Tenth International Conference on Computational Fluid Dynamics (ICCFD10), Barcelona, Spain, July 9-13, 2018

ICCFD10-079

Employing Multiple Levels of Parallelism for CFD at Large Scales on Next Generation High-Performance Computing Platforms

M. Howard*, T. Fisher*, M. Hoemmen*, D. Dinzl*,
J. Overfelt*, A. Bradley*, K. Kim* and S. Rajamanickam*
Corresponding author: mhoward@sandia.gov

* Sandia National Laboratories, Albuquerque, NM, USA.

1 Introduction

Computational fluid dynamics has traditionally been a significant benefactor of advances in high-performance computing (HPC). The computational cost of simulating the dynamics of a fluid in motion is significant and capturing the true physics of nearly any fluid flow for industrial applications requires simulation on the most advanced supercomputers available.

For at least the past 20 years, there have been minimal changes in processor architectures between generations of supercomputers. For our purposes, we define a generation of a supercomputer to be 3 to 4 years. The status quo has been that each advancing generation brings about a faster processor, as measured by either higher clock frequencies or more cores per processor, but no fundamental changes in the processor architecture or the programming models needed to effectively employ the computing devices.

This status quo has gradually started to change as the major computing vendors have introduced new processor technologies targeted for HPC applications. To illustrate this change it is instructive to examine the processing technologies of the Advanced Technology System (ATS) supercomputers that have been acquired by the U.S. National Nuclear Security Administration (NNSA) in recent years. The first of these machines, ATS-1 (named Trinity), has a traditional Xeon (Haswell processors) partition and another partition based on Xeon Phi processors (codenamed Knights Landing or KNL) and was put into production in 2017. The second of these machines, ATS-2 (named Sierra), is slated to be approximately 125 petaflop/s and is scheduled for production use in 2019. Sierra is truly a heterogenous machine, utilizing IBM's Power9 processors and Nvidia Corporation's Volta GPUs, with a significant fraction of the total flop/s coming from the GPUs.

What is unique and disruptive about devices such as KNLs or GPUs is the trend to add more parallelism to each device either through threads and/or vector units. The Xeon Phi KNL processors in Trinity have 68 processing cores, each with 4 hardware threads and multiple 512-bit vector units, and require multiple levels of parallelism (vector operations as well as thread parallelism) to utilize as much of the approximately 3 teraflop/s theoretical peak available per processor. Furthermore, KNL processors have two memory spaces, an on-package high-bandwidth memory (HBM) in addition to traditional DDR memory. The high-bandwidth memory provides approximately 480 GB/s bandwidth while DDR memory provides approximately 90 GB/s bandwidth. The Volta GPUs in Sierra have a theoretical peak of approximately 7 teraflop/s double-precision and have 900 GB/s memory bandwidth. When combined with the Power9 CPUs, a compute node of Sierra has multiple execution spaces (CPUs and GPUs) and multiple memory spaces (host CPUs and GPUs).

Further complicating the architectural changes are the many programming models that have arisen. It is fair to say that the Message Passing Interface (MPI) has been the de facto standard for inter-node parallelism for almost 30 years. However, MPI itself does little to address intra-node parallelism, that is the type of parallelism needed to effectively use Xeon Phi or GPU architectures by employing thread or vector parallelism. To address intra-node (or on-node) parallelism, OpenMP, CUDA, and OpenACC have

evolved. OpenMP has traditionally been focused on thread parallelism for x86 architectures, but with OpenMP 4.0 and newer, offloading to GPUs is now possible. CUDA is a proprietary programming model developed by Nvidia, specifically for Nvidia GPUs. OpenACC is a model that can span CPU and GPU architectures, and is perhaps more general than OpenMP and CUDA, however, with OpenMP's latest GPU offloading support, OpenMP and OpenACC are now similar in nature. The diversity and complexity in the processing architectures and programming model landscape makes writing a CFD code that can effectively utilize multiple HPC platforms a very challenging task.

The purpose of this article is to introduce the approach being taken at Sandia National Laboratories to utilize multiple levels of parallelism in a compressible CFD code for the next-generation HPC platforms being acquired by the NNSA. Additionally, we discuss the problem of *performance portability*. Performance portability is the notion of writing an application code such that it may be "ported" from architecture to architecture without significant application-side, architecture-specific code or whole-sale rewriting using a different programming model while still maintaining acceptable performance across all architectures. Many efforts have focused on GPU-specific CFD codes [1, 2, 3, 4], and several on Intel's Xeon Phi architecture [5, 6]. Far less work has been done on performance-portable implementations across architectures, but there are some notable exceptions [7, 8]. We discuss the salient programming design elements of our code to achieve performance portability and will present performance across standard CPU-based HPC platforms and the ATS platforms for a large-scale application of interest.

