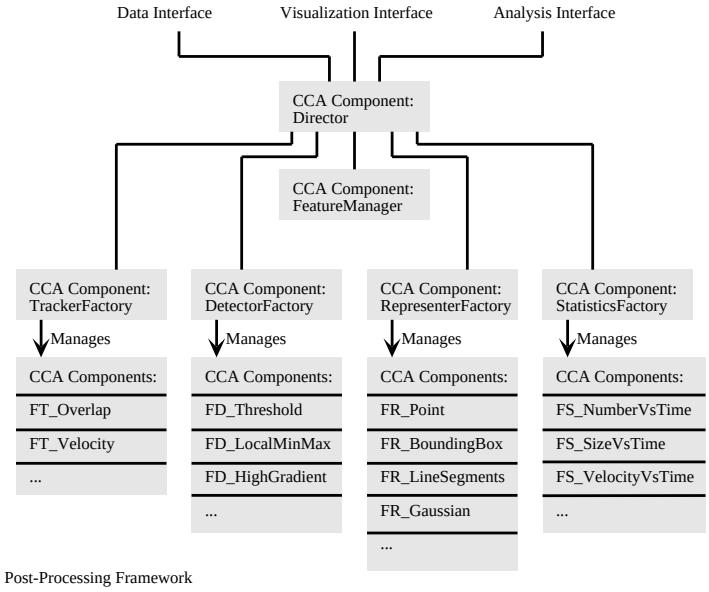

Figure 6.10. Proposed design for a CCA framework to perform post-processing of large scale data.

7. Transient Spherical Flame Model

Pollutant survival in diesel engines depends on the burnout of diffusion flames that are created by the autoignition of fuel-rich parcels of the mixture. Following the suggestion of John Dec (8362), we modified a one-dimensional, transient flame code to compute the burnout of a spherical diffusion flame. The purpose of the simulations is to understand the time evolution of a flame kernel as the heat release rate becomes negligible.

7.1 Model Description

The code solves conservation equations for species and energy of a reacting mixture,

$$\frac{\partial Y_k}{\partial t} = -\frac{\partial}{\partial \psi}\left(\rho r^i Y_k V_k\right) + \frac{\dot{\omega}_k W_k}{\rho} \qquad (k = 1, \ldots, K)$$

$$\frac{\partial T}{\partial t} = \frac{1}{C_p}\frac{\partial}{\partial \psi}\left(\rho r^{2i}\lambda \frac{\partial T}{\partial \psi}\right) - \frac{1}{\rho C_p}\frac{\partial p}{\partial t} - \frac{1}{C_p}\sum_k \rho r^i Y_k V_k C_{p_k}\frac{\partial T}{\partial \psi} - \frac{1}{\rho C_p}\sum_k \dot{\omega}_k h_k W_k$$

where T is temperature, C_p is mean specific heat at constant pressure, ρ is mass density, k is the species index, Y_k are mass fractions, W_k are molecular weights, $\dot{\omega}_k$ are the chemical production rates, V_k are the diffusion velocities, and h_k are the enthalpies. Mass conservation defines the relation between physical space r and the Lagrangian (mass) coordinate ψ,

$$\frac{\partial r}{\partial \psi} = \frac{1}{\rho r^l}$$

with the index $l = 0, 1, 2$ for planar, cylindrical, or spherical geometry. The thermodynamic pressure, p, is uniform in space, $\partial p/\partial \psi = 0$, and the system is closed by the equation-of-state. After discretizing the spatial differences, the system of equations for the dependent variables $S = \{r, p, T, Y_k\}$ is integrated in time following the method-of-lines approach. The boundary conditions are $\partial S/\partial r = 0$ for $\{p, Y_k, T\}$ at both boundaries. The inner boundary is fixed in space at $r(0, t) = 0$, which is the only condition required for the first-order equation for continuity. The physical location of the outer boundary floats to follow the fixed mass of the system as it expands or contracts. The initial outer boundary sets the total system mass. The Lagrangian mass coordinate at any location is established at the initial condition by the integral,

$$\psi(r, 0) = \int_0^r \rho x^l dx.$$

The pressure can be a specified function of time; however, the simulations presented here held the pressure constant in time.

