ARL-CR-0843 • JAN 2020

DEVCOM ARMY RESEARCH LABORATORY

Summary of User Productivity Enhancement, Technology Transfer, and Training (PETTT) Refactoring Assistance for Chemical Kinetics Analysis Tools at CCDC Army Research Laboratory

by Christopher P Stone

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-CR-0843 ● JAN 2020

Summary of User Productivity Enhancement, Technology Transfer, and Training (PETTT) Refactoring Assistance for Chemical Kinetics Analysis Tools at CCDC Army Research Laboratory

Christopher P Stone
Science Applications International Corporation

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
January 2020	Contractor Report	2 September 2017 – 27 August 2019

4. TITLE AND SUBTITLE
Summary of User Productivity Enhancement, Technology Transfer, and Training (PETTT) Refactoring Assistance for Chemical Kinetics Analysis Tools at CCDC Army Research Laboratory

5a. CONTRACT NUMBER
GS04T09DBC0017

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Christopher P Stone

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Science Applications International Corporation
12010 Sunset Hills Road
Reston, VA 20190

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-CR-0843

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Department of Defense (DOD) High Performance Computing Modernization Program (HPCMP)
US Army Engineer Research and Development Center
Vicksburg, MS

10. SPONSOR/MONITOR'S ACRONYM(S)
DOD HPCMP

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
ORCID ID: Christopher P. Stone, 0000-0002-9621-5334

14. ABSTRACT
Over the past two years, User Productivity Enhancement, Technology Transfer, and Training (PETTT) has refactored legacy FORTRAN programs that US Army Combat Capabilities Development Command Army Research Laboratory researchers employ to model the decomposition and combustion of energetic materials. PETTT employed OpenMP threading to incorporate node-level parallelism into four serial applications, significantly reducing their runtimes on platforms at Department of Defense High Performance Computing Modernization Program Supercomputing Resource Centers. This report summarizes the methodologies employed and the performance improvements that were achieved.

15. SUBJECT TERMS
Productivity Enhancement, Technology Transfer, and Training, PETTT, High Performance Computing Modernization Program, HPCMP, Dedicated Short Range Communications, DSRC

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Michael Lasinski
Unclassified	Unclassified	Unclassified	UU	34	19b. TELEPHONE NUMBER (Include area code) (410) 278-9910

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures iv

List of Tables iv

Acknowledgments v

1. Introduction 1

2. Algorithm Descriptions 2

2.1 TMM3 2

2.2 SENKIN 5

2.3 OPPDIF 6

2.4 PREMIX 8

3. Optimization Methods 8

4. Results 14

4.1 TMM3 14

4.2 SENKIN 15

4.3 PREMIX 16

4.4 OPPDIF 19

5. Summary and Conclusions 22

6. References 23

List of Symbols, Abbreviations, and Acronyms 25

Distribution List 27

List of Figures

Fig. 1 PREMIX benchmark using serial and OpenMP parallelism on Centennial (ARL DSRC). Nitrocellulose–nitroglycerin mechanism with 514 species and 937 reactions. Runtime categories shown are the RHS function time (Fun), Jacobian matrix assembly (Jacob), Banded matrix factorization (LU), and all other costs summed (Other). Cases include: single (serial) and multicore (OMP); vectorization disabled (no-vec), standard compiler vectorization (vec), and vectorization with enhanced (BLAS) kernels (vec-opt); parallel RHS function (ParFun); and replacing Linpack functions with Lapack (Lapack). Speed-up is relative to single-core execution with standard compiler vectorization enabled (serial-vec). 17

Fig. 2 Scaling of OPPDIF on Onyx with OpenMP parallel Jacobian matrix assembly, using one CPU socket with up to 22 cores and one thread per core..... 20

Fig. 3 Runtime (minutes) and speed-up of OPPDIF using USC mechanism on Centennial using parallel Jacobian evaluation and sequential or parallel matrix factorization with MKL 21

List of Tables

Table 1 HR model performance on Excalibur: wall-clock times (seconds) for serial and parallel runs on Excalibur..... 16

Acknowledgments

This material is based upon work supported by, or in part by, the Department of Defense (DOD) High Performance Computing Modernization Program (HPCMP) under User Productivity Enhancement, Technology Transfer, and Training contract number GS04T09DBC0017. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the DOD HPCMP.

1. Introduction

To predict the responses of energetic materials (EMs) to a variety of conditions pertinent to their life cycle, US Army Combat Capabilities Development Command (CCDC) Army Research Laboratory (ARL) researchers develop and apply physics-based models of the phenomena that underlie them. Relevant to both performance and vulnerability, responses are strongly related to the EM's rate of decomposition. Therefore, the models must include a chemical kinetics mechanism to compute that rate. Moreover, to provide insights with the potential to inform the development of propellant and explosive formulations, the mechanisms must be "detailed" (i.e., include parameterized formulae for computing the rate coefficients of relevant elementary reactions and the thermochemical properties of the species involved in those reactions).

For applications for which ARL researchers believe reliable formulae for computing reaction rates and thermochemical properties need to be established, they employ state-of-the-art computational approaches to develop them. Over the past decade, significant increases in computing power and advances in computational methods have enabled them to parameterize formulae for reactions involving increasingly large molecules. Added to existing mechanisms and integrated into computational fluid dynamics (CFD) models, they have led to insights into phenomena ranging from the ignition delays in hydrazine-alternative hypergolic propulsion systems,[1] to the prediction of burning rates for disruptive EMs that have yet to be produced in quantities sufficient for them to be measured.[2] However, the time and effort required to establish and validate such formulae has prevented them from being widely developed and used. It typically takes 6–12 months to develop formulae for a new application. Therefore, their potential to positively impact munitions development programs with expected life cycles of 1–3 years can be limited.

A number of factors contribute to the length of time it takes to develop and apply parameterizations for reaction-rate coefficients and species thermochemical properties for new reactions. One is the validation process. Although ARL researchers employ state-of-the-art computational approaches to develop parameterizations, it is only when they are included in a model as part of a detailed mechanism that they can be critically evaluated. Canonical (idealized) combustion models are invaluable for this purpose. However, current mechanisms have as many as eight times more elementary reactions and species than those of a decade ago, and the computational costs associated with these models increase quadratically to cubically with the number of species (NS), making them $O(10–100)$ times more expensive. Such costs also have to be borne for other applications of these models,

including predicting EM burning rates,[2] their response to conditions like slow cook-off,[3] and mechanism reduction.[4–6] Outpacing increases in single-core computing power over that same time period, the (wall) times needed to run the original serial implementations of a core set of programs employed for these purposes were becoming untenable.

Believing wall times could be reduced by refactoring the programs to harness the parallel processing of High Performance Computing Modernization Program (HPCMP) DOD Supercomputing Resource Center (DSRC) platforms, but not having the expertise necessary to do so, ARL researchers requested User Productivity Enhancement, Technology Transfer, and Training (PETTT) reactionary assistance (RA). Through the incorporation of node-level parallelism via OpenMP threading, significant reductions in the runtimes of four programs were achieved. The performance improvements increase the likelihood that mechanism development will be undertaken in the first place, and increase the potential of the mechanisms that are created to accelerate munition development because more time is available to exploit them.