2 Sandia Parallel Aerodynamics and Reentry Code (SPARC)

The SPARC (Sandia Parallel Aerodynamics and Reentry Code) code is the next-generation transonic and hypersonic CFD code being developed at Sandia to support the aerodynamic, aerothermal and aerostructural simulation needs for the lab for at least the next decade and likely much longer. We will briefly describe a primary set of the equations SPARC solves, that is the governing equations for compressible Navier-Stokes for a perfect gas and for a gas in thermochemical non-equilibrium. SPARC also can solve the Reynolds-Average Navier-Stokes (RANS) equations for both perfect and reacting gases; solving the RANS equations was not a part of the performance analysis studies conducted here so we have omitted the commentary on these equations. The primary discretization is a cell-centered finite volume method; the code is written to support multiple discretization types and we are developing and researching high-order finite difference and discontinuous Galerkin discretizations as well. The bulk of the discussion will center on the design features of SPARC that we used to achieve high performance across multiple architectures.

2.1 Governing Equations for a Perfect Gas

The conservation of mass, momentum and energy for a perfect gas (i.e., calorically perfect) can be expressed as

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{F}_i(\boldsymbol{U})}{\partial x_i} - \frac{\partial \boldsymbol{G}_i(\boldsymbol{U})}{\partial x_i} - \boldsymbol{S}(\boldsymbol{U}) = \boldsymbol{0}. \tag{1}$$

The assumptions of a calorically perfect gas are summarized in the equation of state paragraph below.

The conservative variable (or state variable) vector $\boldsymbol{U}$ in equation 1 is written in vector form as

$$\boldsymbol{U} = \begin{pmatrix} \rho \\ \rho v_j \\ \rho E \end{pmatrix} \tag{2}$$

where ρ is density of the fluid, ρv_j is the fluid density times the fluid velocity v_j, ρE is the fluid density times the total energy per unit mass E. The total energy per unit mass is the sum of the fluid's internal energy e and kinetic energy and can be written as

$$E = e + \frac{1}{2}(v_j v_j). \tag{3}$$

The inviscid flux vector $\boldsymbol{F}_i$ is defined as

$$\boldsymbol{F}_i(\boldsymbol{U}) = \begin{pmatrix} \rho v_i \\ \rho v_i v_j + P\delta_{ij} \\ \rho E v_i + P v_i \end{pmatrix} \tag{4}$$

where P is the pressure of the fluid. The viscous flux vector $\boldsymbol{G}_i$ is written by

$$\boldsymbol{G}_i(\boldsymbol{U}) = \begin{pmatrix} 0 \\ \tau_{ij} \\ \tau_{ij} v_j - q_i \end{pmatrix} \tag{5}$$

where τ_{ij} and q_i are the viscous stress tensor and the heat flux vector, respectively, and represent diffusive effects of the fluid. In addition to the advection transport mechanism associated with the motion of the fluid, the fluid has the ability to transport momentum and energy via a diffusion process. In the absence of any diffusion, the viscous Navier–Stokes equations reduce to the *inviscid Euler equations* which account solely for advection. Since viscous effects are of primary concern for most practical aerodynamic problems, the Euler equations will not be further discussed.

The viscous stress tensor τ_{ij} requires a constitutive equation which relates the viscosity and spatial derivatives of the velocity to the stresses. For a Newtonian fluid (i.e., one which has a linear stress/strain relationship) the deviatoric stress tensor is often written as

$$\tau_{ij} = \mu \left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} \right) + \lambda \delta_{ij} \left(\frac{\partial v_k}{\partial x_k} \right) \tag{6}$$

where μ is the viscosity and λ is the bulk viscosity of the fluid. For a Newtonian fluid the bulk viscosity is often expressed as $\lambda = -2\mu/3$.

The heat flux vector q_i is a measure of the thermal energy flow and is typically written using Fourier's law

$$q_i = -\kappa \frac{\partial T}{\partial x_i} \tag{7}$$

where κ is the gas thermal conductivity and T is the gas temperature.

Lastly the source vector $\boldsymbol{S}$ is written as

$$\boldsymbol{S}(\boldsymbol{U}) = \begin{pmatrix} 0 \\ 0_j \\ 0 \end{pmatrix} \tag{8}$$

Equation of State For a calorically perfect gas, an equation of state is needed to relate two independent state variables to the third. Thus, the perfect gas equation of state is usually written as

$$P = \rho R T \tag{9}$$

where R is a constant specific to the type of gas (for air $R = 287.1\ J/kg/K$). The calorically perfect gas assumption has the following requirements: (1) the gas is in thermal equilibrium, (2) the gas is not chemically reacting, (3) the internal energy and enthalpy are dependent only on temperature, and (4) the specific heats (c_v and c_p) are constant.