The initial conditions for the flame kernel are specified using a hyperbolic tangent to establish a jump in the species composition and temperature.

$$T(r,0) = T_f + \left(\frac{T_a - T_f}{2}\right)\left(1 + \tanh\left[\frac{(r-r_c)\pi s}{R}\right]\right)$$

$$Y_k(r,0) = Y_{k,f} + \left(\frac{Y_{k,a} - Y_{k,f}}{2}\right)\left(1 + \tanh\left[\frac{(r-r_c)\pi s}{R}\right]\right)$$

The constants r_c and s control the location and width of the transition between the fuel ($Y_{k,f}$) and surrounding air ($Y_{k,a}$). The initial size of the ignition kernel is estimated from images of OH in engine experiments. The initial composition in the kernel is fuel and air premixed to an equivalence ratio of four, which after ignition, creates a fuel-rich product mixture that establishes a diffusion flame with the surrounding air mixture.

The solution procedure uses a regrid scheme to dynamically adapt the mesh to the solution using a weighted coordinate transformation,

$$\frac{dx}{d\eta} W(x,S) = C$$

where the constant is defined by the integral over the domain,

$$C = \frac{1}{N-1}\int_0^L W(x,S)dx$$

using a weighting function that considers the first and second gradients of the solution,

$$W(x,S) = 1 + b_1 \sum_i \left|\frac{dS_i}{dx}\right| + b_2 \sum_i \left|\frac{d^2 S_i}{dx^2}\right|.$$

The mesh is controlled by parameters for the fraction P of points used for adaption ($0 \le P \le 1$), and the ratio R of gradient to curvature, $R = R_1/R_2$, where the fraction of points dedicated to resolving gradient and curvature are defined by,

$$R_1 = \frac{b_1 \int \sum_i \left|\frac{dS_i}{dx}\right| dx}{\int W(x,S)dx} \quad \text{and} \quad R_2 = \frac{b_2 \int \sum_i \left|\frac{d^2 S_i}{dx^2}\right| dx}{\int W(x,S)dx}.$$

The identity $P = R_1 + R_2$ and the definition of R combine to specify R_1 and R_2, from which b_1 and b_2 are computed. The mesh solution uses one loop to compute partial sums in the integrals, and another to interpolate into constant intervals in the generalized coordinate space η. After the new mesh points are defined, the solution values are interpolated onto the new mesh. Note that this interpolation introduces some error, so the regrid is performed only when a criteria for the variation in the solution (measured by the weighting function) becomes significantly larger than an equally spaced distribution in η would produce.

7.2 Results

Computations have been performed using a hydrogen-air reaction mechanism for a spherical flame. The example presented in Figures 7.1-4 demonstrates the flame behavior. The initial H_2-air mixture in the 1 cm diameter kernel is an equivalence ratio of 4. The pressure is 50 atm, constant in space and time. The initial temperature is 1000 K in both the kernel and surroundings.

After the kernel ignites, the diffusion flame forms between the remaining H_2 in the kernel and the surrounding air. Figure 7.1 shows the evolution of the temperature profiles as the flame burns out. The first profile in the sequence is the one with the highest peak temperature and the lowest centerline value. The initial profile is characteristic of a diffusion flame, with the peak temperature at the position where the fuel and oxygen are mixed in stoichiometric proportions. The initially thin diffusion flame thickens as the fuel is consumed from the interior of the flame kernel. By the end of the simulation, the temperature profile is essentially that of a boundary layer solution: the largest temperature is at the centerline, with a smooth decay to the cooler surroundings. Diffusion dominates over what little heat release remains.

Figure 7.2 shows the H_2 mole fraction for the same sequence of solutions. The centerline concentration decreases as more of the H_2 diffuses into the flame. Comparing these curves to the evolution of heat release in Fig. 7.3, there is some 5-10 % hydrogen (by mole) in the kernel when the heat release is essentially complete. The heat release diminishes as the fuel fraction drops to the point where diffusion through the mild species gradients cannot support the flame.

Figure 7.4 shows the OH mole fraction as the kernel burns out. Chemical reaction has not completely shut down at the final time in the simulation, because there remains a local maximum in the OH concentration. While the gradients on either side of this maximum are extremely weak, the existence of this intermediate species suggests that fuel oxidation still proceeds at some slow rate.