2. Algorithm Descriptions

All the programs for which refactorings were sought were ARL-customized versions of subroutines in the venerable CHEMKIN package for the analysis of gas-phase reaction dynamics. CHEMKIN is now a commercial[7] product; however, all work involved precommercial versions of subroutine packages developed by the US Department of Energy. Referred to herein by their CHEMKIN name, they included SENKIN, OPPDIF, and PREMIX. SENKIN computes the time evolution of a homogeneous reacting gas mixture in a closed system. (Its results are referred to hereafter as homogeneous reactor [HR] simulations.) SENKIN is also employed as a subroutine in a mechanism reduction program called TMM3.[4–6] OPPDIF computes species and temperature profiles for steady-state opposed-flow diffusion flames. PREMIX computes species and temperature profiles for steady-state burner-stabilized and freely propagating premixed laminar flames. It is also employed as a subroutine for a program employed to predict EM burning rates called CYCLOPS.[2,8] Aspects of these programs that were relevant to their refactoring are as follow.

2.1 TMM3

TMM3 was the first program that was refactored. It produces skeletal/reduced mechanisms from a full (i.e., comprehensive) mechanism. It accomplishes that by the following:

1. Randomly reordering the full mechanism's reactions.
2. Sequentially eliminating individual reactions from the randomized mechanism on a trial basis.
3. Running (SENKIN) HR simulations with the (trial) mechanism created by each elimination.
4. Permanently eliminating a reaction if changes to selected results of the simulations do not exceed specified criteria.

If all reactions involving a chemical species are eliminated, that species is also removed from the mechanism.

The first (major) step of the reduction process—randomly ordering the sequence of reactions in the full mechanism—is performed because the impact that the elimination of a reaction has on an HR simulation can depend on whether other reactions have already been eliminated. Because of this dependence, ARL researchers attempt to increase the probability that a viable skeletal mechanism will be produced by applying the screening process to many different orderings; 40 to 100 will typically be processed. With appropriate job submission scripts, different orderings could be (and were) reduced in parallel on HPCMP DSRC platforms, but each reduction was done in serial. Given this framework, I focused on reducing the runtime for a given ordering by exploiting parallelism to reduce the runtimes of the individual HR simulations.

Several factors determine the computational cost of HR simulations run for a TMM3-based reduction. The HR model is defined by a set of ordinary differential equations (ODEs).

$$\frac{\partial \boldsymbol{u}}{\partial t} = f(\boldsymbol{u}), \tag{1}$$

where $\boldsymbol{u}$ is the vector of state variables and $f(\boldsymbol{u})$ is the right-hand-side (RHS) function that prescribes species and energy conservation (governing equations). State variables include the mass fractions (Y_j) of each species (j) and (for all reductions performed to date) temperature (T). Thus, the number of species (NS) in the chemical kinetics mechanism establishes the number of ODEs: NS+1. The number of reactions (NR) in the mechanism also impacts the cost because each contributes a source term to each conservation equation.

Because reaction rates (typically) have an exponential dependence with respect to $1/T$, and possibly T^n as well, the RHS function is highly nonlinear, resulting in the ODEs being stiff. Stiffness lacks a rigorous mathematical definition but, in general,

refers to systems in which the integration step (h) of a numerical method is constrained by stability, not by accuracy. That is, h is forced to be far smaller than necessary to satisfy the specified level of accuracy. Stiffness often arises in combustion modeling due to the wide range of time-scales between fast- and slow-reaction chemical species. Stiff systems can be efficiently integrated with implicit integration methods that are stable regardless the size of h allowing the step-size to be adjusted only based on the accuracy requirements. In TMM3, the ODEs of the HR model are integrated in time using version 3.0 of DASPK.[9] The DASPK library employs a variable-order (between first- and fifth-order) backward difference formula with variable h.

DASPK, like most implicit integration methods, requires the computation of the ODE system's Jacobian matrix (J):

$$J = \frac{\partial f(\boldsymbol{u})}{\partial \boldsymbol{u}} \tag{2}$$

In the HR models called by TMM3, J is a dense matrix with $(NS+1)^2$ elements. Analytical expressions for computing its elements may be derived,[10] but doing so would be difficult because there are many different types of reactions (and thus source term types) involved in the mechanisms developed by ARL scientists. For this reason, the following numerical approximation based on finite-differences is used instead:

$$J_{ij} \cong \frac{f_i(u_j+\varepsilon)-f_i(u_j)}{\varepsilon} \tag{3}$$

where for RHS (sub)function i, ε is a value that is small relative to u_j.

Each approximation of J requires NS+2 RHS evaluations. Since the number of source terms that must be computed for each of those evaluations is proportional to NR, which also generally increases in proportion to NS, the cost of the finite-difference approximation of J typically increases quadratically with NS and can be very costly. Fortunately, each column vector can be evaluated independently, and this was the primary source of parallelism implemented in TMM3. In addition, since J does not contribute directly to the updated $\boldsymbol{u}$, an approximation of J is sufficient so long as the system converges efficiently. Exploiting this fact, DASPK will reuse the same J for up to six timesteps to reduce the number of times it has to be recomputed.

A lower-upper (LU) factorization of the iteration matrix (i.e., $[I - \beta hJ]$) is also needed to update the solution vector for each iteration. Since the matrix is dense, the cost is proportional to $(NS+1)^3$ (i.e., cubic), and thus very high for large NS. DASPK originally used subroutines in the LINPACK[11] library to perform the

factorization. Basic Linear Algebra Subprogram (BLAS)[12] functions were also used extensively for vector–vector operations (i.e., Level-1 BLAS). More-efficient factorization algorithms are available in the LAPACK[13] library, which provides the same functionality as LINPACK and, in some instances, node-level parallelism.

2.2 SENKIN

Representing the simplest canonical combustion problem, an HR model derived from SENKIN is the first one employed by an ARL researcher to evaluate a new mechanism. Solutions are analyzed to determine if results for state variables and (possibly) derived properties are consistent with expectations. In addition, the solutions are analyzed to identify individual reactions or groups of reactions that have the most impact on a system's dynamics. That is facilitated by the model's capacity to perform sensitivity analyses.

Sensitivity analyses* involve computing the sensitivity coefficient s_j for a parameter p_j (i.e., $s_j = \frac{\partial \boldsymbol{u}}{\partial p_j}$) via the forward sensitivity equation:

$$\frac{\partial s_j}{\partial t} = \frac{\partial f(\boldsymbol{u})}{\partial \boldsymbol{u}} s_j + \frac{\partial f(\boldsymbol{u})}{\partial p_j} . \tag{4}$$

To exploit the fact that the coefficient of the first term of the RHS is the Jacobian of the ODE system, DASPK implements a staggered corrector method[9] in which the governing ODE system is solved first (i.e., the Newton iteration for a timestep is converged) and then the same *J* is used to update the sensitivity equations, thus avoiding an additional, costly *J* evaluation.

An arbitrary number of sensitivity equations can be solved in SENKIN. However, ARL researchers commonly compute sensitivity coefficients with respect to the pre-exponential factor of the modified Arrhenius function employed to compute reaction-rate coefficients. For such computations, there are NR p_j terms resulting in [NR*(NS+1)] sensitivity equations that must be solved. Although *J* (and its LU factorization) can be reused for each sensitivity equation, the triangular forward and backward solves needed for the LU decomposition for each sensitivity equation are very costly. Furthermore, each s_j requires the computation of $\frac{\partial f(u)}{\partial p_j}$ terms, which like the elements of *J*, are calculated via finite differences. Therefore, even though the sensitivity equations are linear, and thus generally easier to solve than the governing equations, when NR and NS are large, their cost can be considerable, indeed often being the dominant cost of a simulation.

* For an introduction to forward sensitivity equation solution methods, see the CVODES[14] documentation.