In accordance with these assumptions, the internal energy and enthalpy are computed by the equations

$$e = c_\mathrm{v} T \quad , \quad h = c_\mathrm{p} T \tag{10}$$

and the specific heats are written as

$$c_\mathrm{v} = \frac{R}{\gamma - 1} \quad , \quad c_\mathrm{p} = \frac{\gamma R}{\gamma - 1} \tag{11}$$

where γ is the ratio of specific heats (for air $\gamma = 1.4$) and is expressed as

$$\gamma = \frac{c_\mathrm{p}}{c_\mathrm{v}}. \tag{12}$$

An alternative but equivalent form of the perfect gas equation of state can be obtained by writing the temperature as $T = e(\gamma - 1)/R$. Inserting this temperature expression into equation 9 we obtain the following form of the ideal gas equation

$$P = (\gamma - 1)\rho e. \tag{13}$$

The perfect gas assumptions begin to break down when the temperature of the fluid reaches roughly 600 K. The differences between a perfect gas model and a thermochemical gas model become important (on the order of 15–25% of QoI's) around 2000 K. Above this temperature a perfect gas approximation will significantly over–predict gas temperatures. It is noted that this limit usually coincides with a Mach number of approximately 5, which is generally considered the hypersonic limit where chemical reactions and thermal non–equilibrium begin to occur.

Transport Properties The viscosity and thermal conductivity transport properties needed in the preceding equations are computed according to Sutherland's law (or a number of other functional forms) and the constant Prandtl number to relate viscosity to thermal conductivity, respectively.

For the purposes of the performance analysis of the perfect gas model in this paper, we consider a laminar flow case (i.e., no turbulence model), giving 5 equations per mesh entity (i.e., cell).

2.2 Governing Equations for a Reacting Gas

The conservation of mass, momentum and energy for a gas in thermochemical non-equilibrium (i.e., in a state of chemical and thermal non-equilibrium) can be expressed as

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{F}_i(\boldsymbol{U})}{\partial x_i} - \frac{\partial \boldsymbol{G}_i(\boldsymbol{U})}{\partial x_i} - \boldsymbol{S}(\boldsymbol{U}) = \boldsymbol{0}. \tag{14}$$

The conservative variable vector $\boldsymbol{U}$ in equation 14 is written as

$$\boldsymbol{U} = \begin{pmatrix} \rho_s \\ \rho v_j \\ \rho E \\ \rho e_e \end{pmatrix} \tag{15}$$

where ρ_s is density of each species s, ρv_j are the momentum variables, ρE denotes the density times the total energy per unit mass, and e_e is the internal energy associated with each internal energy mode e being modeled for the molecules. Similar to a perfect gas, the total energy is the sum of all the fluid's internal energy modes and its kinetic energy

$$E = e_\mathrm{tot} + \frac{1}{2}(v_j v_j). \tag{16}$$

The inviscid flux vector $\boldsymbol{F}_i$ is defined as

$$\boldsymbol{F}_i(\boldsymbol{U}) = \begin{pmatrix} \rho_s v_i \\ \rho v_i v_j + P\delta_{ij} \\ \rho E v_i + P v_i \\ F^\mathrm{iem}_{i,e} \end{pmatrix} \tag{17}$$

where P is the pressure of the fluid and $F^\mathrm{iem}_{i,e}$ are internal energy mode inviscid fluxes associated with the i-th coordinate direction and the e-th internal energy mode equation. $F^\mathrm{iem}_{i,e}$ typically take the form of $F^\mathrm{iem}_{i,e} = \rho e_e v_i$ and will be further defined in the section on internal energy relaxation processes. The viscous

flux vector $\boldsymbol{G}_i$ is written by

$$\boldsymbol{G}_i(\boldsymbol{U}) = \begin{pmatrix} \rho_s v_{i,s} \\ \tau_{ij} \\ \tau_{ij} v_j - q_i + \sum_s \rho_s v_{i,s} h_s \\ G_{i,e}^{\text{iem}} \end{pmatrix} \tag{18}$$

where $\rho_s v_{i,s}$ are the species diffusion fluxes, τ_{ij} is the viscous stress tensor, and q_i is the heat flux vector. $G_{i,e}^{\text{iem}}$ are the viscous fluxes associated with the internal energy modes for the e-th internal energy mode equation. $G_{i,e}^{\text{iem}}$ typically take the form $G_{i,e}^{\text{iem}} = -q_{i,e} + \sum_s \rho_s v_{i,s} e_{s,e}$, where $e_{s,e}$ is the internal energy associated with species s for mode e.