The total elapsed time of this simulation is about 2 seconds, which obviously is much longer than exists in an engine. The timescale for this kernel burnout is orders of magnitude off because the simulation neglects two very important phenomena: turbulence and expansion cooling. While the simulation suggests that fuel can survive the burnout process as the kernel collapses on itself, several questions remain. How complete is the fuel consumption in the real environment? Does expansion quench the kernel before it burns out naturally? What effect does local turbulence have on the diffusion flame kernel?

Future extensions of this laminar model to include the enhanced mixing due to turbulent flow will

have to be considered to address these questions. In addition, future computations will need to examine burnout of propane flames, where the fuel behavior is more like the larger hydrocarbons characteristic of diesel fuel.

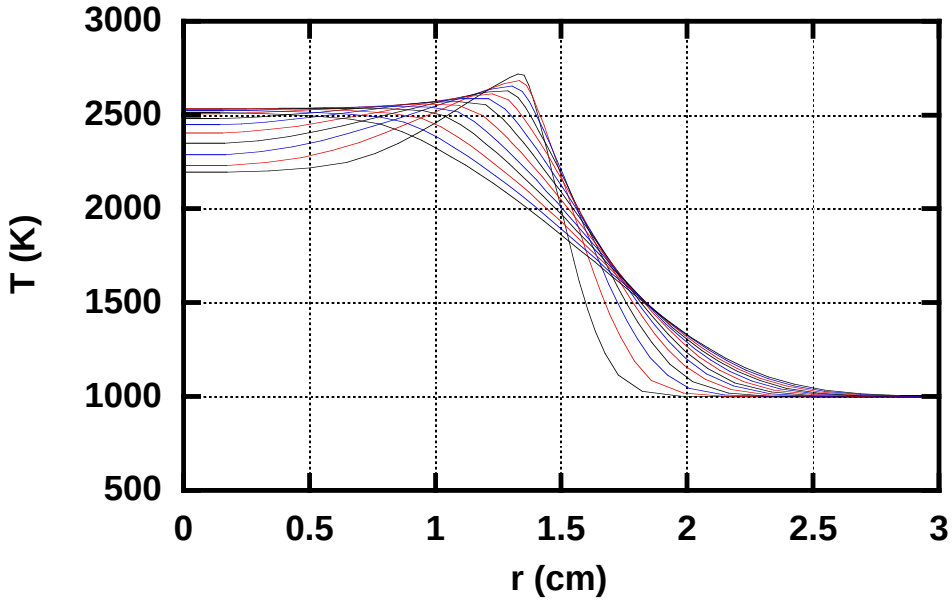

Figure 7.1. Temperature profiles during burnout of hydrogen flame kernel. Time increment between curves is 0.1875 s. First profile has the lowest centerline temperature, corresponding to a point in time after ignition of the kernel, when the diffusion flame has been established.

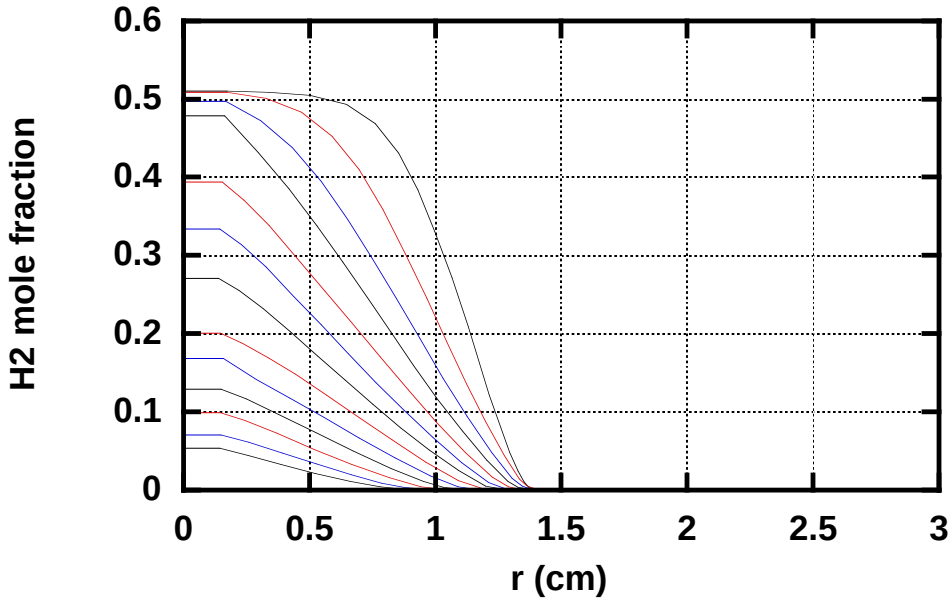

Figure 7.2. Profiles of hydrogen mole fraction during burnout of hydrogen flame kernel. Time increment between curves is 0.075 s. First profile is the right-most curve and has the highest centerline concentration.