2.3 OPPDIF

OPPDIF[15] attempts to find the steady-state solution for opposed-flow diffusion flame problems. It supposes that two circular, gaseous jet streams with opposed velocities impinge and form a stagnation plane with a stationary diffusion flame. Presuming radial symmetry, such problems reduce to one (spatial) dimension oriented in the axial direction. The governing equations include NS+1 equations to enforce species and energy conservation and three equations to enforce mass and momentum conservation (i.e., NS+4 total governing equations). Boundary conditions are specified at both ends of the domain, establishing a two-point boundary value problem (BVP). OPPDIF is often used to analyze the impact of strain-rates on a flame's properties, including extinction.

Accounting for the molecular transport of species, momentum and energy in a multicomponent gaseous mixture requires the computation of diffusion coefficients, viscosities, thermal conductivities, and thermal diffusion coefficients. For OPPDIF simulations, these parameters are computed via calls to a library called TRANLIB, which is a companion of CHEMKIN. It can implement either mixture-averaged or full multicomponent approaches to compute them. The former involves determining a gas's properties from properties for pure species via mixture averaging rules. The latter is more accurate, but is much more computationally expensive. The evaluation of ordinary multicomponent diffusion coefficients requires the inversion of an NS × NS matrix, and the evaluation of thermal conductivity and thermal diffusion coefficients requires solving a 3*NS × 3*NS system of algebraic equations.

The steady-state equations are discretized on a nonuniform 1-D grid. Convective terms are computed based on first-order upwind differences, and diffusive terms are computed based on second-order central differences. The mesh is automatically refined to resolve steep gradients.

The BVP is solved via the TWOPNT math library. It uses Newton's method to search for solutions. If a search fails, TWOPNT will attempt to generate a better initial guess for it by marching the system forward in pseudo-time using an implicit method. Both the pseudo-time and Newton algorithms require computing a Jacobian for the problem being solved. Since there are (NS+4) unknown state variables at each mesh point, but each mesh point is dependent only upon its immediate neighbors, the Jacobian is block-tridiagonal with square submatrix blocks each with (NS+4) × (NS+4) elements. As in TMM3 and SENKIN, finite differences are employed to evaluate elements of the matrix.

TWOPNT stores the Jacobian in a banded format with a bandwidth of [2*(NS+4)-1] above and below the diagonal. There are [(NS+4)*NP] unknown state variables in OPPDIF problems, where NP is the number of mesh points. The banded format requires approximately [4*NP(NS+4)^2] terms. (Note that this storage allocation is larger than needed for the block-tridiagonal matrix; the additional terms are included to support partial-row pivoting in the LU factorization.)

The overall cost of OPPDIF is difficult to predict due to the manner in which TWOPNT searches for a solution. In some cases, the search will converge rapidly on a coarse mesh and all subsequent refined meshes. However, in other instances, thousands of pseudo-timesteps and many Newton iterations may be required to find the solution on the final mesh. Additionally, as will be discussed later, the cost grows exponentially with the NP, and most of the computational time is spent finding solutions on refined meshes. For this reason, I focused on accelerating the per-iteration cost when NP is large.

As in TMM3 and SENKIN, the major cost of a pseudo-timestep or a Newton iteration in OPPDIF involves the assembly and factorization of the Jacobian. Indeed, due to the spatial dimension (i.e., the mesh points), their size can be of O(10–100) times larger than those encountered in TMM3 or SENKIN, making the cost of their assembly and factorization extremely high. Furthermore, the LU factorization of the Jacobian can be exceedingly[†] costly. As in TMM3 and SENKIN, the Jacobian can be assembled in parallel when using the finite-difference approximation. Note that while there are [NP*(NS+4)] columns, only [3*(NS+4)+1] RHS evaluations are needed to form the Jacobian due to its block-tridiagonal structure.[‡] In addition, though their scalability is generally less efficient than for a dense matrix, parallel LU factorization methods are available for banded matrices.

Another significant cost on a per-iteration basis is the computation of the RHS function. This involves 1) evaluating the convective and diffusive fluxes between all mesh points and 2) evaluating the net reaction rates at each point. NP evaluations of the reaction rates (i.e., one per point) can become quite costly, as can the evaluation of the diffusion coefficients. In the latter, the evaluation of the NS^2 terms of the binary diffusion coefficient matrix must be factorized and solved with O(NS^3) cost if the multicomponent formulation approach is employed. An O(NS^2)

[†] Iterative Krylov methods for solving the linear systems are supported in OPPDIF. These may be useful for larger mechanisms when the cost of direct factorization becomes prohibitive in terms of memory or time.

[‡] Since a mesh point is only dependent upon its immediate (±1) neighbors, every third block-state vector can be perturbed, thereby evaluating multiple, linearly independent columns vectors with each RHS evaluation.

cost is incurred if the mixture-averaged approach is employed. The fluxes, diffusion, and reaction-rate computations can be evaluated concurrently, providing some parallelism in the computation of the RHS function.

2.4 PREMIX

PREMIX[16] is designed to set up and find steady-state solutions for two premixed flame types: 1) laminar, freely propagating flames and 2) burner-stabilized flames. Measured (or modeled) heat loss to the burner device and measured temperature problems can be specified in the burner-stabilized model. Laminar flame speed is often the target output of the freely propagating model.

PREMIX is very similar to OPPDIF in that a 1-D flame is simulated on a finite-difference mesh and solved with TWOPNT. Its solution search algorithm and discretization method are the same as OPPDIF's. Computational aspects such as the Jacobian's structure and RHS evaluation details are also the same. Since these account for the bulk of the computational cost, the OPPDIF analysis applied to PREMIX. (The number of governing equations in PREMIX is NS+2, two less than OPPDIF. For large mechanisms, the difference in computational cost is negligible.) Therefore, the optimization methods implemented were identical.

3. Optimization Methods

As already discussed, all four programs and the supporting libraries were serial codes, and thus could only use a single central processing unit (CPU) core on DSRC computing platforms. They were originally developed in the 1980s and 1990s at a time when the dominant high-performance computing (HPC) paradigm was vector parallelism on a single CPU. This approach exploits data parallelism in which the same operation (e.g., addition) can be applied concurrently to separate data (e.g., increment each element of an array concurrently). This approach is still supported in modern HPC systems via single-instruction, multiple data (SIMD) parallelism within each CPU core. However, all modern HPC systems are multicore with $O(10)$ CPU cores per node. As such, to fully harness the computing power available on DSRC platforms, applications must exploit multicore parallelism.

The two dominant parallel programming paradigms for modern multicore HPC systems are shared memory and distributed memory. Both are suitable for programs in which domains can be partitioned into sections and processed concurrently.

Shared-memory (shmem) parallelism (generally) uses a multithreaded programming method so that all threads can view the same memory (along with their own private memory). Multiple threads of execution are spawned from the

parent process. In the HPC context, one (or several) threads are assigned to each CPU core. This approach allows an application to use all the cores on a single compute node but *does not* support parallelism across multiple nodes. The threads in a shmem application communicate, implicitly or explicitly, by updating shared memory locations and by synchronizing. Since threads can access the same memory, care must be taken to ensure only one thread updates a shared variable at a time. Otherwise, the value of the shared variable becomes undetermined. This is referred to as a race condition.

Distributed-memory parallelism uses multiple processes, each with its own (unshared) memory, that communicate explicitly. The most common communication method is via the message passing interface (MPI) standard. There is no limit to the number of processes that can be run in parallel in the distributed-memory paradigm; all cores across hundreds or thousands of nodes can be used concurrently. However, the development of such applications is generally much more difficult than shmem applications. In particular, the developer must explicitly partition the problem domain and assign each MPI process a portion of the partitioned workload. Furthermore, all communication is explicit: data must either be sent and received between individual processes (i.e., point-to-point messages) or all processes must participate in collective communication operations (e.g., all-to-all, one-to-all, or all-to-one).