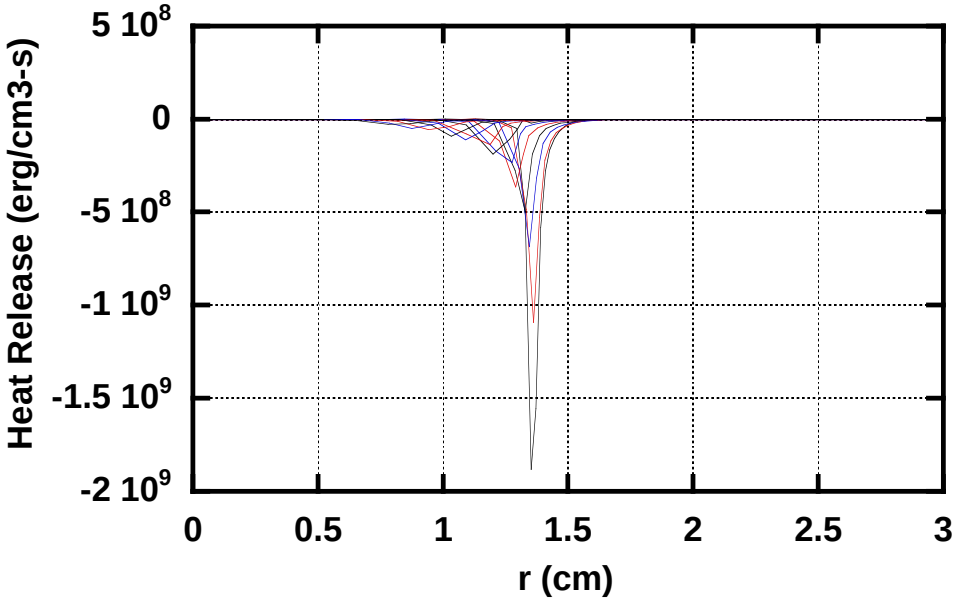

Figure 7.3. Profiles of heat release during burnout of hydrogen flame kernel. Negative values mean exothermic energy release resulting from the chemical reactions. First profile has the maximum peak heat release.

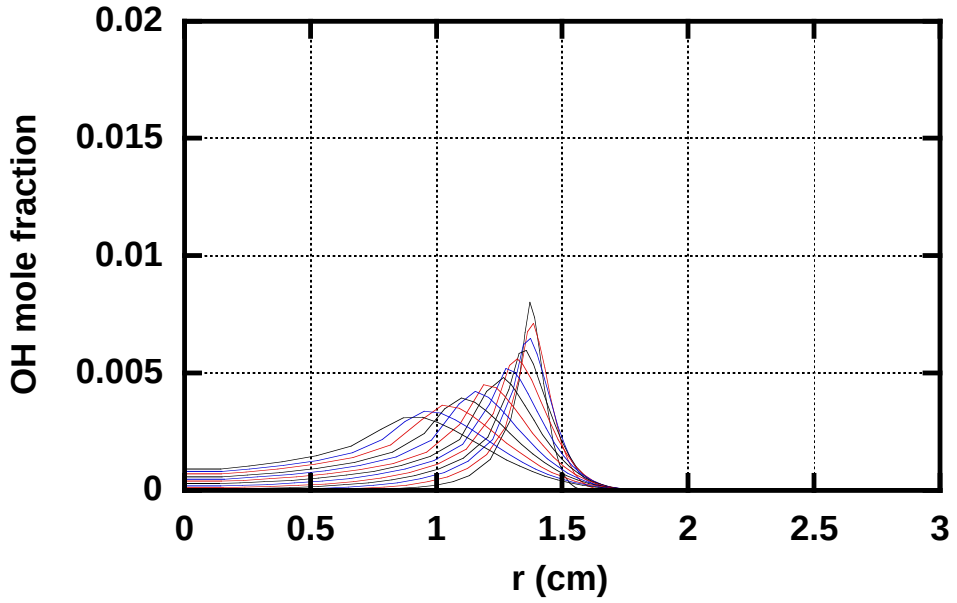

Figure 7.4. Profiles of OH mole fraction during burnout of hydrogen flame kernel. First profile has the largest peak concentration in the the diffusion flame.

8. Materials and Surface Chemistry for Catalytic Conversion

Developing a reliable device-scale model of a catalytic converter for exhaust remediation is fraught with complexities due to a large gap in the knowledge of surface mediated chemical reactions. Even the simple oxidation of CO by O_2 over platinum metal exhibits complex ignition-extinction behavior due to the microkinetic influences of the heterogeneous chemistry. In devising a detailed reaction mechanism for incorporation into fluid solvers, it is imperative to consider the nature of the catalytic surface that participates in the oxidation event. Adsorbed molecules, like CO and O_2, have a number of accessible configurations and energy states which not only alter the reactivity of the adsorbate, but can modify or forcibly reconstruct the surfaces to which they are bonded. For CO in particular, both the reaction order and activation energy for oxidation above and below the ignition temperature are strong functions of CO surface coverage due to, among other things, repulsive adsorbate interactions. These effects are dynamic, and when coupled to the influences of heat and mass transport, produce a reactive system that displays complex-transient and sometimes chaotic behavior.

During the course of the LDRD work we identified potential areas for improvement in our numerical capabilities that, if implemented, would better position Sandia to successfully compete for SSI funding in the Materials and Surface Chemistry Regime. These areas are: (1) the need to develop or acquire CHEMKIN-compatible transient 0-, 1-, and 2-D channel-flow codes; (2) the need to develop or acquire CHEMKIN compatible software tools for parameter optimization and reaction path analysis; and (3) the possibility of modifying or extending the SURFACE CHEMKIN formalisms to include reactions between interfaces or phases possessing non-uniform site densities.

Besides amassing a large body of literature relevant to CO oxidation over platinum-crystal and supported metal surfaces, we numerically investigated the oxidation of hydrogen over palladium. This system was chosen for its mechanistic simplicity, exothermicity, importance in three-way catalytic converter (TWCC) chemistry, and existing experimental data. In the process of our investigation, we identified a potential limitation to the SURFACE CHEMKIN formalism that stems from the fact that SURFACE CHEMKIN was developed primarily for CVD processes and has not been rigorously tested for heterogeneously catalyzed reaction systems such as those observed in TWCCs. In essence, the specification of bimolecular reaction rates in terms of concentration units instead of a site turnover frequency causes the program to calculate an incorrect pre-exponential factor when reaction relationships incorporate different site densities. Program modification is not a trivial exercise and may require a special version of SURFACE CHEMKIN to be created in order to effectively model multi-site interfacial

reactions.

Activities in FY99 also included an evaluation of parameter optimization software. The objective was to develop a capability which could perform a global optimization based on a random initial sampling of parameter space followed by local optimization at the best 10goodness-of-fit criteria. In addition, the parameter optimization routine should provide sensitivity analysis of the best solutions through interfacing with the differential algebraic system solver, DASAC. To evaluate the optimization methodology, we again examined the hydrogen oxidation over palladium mechanism with special attention to the vast amount of experimental data obtained by Sandia for hydrogen adsorption on palladium. We determined that the mechanistic model for hydrogen interaction with palladium was unable to describe the experimental data due to simplifications in model construction.

Finally, we addressed the development of transient numerical codes for simulating plug-flow and well-mixed reactors. Initially, the design equations for PLUG and SENKIN were modified to accommodate time-dependent integration, however, validation of these reactor models is contingent upon modifications to SURFACE CHEMKIN.

9. Closure

The wide range of the work described above is a reflection of the difficulties associated with detailed modeling and understanding of processes occuring in diesel engines and other complex combustion devices, and of the multidisciplinary approaches necessary for tackling them. Massively parallel codes are necessary, along with optimized processor-level coding of cpu-intensive routines. These tools must use specialized/optimized time integration approaches to allow for maximum efficiency in computations of time evolving reacting flow. They also require specialized adaptive mesh refinement techniques to further reduce cpu-time and memory requirements. Such tools are best developed in a modular reusable object-oriented environment that allows for efficient collaborative work by groups of researchers and for code portability. Moreover, ancillary efforts are necessary for development of requisite post-processing tools, and for targeted studies at system components for improvement of submodel constructions and improved understanding of localized phenomena (e.g. flame kernel burnout, catalysis).