Shmem, while limited to node-level parallelism, was selected for this refactoring effort because (generally) it is faster to incorporate into existing serial applications and supports incremental optimization. I employed the OpenMP[17] multithreading language to implement it.

OpenMP is commonly used to implement shmem parallelism in computational science applications. It supports the fork-join threading model[§] in which the application runs serially until a parallel region is encountered. In the fork-join model, a fixed number of threads are created (i.e., forked) from the master (serial) thread upon entry into a parallel region. The workload inside the region is partitioned via a work-sharing partitioning algorithm, and each thread executes its assigned portion of the overall workload. At the end of the parallel region, all threads synchronize and are joined back into the master thread. At this point the code returns to serial mode until the next parallel region is encountered.

A common example of the OpenMP work-sharing approach is partitioning iterations of a loop and executing the partitions in parallel. Briefly, the *k* loop

[§] Fork-join parallelism with work-sharing is often sufficient for applications dominated by large loop structures. OpenMP also supports task-based parallelism for unstructured parallelism.

iterations are split into p contiguous, non-overlapping chunks of size (k/p). (In OpenMP, k does not need to be an integer multiple of p. At most, q threads will execute an extra loop iteration where q = *[k modulo p]*.) Commonly, p equals the number of threads. The partitioned loop iterations are then executed concurrently and, in the ideal case, will complete p times faster than the serial version. In practice, this perfect scaling is rarely achieved due to the overhead associated with forking the threads and synchronization.

Any number of parallel regions may be set up with OpenMP. This allows for incremental parallelization of suitable regions of the code. Generally, incremental refactoring follows this workflow:

1) Select the most-expensive loop structure identified through loop-level profiling.
2) Refactor the loop operations to avoid race conditions and/or inefficient data sharing.
3) Add OpenMP parallel loop declarations with suitable directives regarding privatized variables.
4) Verify the results produced by the parallel region match those produced by the serial version.
5) Profile the refactored code to assess the parallel efficiency.
6) If efficiency is unacceptable, repeat steps 1–4 to improve the efficiency.

In the four models considered, the high cost of computing J was common to all of them. In each case, this operation was refactored following the same strategy, though the specific implementations were unique. It is computed in the following manner:

1) Evaluate $f(\boldsymbol{u})$
2) Enter an OpenMP parallel region
3) For all columns $j = 1, N$
    a. perturb the state vector: $\boldsymbol{u}^+ = \boldsymbol{u} + \delta_j$
    b. evaluate $f(\boldsymbol{u}^+)$
    c. update the column vector: $J_j = [f(\boldsymbol{u}^+) - f(\boldsymbol{u})] \,/\, \delta_j$
4) Close the OpenMP parallel region

The N iterations over the columns of J are independent and could be evaluated in parallel using OpenMP parallel loop directives. Each thread needed private arrays for the perturbed state vector and resulting RHS vector. Additionally, each thread needed private storage internally for the RHS function. OpenMP will automatically allocate fixed-length private arrays; however, some larger, dynamic arrays with variable length had to be allocated and deallocated manually inside the parallel region.

The RHS functions of all four applications call several CHEMKIN functions (e.g., CKWYP to compute net species production rates). OPPDIF and PREMIX also call TRANLIB functions (e.g., MCMDIF to compute multicomponent diffusion coefficients). These libraries have internal data structures that combine read-only data (e.g., Arrhenius rate coefficients and species molecular weights) with intermediate scratch arrays employed for computing complex functions such as reaction-rate coefficients. Each OpenMP thread needed private copies of the CHEMKIN scratch arrays. Ideally, all threads would share a single copy of the read-only data. However, CHEMKIN and TRANLIB store all internal data (i.e., read-only and scratch) in contiguous integer and floating-point arrays that are difficult to separate. Therefore, each thread was programmed to create a private (redundant) copy of the read-only data. Although this is a one-time allocation—the data are reused for the lifetime of the simulation—it wastes memory and may negatively impact cache usage.

Because it does not consider the internal features of the RHS function, the parallelization method implemented to compute the Jacobian is generic. It can be applied to most ODE systems solved by DASPK. An additional optimization was implemented that is specific to the HR model and TMM3. The CHEMKIN function CKWYP computes the net species-production rates given pressure, temperature, and species mass fractions. Internally, CKWYP computes the molar concentration via CKYTCP and then calls CKRATT and CKRATX to compute, respectively, the temperature- and concentration-dependent reaction-rate terms. In the finite-difference approximation employed to compute the Jacobian, the temperature is only perturbed for one vector column. That is, the temperature is constant for NS+1 of the NS+2 RHS evaluations. Based on this observation, it was possible to reduce the cost of computing the Jacobian by directly calling CKRATT and CKRATX instead of CKWYP. Additional logic was added to the Jacobian assembly function to precompute and share the temperature-dependent rate terms across all threads. Note that this approach also applies to the serial implementation and will reduce its cost and execution time.

Conceptually, this method would also work in SENKIN. However, the cost to compute the Jacobian was less than the cost to compute terms for the sensitivity

equations. Furthermore, the SENKIN RHS function allows perturbations of other model parameters for sensitivity analysis, complicating this method of optimization. Therefore, the cost (in labor) to refactor the Jacobian's evaluation in it was not considered warranted.

Like the evaluation of the Jacobian, the $\frac{\partial f(u)}{\partial p_j}$ terms of the sensitivity equations are evaluated by finite differences. SENKIN supports both first-order forward differences (like those used in evaluating the Jacobian) and second-order central differences. The second-order formulation is used here. It requires two evaluations of the governing RHS function for each sensitivity equation. As already noted, each sensitivity equation is independent, and there may be many times more sensitivity equations than governing equations. For example, there are 1,930 reactions and 456 species in ARL's DMAZ-RFNA mechanism. Therefore, because there are 4.23 times more reactions than species, the RHS function for the sensitivity equations could take approximately 8.5 times longer to compute than the system's Jacobian.

The OpenMP parallel loop implementation for the evaluation of the sensitivity parameter derivatives is very similar to that described above for the evaluation of the Jacobian matrix. That is,

1) Evaluate $f(\boldsymbol{u})$
2) Enter an OpenMP parallel region
3) For each sensitivity equation $j = 1, N_{sens}$
    a. perturb the state vector: $\boldsymbol{u}^+ = \boldsymbol{u} + \delta_j$
    b. evaluate $f(\boldsymbol{u}^+)$
    c. perturb the state vector: $\boldsymbol{u}^- = \boldsymbol{u} - \delta_j$
    d. evaluate $f(\boldsymbol{u}^-)$
    e. update the column vector $\frac{\partial \mathrm{f}(\boldsymbol{u})}{\partial \mathrm{p}}\Big|_j = [f(\boldsymbol{u}^+) - f(\boldsymbol{u}^-)]/2\delta_j$
4) Close the OpenMP parallel region

It required an additional thread-private array for both RHS vectors evaluated with the forward and backward perturbed state vectors.

The last unique optimization implemented in SENKIN was the parallel update of the sensitivity equations. Once the primary state vector is solved at a given timestep, all sensitivity equations are updated using the staggered approach. Each equation requires the linear solution of the factorized iteration matrix with a unique RHS

vector. The LINPACK function DGESL is used to solve the linear system from the previously factored Jacobian matrix via forward and backward triangular matrix solves:

1) Enter an OpenMP parallel region
2) For each sensitivity equation $j = 1, N_{sens}$
    a. solve $\mathrm{J}\Delta = LU\,\Delta = r_j$
        i. solve $\mathrm{L}y = r_j$
        ii. solve $\mathrm{U}\Delta = y$
    b. update the sensitivity equation $s_j = s_j + \Delta$
3) Close the OpenMP parallel region

Here, Δ is the update vector needed to advance the solution one timestep.

In refactoring PREMIX and OPPDIF, the approach employed to parallelize *J*'s computation in SENKIN was followed and will not be repeated here. Two unique optimizations were implemented in both applications: 1) evaluating each mesh point in the RHS function in parallel and 2) replacing LINPACK functions for banded matrix factorization and linear solves with LAPACK functions.

The RHS function in both PREMIX and OPPDIF requires evaluation of convective and diffusive fluxes at the midpoint between the mesh cells, the net species production rates at the cell centers, and evaluation of the energy equation at the cell centers. The fluxes are relatively cheap to evaluate but the cost to compute transport properties (i.e., diffusion, conduction, and viscosity) can be very high. The fluxes and coefficients at the mesh midpoints can be evaluated in parallel, as can the reaction rates at the cell centers. To date, the midpoint fluxes, transport coefficients, and reaction rates have been evaluated in parallel in PREMIX but not in OPPDIF.

Note that when the parallel Jacobian is being constructed, the parallel directives in RHS are automatically disabled by OpenMP. By default, OpenMP does not allow nested parallelism (i.e., a thread in a parallel loop creating more threads in an inner, nested loop). In this case, the RHS function is executed serially by each thread while constructing a column of the Jacobian.

Since the size of the linear system can be *O*(10–100) times larger for OPPDIF and PREMIX than for TMM3 or SENKIN, matrix factorization can be the dominant cost of a simulation. OPPDIF and PREMIX use a banded matrix format with a bandwidth of approximately [2*(NS+2)]. LINPACK functions DGBFA and DGBSL were used to factor and solve the linear systems in the original programs.

They only supported vector parallelism (i.e., vector operations) within the BLAS functions with a maximum length of [2*(NS+2)]. In the classic Crout LU factorization algorithm with scalar elements, only limited vector parallelism is possible.

LAPACK uses a block-matrix formulation for all matrix–matrix operations, including factorization. This provides for a much-higher-level parallelism than LINPACK. LAPACK also makes much more efficient use of the cache hierarchy.[**] There are several highly optimized implementations of the LAPACK library. I used the Intel Math Kernel Library (MKL) for this effort. It provides single- and multicore parallel versions of LAPACK. In particular, the DGBTRF and DGBTRS functions were implemented to factor and solve the band-matrix system.

The TRANLIB library uses LAPACK functions DGETRF and DGETRS to factor and solve the dense (NS^2) diffusion coefficient matrix, D_{jk}, when the multicomponent transport formulation is requested by PREMIX and OPPDIF. When evaluating the RHS function in parallel, each thread must factor and solve its own unique copy of D_{jk} serially. MKL serial versions were used to avoid nested parallelism when the RHS or Jacobian functions are computed via OpenMP parallel loops. Note that the serial MKL implementations of the LAPACK functions are also heavily optimized.

TWOPNT also computes an approximation condition number for the linear system (i.e., $||A|| * ||A^{-1}||$). Used to steer the iterative solver, it was evaluated by the LAPACK function DGBCON.

4. Results

The following are the performance enhancements achieved for the four refactored applications. To demonstrate their portability, testing was done on several different HPCMP DSRC platforms.

4.1 TMM3

At the outset of the RA activity, serialized instances of the TMM3-based reductions were being run concurrently on individual CPU cores. This permitted coarse-grain parallelism, but runtimes for large mechanisms were severely compromising what could be accomplished. A case in point was the reduction of a mechanism developed to model ammonium perchlorate-hydroxyl-terminated polybutadiene

[**] Cache efficiency was a principle design goal of the LAPACK library and an important optimization consideration for modern multicore systems.

(AP-HTPB) combustion.[4] It had 2,634 reactions and 639 species and needed to be reduced for use in a CFD model of AP-HTPB composite propellant deflagration. Validity over a wide range of pressures was desired. Given also that the gas phase would have regions that ran from very fuel rich to very fuel lean, ARL researchers would have liked to have employed as many as six different HR simulations as bases for the screening process. However, due to the lengths of the simulations, if more than two were employed, a (serial) reduction could not complete within the allowable 168-h (7-day) wall-time limit on Excalibur, a Cray XC40 HPC system at the ARL DSRC. Moreover, even with only two employed, reductions of well over half of the orderings that were submitted failed to complete within the allowable time.

Using the refactored version with eight OpenMP threads (i.e., eight cores) per ordering on Excalibur, the overall runtime was reduced by 3.5 times versus the original one core per ordering, and all the reductions that were submitted completed within the 168-h wall-time limit.

The optimal number of threads per reduction can be tuned to maximize the throughput. The only parallelism in TMM3 within each realization is the evaluation of the Jacobian's elements. The matrix has NS+1 columns, and runtimes could, theoretically, be further reduced by using that many threads (cores) in parallel. However, each OpenMP thread should evaluate many columns to overcome the overhead associated with the fork-join model. Amdahl's law dictates that the maximum speedup of a fork-join style of parallel application is

$$\text{Speedup} \leq \left\{ \frac{1}{(1-f_p)} = \frac{1}{f_s} \right\} \tag{5}$$

where f_p and f_s are the percentage of the application executing in parallel and serial, respectively, and $f_s = (1 - f_p)$. For example, if 10% of the application executes serially, the speedup limit is 10× with an infinite number of parallel threads and cores. (Of course, you cannot exceed the number of cores on a single compute node.) Therefore, as TMM3 reduces the mechanism's number of species, the Jacobian's computation constitutes a smaller fraction of the overall cost, increasing relative cost of the serial section and reducing the impact of the parallelization. Since TMM3 can already harness some coarse-grained parallelism, I recommend keeping the number of threads between 4 and 16 per realization.

4.2 SENKIN

For SENKIN, the sensitivity equation RHS function and solution updates were parallelized. Table 1 shows the original (serial) runtimes and the parallel

runtimes (and speed-up) for three representative mechanisms using 16 and 32 cores on a single compute node on Excalibur. Serial times show the rapid cost increase of the simulations with increase in mechanism size. For the small hydrogen–air mechanism, the code actually ran slower with OpenMP. Due to the small amount of parallel work (i.e., Amdahl's law in action), this result is not noteworthy. For the larger mechanisms, the simulations completed much faster, and the speed-up approached 8× on a full compute node.

Table 1 HR model performance on Excalibur: wall-clock times (seconds) for serial and parallel runs on Excalibur

Mechanism	Mechanism size	Serial time	Parallel time (speed-up)	
			16 Cores	32 Cores
H_2-O_2	10/18	2.6	2.7 (0.96)	…
MMH-RFNA	106/680	270	41.4 (6.3)	34.0 (7.9)
DMAZ-RFNA	456/1930	2,670	630 (4.2)	514 (5.2)

4.3 PREMIX

Figure 1 shows the evolution of optimizations implemented in PREMIX when they were employed to simulate a burner-stabilized, premixed flame. The computing platform was Centennial, an SGI ICE XA HPC system at the ARL DSRC. The mechanism that was the basis for the simulation had 514 chemical species and 937 reactions. It was assembled to model the combustion of propellants formulated with nitrocellulose and nitroglycerine.