The efficient combination of such multidisciplinary approaches, bringing together skills in numerical methods, computer science, combustion, and chemical sciences, is a prerequisite for desired advances in the state of the art of understanding and predictive modeling of diesel engines and other complex systems.

References

[1] A.L. Araújo, A. Murua, and J.M. Sanz-Serna, *Symplectic methods based on decompositions*, SIAM J. Num. Anal., **34**, 5, (1997) 1926-1947.

[2] Armstrong, Rob, Dennis Gannon, Al Geist, Katarzyna Keahhey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski, *Toward a Common Component Architecture for High-Performance Scientific Computing*, Proceedings of the 1999 Conference on High Performance Distributed Computing (HPDC '99), Rendo Beach, CA, 4-6 Aug 1999.

[3] C. Bays, *Candidates for the Game of Life in Three Dimensions*, Complex Systems, **1**, 373-400, (1987).

[4] A. Bellen and M. Zennaro, *Stability properties of interpolants for Runge-Kutta methods*, SIAM J. Num. Anal., **25**, 2, (1988) 411-432.

[5] M. J. Berger and P. Collela Local adaptive mesh refinement for shock hydrodynamics. *J. Comp. Phys.*, 82:64–84, 1989.

[6] J.C. Butcher, *The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods*, John Wiley & Sons, Chichester (1987).

[7] http://www.caip.rutgers.edu/~parashar/TASSL/research.htm

[8] M.P. Calvo, A. Iserles, and A. Zanna, *Numerical solution of isospectral flows*, Math. Comp., **66(220)**, (1997) 1461-1486.

[9] M.P. Calvo, J. de Frutos, and J. Novo, *Linearly implicit Runge-Kutta methods for advection-reaction-diffusion equations*, Report 1999/3, Department of Appl. Math. & Comp., University of Valladolid, Spain (1999).

[10] Common Component Architecture Forum Web Page: `http://www.acl.lanl.gov/cca`.

[11] http://www.llnl.gov/sccd/lc/FAST.resources.html

[12] M. Gardner, *On Cellular Automata, Self-reproduction, the Garden of Eden, and the Game of 'Life'*, Scientific American, **224**, 2, 112-117, (1971).

[13] E. Hairer, *Order conditions for numerical methods for partitioned ordinary differential equations*, Numer. Math., **36**, 4, (1981) 431-445.

[14] E. Hairer, S.P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations* I, *Nonstiff Problems*, 2ed., Springer-Verlag, Berlin (1993).

[15] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations* II, *Stiff and Differential-Algebraic Problems*, 2ed., Springer-Verlag, Berlin (1996).

[16] E. Hairer, *Numerical Geometric Integration*, Unpublished Notes, Section de Mathématiques, Université de Genève, Genève (1999).

[17] Z. Jackiewicz and R. Vermiglio, *Order conditions for partitioned Runge-Kutta methods*, Applic. Math., **45**, 4 (2000) 301-316.

[18] C.A. Kennedy, M.H. Carpenter, and R.H. Lewis, *Low-storage, explicit Runge-Kutta schemes for the compressible Navier-Stokes equations*, Appl. Num. Math., **35**, 3 (2000) 177-219.

[19] H.N. Najm, R.W. Schefer, R.B. Milne, C.J. Mueller, K.D. Devine, and S.N. Kempka *Numerical and Experimental Investigation of Vortical Flow-Flame Interaction* Technical Report SAND98-8232, Sandia National Laboratories (1998).

[20] J.Ray, H.N. Najm, R.B. Milne, K.D. Devine, and S.N. Kempka, *Triple Flame Structure and Dynamics at the Stabilization Point of an Unsteady Lifted Jet Diffusion Flame*, Proc. Comb. Inst, **28**, in press, 2000.

[21] H.N. Najm, P.S. Wyckoff, and O.M. Knio, *A Semi-implicit Numerical Scheme for Reacting Flow*, J. Comp. Phys., **143**, 381-402 (1998).

[22] S.P. Nørsett and P.G. Thomsen, *Local error control in SDIRK-methods*, BIT, **26**, 1, (1986) 100-113.