The species concentrations and temperature versus distance profiles employed as the starting point for the simulation corresponded to a (converged) solution obtained for a similar set of boundary conditions. The program will estimate the profiles based on other input types, and sometimes it is necessary to initialize them with an ad hoc profile (e.g., equilibrium concentration in the product region with a planar *T* profile). However, such approaches can be costly and are prone to failure.

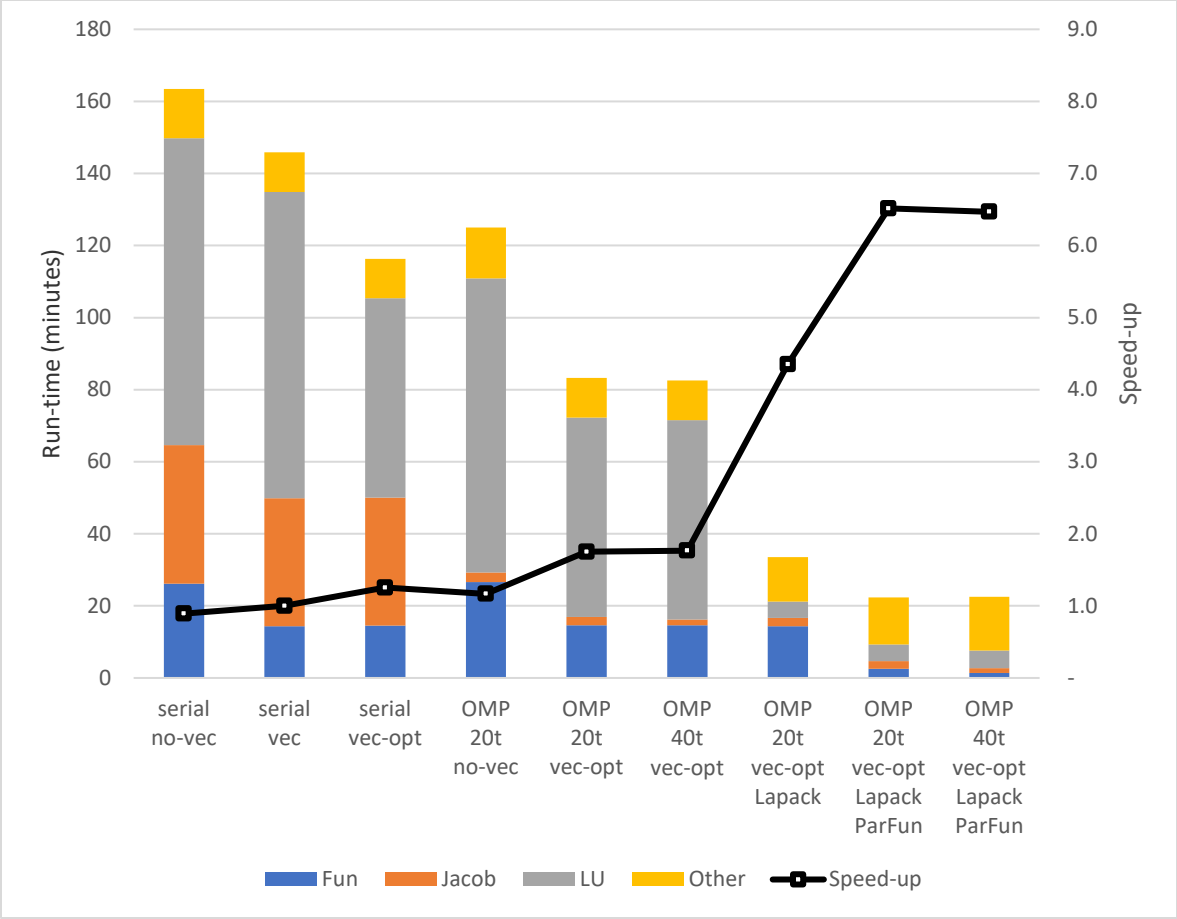

Fig. 1 PREMIX benchmark using serial and OpenMP parallelism on Centennial (ARL DSRC). Nitrocellulose–nitroglycerin mechanism with 514 species and 937 reactions. Runtime categories shown are the RHS function time (Fun), Jacobian matrix assembly (Jacob), Banded matrix factorization (LU), and all other costs summed (Other). Cases include: single (serial) and multicore (OMP); vectorization disabled (no-vec), standard compiler vectorization (vec), and vectorization with enhanced (BLAS) kernels (vec-opt); parallel RHS function (ParFun); and replacing Linpack functions with Lapack (Lapack). Speed-up is relative to single-core execution with standard compiler vectorization enabled (serial-vec).

In this specific case, the starting mesh had 228 points, and no refinement was needed for the new solution. The total number of unknowns was 117,648 (i.e., [NP*(NS+2)]). Transport properties were computed on the basis of the mixture-averaging rules implemented via TRANLIB. Again, this approach costs less than TRANLIB's multicomponent formulation.

As noted earlier, the original version of the PREMIX code used the LINPACK linear algebra library with level-1 BLAS functions. These functions rely only upon vector data parallelism (i.e., SIMD parallelism) within a single CPU core. To highlight the impact of vector parallel processing, automatic compiler vectorization

was explicitly disabled[††] and compared with normal compiler optimization. The case without automatic vectorization (serial/no-vec) ran 11% slower than the baseline case (serial/vec) case. The cost breakdown shown in Fig. 1 indicates that the RHS function (mostly the CHEMKIN and TRANLIB functions) significantly benefited from vectorization. The BLAS functions were then refactored to improve automatic vectorization (serial/vec-opt). This reduced the LU factorization time by 35%. Note these studies were performed with only one of the 40 available CPU cores on a Centennial node.

When performed with the serial implementation of the program, the computation of the Jacobian (with all vectorization enabled) accounted for 31% of the total runtime. When performed with OpenMP and 20 CPU cores, the runtime needed to compute the Jacobian was reduced by 14.5 times, becoming only 2.1% of the total runtime. The overall runtime for the simulation was reduced by 1.75 times.

Performing the simulation with all 40 cores (i.e., 40 OpenMP threads) on a node did not produce an appreciable (additional) reduction in the runtime. This is not uncommon with OpenMP. A Centennial node comprises two CPU sockets, each with 20 cores. Each socket has its own dynamic-random-access-memory main memory and all 20 cores share the L3 cache. Threads on socket A can access the data on socket B, but there is a latency and bandwidth penalty (i.e., nonuniform memory access [NUMA]). As a result, the runtime for the evaluation of the Jacobian's column vectors, which was the only parallelized process, was only 31% less than that achieved with 20 cores. Moreover, since the Jacobian's evaluation accounted for such a small portion of the simulation's overall runtime, the reduction in the simulation's overall runtime was virtually undetectable.

The costliest components of the PREMIX simulation are the banded matrix LU decomposition and estimation of the matrix condition number. With the parallel Jacobian assembly operation, the LU factorization accounted for 67% of the net runtime. LAPACK functions were substituted for LINPACK versions and linked using the optimized Intel MKL. Using 20 cores on a single CPU socket, the factorization was accelerated by 18.9 times compared with the baseline LINPACK implementation, and the total runtime was reduced by 4.4 times. Note that the category Other in Fig. 1 includes all time not spent in the other three categories. It was dominated by the linear system solve operation (i.e., solving the linear systems with the previously factored Jacobian matrix). A LAPACK version was used for this as well, but it did not provide any significant speed-up because this operation was not thread efficient.

[††] Modern optimizing compilers will apply SIMD vector instructions when possible by default.

The last optimization involved the evaluation of the transport coefficients, convective fluxes, diffusive fluxes at the mesh midpoints, and the reaction rates at the mesh points in parallel (i.e., evaluate the RHS function in parallel). The transport property evaluations, even though based on mixture-averaged terms, were the dominant cost in computing the RHS function.

With 20 cores on a single CPU socket, the RHS function evaluations were accelerated by 5.6 times and the total runtime was 6.5 times less than the baseline serial code's. With all 40 cores, the total runtime was greater even though RHS function evaluation, Jacobian matrix evaluation, and LU factorization were 47%, 37%, and 15.5% faster, respectively. In this case, the time to estimate the condition number increased 19%, and the time spent solving the linear systems, which accounted for 59% of the net runtime, increased 15%. Parallel triangular matrix solves, which were a significant cost of the condition number estimation, required repeated synchronization and significant data sharing and was impeded by NUMA effects. These operations also had lower computational intensities than factorization (i.e., quadratic versus cubic theoretical cost-scaling), which explains why they did not scale as well in parallel. By contrast, the RHS and Jacobian function evaluations required no thread synchronization, and each thread operated on its own data exclusively, making it more cache-efficient.

It is possible to separately control the number of threads used by MKL and the main OpenMP parallel code in PREMIX. By setting the number of threads used by MKL to 20, and presumably forcing those 20 threads to use one CPU socket, while using all 40 threads (cores) for the remaining OpenMP regions, the total speedup increased to 7.3 times (not shown). In this case the performance of the triangular solves with LAPACK was not hindered by the NUMA overhead, while the RHS and Jacobian functions could still efficiently harness all 40 cores on the compute node.

4.4 OPPDIF

The parallel computation of the Jacobian's elements implemented in the OPPDIF application followed the same strategy employed for PREMIX. The problem employed to evaluate performance improvements was an opposed-flow diffusion flame simulation with HMX as the oxidizer and a combination of HMX (93%) and R45M (7%) in the fuel stream. Although TWOPNT failed to produce a solution that met specified converge criteria for this problem, using 22 threads on one CPU socket on the Onyx HPC system (Cray XC50) at the US Army Engineer Research and Development Center DSRC, the parallel Jacobian assembly method *did* reduce

the total runtime for the calculations that were performed from 4.5 to 3.2 h. The speed-up as a function of the number of cores used is shown in Fig. 2.

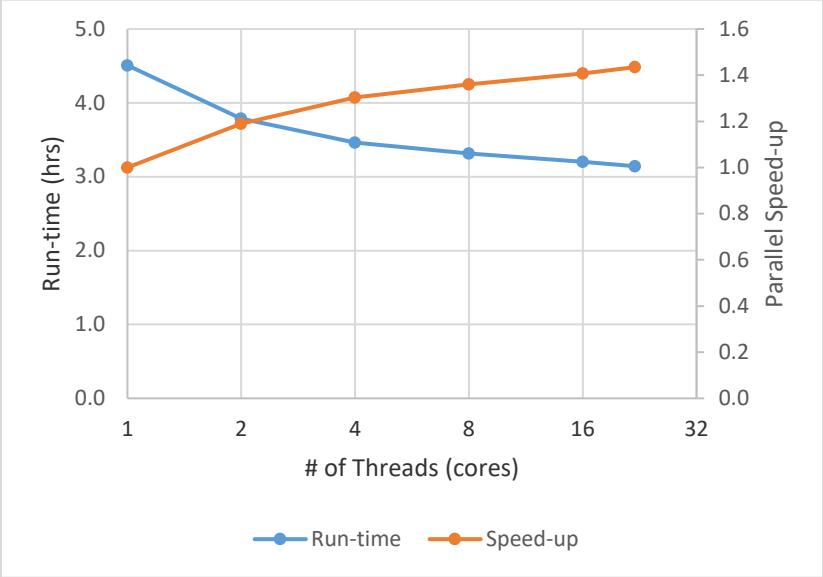

Fig. 2 Scaling of OPPDIF on Onyx with OpenMP parallel Jacobian matrix assembly, using one CPU socket with up to 22 cores and one thread per core

In addition, although the Jacobian's construction was costly, it was not the dominant expense of the simulation. As found in refactoring PREMIX, the LU factorization of the banded matrix was more expensive for large mechanisms and large numbers of grid points. Therefore, LAPACK routines were implemented as before to reduce the factorization cost. Direct comparison of the runtimes may be somewhat misleading because neither case converged to a solution, and they failed in different ways. As noted, it is common for solutions to diverge when starting from an ad hoc initial profile. The difference was likely due to (small) differences in the condition number approximation used to steer the Newton iteration method being amplified by the nonlinear system. And because the solutions diverged in different manners, the number of nonlinear iterations and the refined mesh sizes were different. Nonetheless, the course of the simulations were similar enough that I believe the comparison reflects the performance enhancements that can be expected with the parallelized code.

Another simulation employed to evaluate the impact that various parallel implementations would have on OPPDIF runtimes involved the USC 111 species-784 reaction (USC) mechanism. This mechanism was designed to model H_2/CO/C1-C4 hydrocarbon combustion.[‡‡] Transport properties were computed on

[‡‡] University of Southern California. http://ignis.usc.edu/Mechanisms/USC-Mech%20II/USC_Mech%20II.htm.

the basis of TRANLIB's mixture-averaging rules. The fuel stream contained 9.5% CH_4 by volume, and the oxidizer stream was air at atmospheric pressure and temperature. The starting temperature and species concentrations versus distance corresponded to a converged solution on a grid with 83 mesh points. With more-stringent mesh-refinement criteria specified, a new solution was produced on a grid with 156 points.

The runtimes and speed-up compared with the serial, baseline code with (SeqMKL) and without LAPACK functions, and the parallel Jacobian evaluation with (ParMKL) and without LAPACK functions, are shown in Fig. 3.

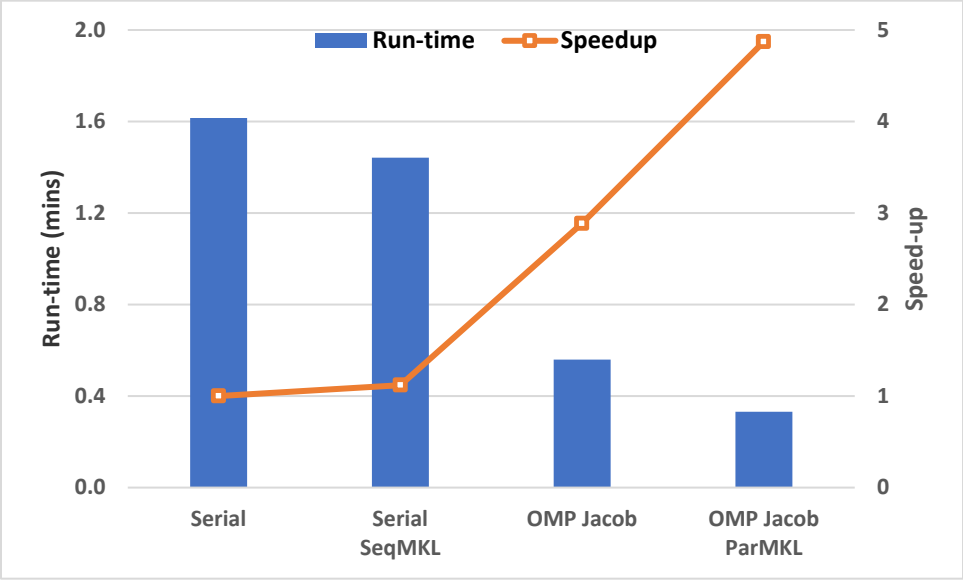

Fig. 3 Runtime (minutes) and speed-up of OPPDIF using USC mechanism on Centennial using parallel Jacobian evaluation and sequential or parallel matrix factorization with MKL

Serially, the Jacobian's assembly and factorization combined to account for 88% of the runtime. Using LAPACK's DGBTRF with MKL, the factorization time dropped 12% and the total runtime dropped the same amount. Evaluating the Jacobian in parallel alone (i.e., without LAPACK) reduced its runtime by 75% and reduced the overall runtime by 2.9 times. Combining both, the runtime was reduced by 4.9 times.