[23] N. Nystrom and R. Subramanya, *Sandia CHEMKIN-III Auto-CHEMKIN Performance Analysis*, Report PSC-CHEMKIN-PA-2.0, Pittsburgh Supercomputing Center, Carnegie Mellon University, (1999).

[24] http://www.llnl.gov/CASC/Overture/

[25] http://www.llnl.gov/CASC/Overture/henshaw/overtureDocumentation/overtureDocumentation.html

[26] B. Owren and M. Zennaro, *Order barriers for continuous explicit Runge-Kutta methods*, Math. Comp., **56(194)**, (1991) 645-661.

[27] M. Parashar A programming environment for adaptive multi-resolution, multi-physics applications *in preparation, IEEE Computational Science*, April 2000.

[28] M. Parashar and J. C. Browne Distributed dynamic data-structures for parallel-adaptive mesh-refinement. *Proceedings of the International Conference for High Performance Computing*. pp 22-27, December 1995.

[29] M. Parashar and J.C. Browne A computational infrastructure for parallel adaptive algorithms *IEEE Computational Science*, August 1999

[30] M. Parashar, J.C. Browne et al A common data management infrastructure for parallel adaptive algorithms for PDE solutions. *Proceedings of Supercomputing 97*, November 1997.

[31] M. Parashar and J. C. Browne Object-oriented programming abstractions for parallel adaptive mesh refinement. *Parallel Object-Oriented Methods and Applications (POOMA)*, February 1996.

[32] Remote UI wire protocols, pre-XML, for the CCA component framework CCAFFEINE, `http://z.ca.sandia.gov/ baallan/drafts/doc/designUI.html`.

[33] J. Sand, *RK-predictors: Extrapolation methods for implicit Runge-Kutta formulae*, DIKU Report Nr. 88/18, Datalogisk Institut, Københavns Universitet, Copenhagen (1988)

[34] R. Subramanya and R. Raghurama, *SANDIA R3 - DNS Code Requirements, Analysis and Design*, Pittsburgh Supercomputing Center, (1999), Report, PSC-CRF-R3-RAD-1.0, Carnegie Mellon Univ, Pittsburgh, PA.

[35] R. Subramanya and R. Raghurama, *SANDIA (2D/3D) - DNS Codes Final Report*, Pittsburgh Supercomputing Center, (1999) Report, PSC-Sandia-FR-3.0, Carnegie Mellon Univ, Pittsburgh, PA.

[36] Ch. Tsitouras and S.N. Papakostas, *Cheap error estimation for Runge-Kutta methods*, SIAM J. Sci. Comp., **20**, 6 (1999) 2067-2088.

[37] J.H. Verner, *High-order explicit Runge-Kutta pairs with low stage order*, Appl. Num. Math., **22**, 1-3 (1996) 345-357.

[38] P.S. Wyckoff and H.N. Najm, Private Communication, Sandia National Laboratories, (1998).

Unlimited Disribution

External Distribution:

N. Nystrom

Pittsburgh Supercomputing Center

Carnegie Mellon Universtiy

Mellon Institute Bldg.

4400 Fifth Ave.

Pittsburgh, PA 15213

R. Subramanya

Pittsburgh Supercomputing Center

Carnegie Mellon Universtiy

Mellon Institute Bldg.

4400 Fifth Ave.

Pittsburgh, PA 15213

R. Reddy

Pittsburgh Supercomputing Center

Carnegie Mellon Universtiy

Mellon Institute Bldg.

4400 Fifth Ave.

Pittsburgh, PA 15213

Internal Distribution:

5	MS-9051	H. Najm
5	MS-9051	J. Chen
5	MS-9051	J. Grcar
1	MS-9217	R. Armstrong
1	MS-9051	J. Ray
1	MS-9051	W. Koegler
1	MS-9051	C. Kennedy
1	MS-9053	A. Lutz
1	MS-9052	T. McDaniel

1	MS-9052	M. Allendorf
1	MS-1140	N. Varnado
1	MS-9054	B. McLean
1	MS-9054	F. Tully
1	MS-9012	P. Nielan
3	MS-9018	Central Technical Files, 8940-2
1	MS-0899	Technical Library, 4916
1	MS-9021	Classification Office, 8511/Technical Library, MS-0899, 4916
1	MS-9021	Classification Office, 8511 For DOE/OSTI
1	MS-0188	D. Chavez, LDRD Office, 4001