The RHS function itself has not (to date) been refactored as done for PREMIX. Doing so should further reduce runtimes because the RHS function now accounts for 51% of the total runtime for this simulation when using the parallel Jacobian assembly and factorization methods. Even though the RHS is running serially, the MKL LAPACK functions for dense matrices can accelerate the application when the multicomponent formulation for computing transport properties is used. To investigate this matter, the solution produced on the basis of mixture-averaged

transport properties was employed as the starting point of a simulation based on TRANSLIB's multicomponent formulation. Performed with MKL within the serial code, the runtime was reduced by 2.4 times. The speed-up was directly due to the improved factorization performance obtained with the optimized LAPACK libraries.

5. Summary and Conclusions

This report details the motivation and methods for refactoring a suite of CHEMKIN-based programs for multicore parallelism. OpenMP multithreading was used to implement shared-memory (loop) parallelism in four applications: TMM3, SENKIN, OPPDIF, and PREMIX. The net speed-up factors, which varied between the applications, were significant.

Parallelization of finite-difference-based Jacobian evaluations was a common strategy, and it was implemented in each application. However, as mechanisms get larger, it may be useful to investigate the efficiency of analytical Jacobian evaluations for TMM3 and SENKIN.

For PREMIX and OPPDIF, which both involve one spatial dimension, it was found that runtimes were driven by band-matrix LU factorizations of Jacobians. Replacing legacy LINPACK factorization methods with LAPACK functions and linking them with the highly optimized MKL functions significantly improved their performances. Note these changes required only minor changes to the application code.

As mechanism sizes increase, it might be useful to consider alternative solution strategies for the matrix systems. Kyrlov iterative methods (e.g., GMRES) might be more efficient at solving the linear systems for large mechanisms if the factorization of dense matrices becomes problematic.

6. References

1. Chen CC, McQuaid MJ. A thermochemical kinetic-based study of ignition delays for 2-azidoethanamine-red fuming nitric acid systems: 2-azido-N-methylethanamine (MMAZ) vs. 2-azido-N,N-dimethylethanamine. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2014 Feb. Report No.: ARL-TR-6787.

2. Chen CC, McQuaid MJ. Predictions for the deflagration of notional extended solids of carbon monoxide with $(C_4O_4H_2)_n$ stoichiometry. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2017 Mar. Report No.: ARL-TR-7984.

3. McQuaid MJ, Chen CC, Veals J. Simulating accelerating rate calorimetry experiments with a model having a detailed, gas-phase, finite-rate chemical kinetics mechanism: results for RDX. Proceedings of the 40th JANNAF PEDCS Meeting; 2017 May 22–25; Kansas City, MO.

4. Chen CC, McQuaid MJ. A skeletal, gas-phase, finite-rate, chemical kinetics mechanism for modeling the deflagration of ammonium perchlorate-hydroxyl-terminated polybutadiene composite propellants. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2016 May. Report No.: ARL-TR-7655.

5. Kotlar AJ. A general approach for the reduction of chemical reaction mechanisms I: methodology and application to MMH-RFNA. Proceedings of the 5th JANNAF Liquid Propulsion Subcommittee Meeting; 2010 May 3–7; Colorado Springs, CO.

6. McQuaid MJ The trial mechanism method for chemical kinetics mechanism reduction: an approach to developing mechanisms with wider ranges of applicability. Proceedings of the 7th JANNAF Liquid Propulsion Subcommittee Meeting; 2013 April 29–May 2; Colorado Springs, CO.

7. Reaction Design. CHEMKIN. [accessed 2019 Dec 9]. http://www.reactiondesign.com/products/chemkin.

8. Miller MS, Anderson WR. Burning-rate predictor for multi-ingredient propellants: nitrate-ester propellants. J Prop and Pow. 2004;20(3):440–454.

9. Li S, Petzold L. Design of new DASPK for sensitivity analysis. University of California at Santa Barbara Technical Report; 1999 [accessed 2019 Dec 9]. https://dl.acm.org/citation.cfm?id=902467.

10. Safta C, Najm H, Knio O. TChem – a software toolkit for the analysis of complex kinetics models. Albuquerque (NM): Sandia National Laboratories; 2011 May. Report No.: SAND2001-3282.

11. LINPACK [accessed 2019 Dec 9]. http://www.netlib.org/linpack/.

12. BLAS [accessed 2019 Dec 9]. http://www.netlib.org/blas/.

13. LAPACK [accessed 2019 Dec 9]. http://www.netlib.org/lapack/.

14. Hindmarsh AC, Serban R. User documentation for CVODES v4.1.0 (SUNDIALS v4.1.0). Livermore (CA): Lawrence Livermore National Laboratory; 2019 Feb. 12. Report No.: UCRL-SM-208111.

15. Kutz AE, Kee RJ, Grcar JF, Rupley FM. OPPDIF: A FORTRAN program for computing opposed-flow diffusion flames. Albuquerque (NM): Sandia National Laboratories; 1997. Report No.: SAND96-8243.

16. Kee RJ, Grcar JF, Smooke MD, Miller, JA. A Fortran program for modeling steady laminar one-dimensional premixed flames. Albuquerque (NM): Sandia National Laboratories; 1985. Report NO.: SAND85-8240.

17. OpenMP API Specification v5.0 [accessed 2019 Aug 14]. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

List of Symbols, Abbreviations, and Acronyms

1-D	1-dimensional
AP-HTPB	ammonium perchlorate-hydroxyl-terminated polybutadiene
ARL	Army Research Laboratory
BLAS	Basic Linear Algebra Subprogram
BVP	boundary value problem
CCDC	US Army Combat Capabilities Development Command
CFD	computational fluid dynamics
CPU	central processing unit
DOD	Department of Defense
DSRC	DOD Supercomputing Resource Center
EM	energetic material
Fun	function
HPC	high-performance computing
HPCMP	High Performance Computing Modernization Program
HR	homogeneous reactor
LU	lower-upper
MPI	message passing interface
MKL	Math Kernel Library
no-vec	vectorization disabled
NP	number of mesh points
NR	number of reactions
NS	number of species
NUMA	nonuniform memory access
ODE	ordinary differential equation
OMP	multicore
ParFun	parallel RHS fuction

PETTT	User Productivity Enhancement, Technology Transfer, and Training
RA	reactionary assistance
RHS	right-hand-side
serial-vec	single-core execution with standard compiler vectorization enabled
shmem	shared memory
SIMD	single-instruction, multiple data
vec	standard compiler vectorization
vec-opt	vectorization with enhanced BLAS kernels

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	CCDC ARL FCDD RLD CL TECH LIB
2 (PDF)	NAWCWD-CHINA LAKE E WASHBURN B BOJKO
2 (PDF)	NRL B FISHER R JOHNSON
1 (PDF)	COMPUTATIONAL SCIENCE C STONE
8 (PDF)	CCDC ARL FCDD RLC S M LASINSKI `FCDD RLW L T SHEPPARD FCDD RLW LB E BYRD J BRENNAN FCDD RLW LD M MCQUAID C-C CHEN J VEALS M NUSCA