The species diffusion fluxes $\rho_s v_{i,s}$ result from gradients in species, temperature and pressure. However, a valid assumption for hypersonic flows is that species diffusion is only driven by species gradients, so the species diffusion fluxes may then be expressed by Fick's law as

$$\rho_s v_{i,s} = -\rho D_s \frac{\partial c_s}{\partial x_i} \tag{19}$$

where D_s is the effective diffusion coefficient for species s and c_s is the mass fraction for species s ($c_s = \rho_s/\rho$).

The viscous stress tensor τ_{ij} remains unchanged from the form given in Equation 6. However, the heat flux is now a total heat flux which includes contributions from each internal energy mode and is written as

$$q_i = -\kappa \frac{\partial T}{\partial x_i} - \sum_e \kappa_e \frac{\partial T_e}{\partial x_i} \tag{20}$$

where κ_e is the thermal conductivity and T_e is the temperature associated with each energy mode.

Finally the source vector $\boldsymbol{S}$ is written as

$$\boldsymbol{S}(\boldsymbol{U}) = \begin{pmatrix} \dot{\omega}_s \\ 0_j \\ 0 \\ \dot{\omega}_e \end{pmatrix} \tag{21}$$

where $\dot{\omega}$ is the rate of species or internal energy production/destruction for species s and energy mode e, respectively.

Equation of State The equation of state for a reacting gas is assumed to be a mixture of thermally perfect gases and obeys Dalton's law of partial pressures. As such, it is written as

$$P = \sum_s P_s = \sum_s \rho_s \frac{R_{\text{univ}}}{M_s} T \tag{22}$$

where R_{univ} is the universal gas constant ($R_{\text{univ}} = 8314.45 J/kmol/K$) and M_s is the molecular mass for species s.

The internal energy state for a reacting gas depends on whether or not the gas is in thermal equilibrium. For diatomic molecules, which we have when the gas we are considering is air (composed of N_2 and O_2), we consider three modes in which the molecule may contain internal energy: a translational mode, a rotational mode, a vibrational mode (due to back–and–forth vibration between the two atoms). For the case of thermal equilibrium, these modes are assumed to be in equilibrium with each other, and a single internal energy representation is sufficient (which is governed by a single temperature T). For the case of thermal non–equilibrium, we must assume that each mode has the potential to carry its own internal energy, and that each internal energy is governed by its own temperature, a translational temperature T_{t}, a rotational temperature T_{r}, and a vibrational temperature T_{v}. Conditions for thermal non–equilibrium occur when the gas is in a state of low pressure and high temperature with large temperature gradients. Such conditions exist in high temperature expansions and behind strong shock waves.

A common assumption for hypersonic flows is that, since translational and rotational energy modes equilibrate very quickly, these modes may be governed by a single translational/rotational temperature, that

is $T = T_{\rm tr} = T_{\rm t} = T_{\rm r}$. For a non–ionized flow, this means that the vibrational energy modes of the molecules must be modeled by a separate temperature, $T_{\rm v}$. It is typically assumed that one vibrational temperature applies to all species, but it's also possible to treat each molecule with its own vibrational temperature.

The total internal energy, $e_{\rm tot}$, first expressed in Equation 16 is written as the sum of all the individual energy modes

$$e_{\rm tot} = \underbrace{\sum_s c_s e_{{\rm t}_s}}_{e_{\rm t}} + \underbrace{\sum_s c_s e_{{\rm r}_s}}_{e_{\rm r}} + \underbrace{\sum_s c_s e_{{\rm v}_s}}_{e_{\rm v}} + \sum_s c_s h_s^o \tag{23}$$

where c_s are the species mass fractions and h_s^o are the heats of formation for each species. The details of computing the species translational, rotational and vibrational internal energies are not included here for the sake of brevity.

Transport Properties As with a perfect gas, the viscosity μ and thermal conductivity κ are needed to evaluate the viscous stress tensor τ_{ij} and the heat flux vector q_i when the gas is reacting. Models are available for individual species viscosities and thermal conductivity and then a mixing rule is applied to determine a mixture viscosity and thermal conductivity.

Wilke's mixing rule is used to compute a mixture viscosity and thermal conductivity from individual species values. Blottner's model is used to compute species viscosities while the thermal conductivities for the translational, rotational and vibrational internal energy modes are often computed using the Eucken relation, which relates the viscosity to the thermal conductivity via translational, rotational and vibrational specific heats. Finally, species diffusion coefficients D_s appearing in Equation 19 are assumed equal if the molecular weights of the species are similar. A constant Lewis number, Le, assumption is then used to compute a single binary diffusion coefficient.

Species Source Term - Chemical Kinetics The species production/destruction rates, $\dot{\omega}_s$, appearing in Equation 21 have yet to be defined. These rates are dependent on the particular chemical reaction set being considered. For air, a 5–species and an 11–species model are common choices for hypersonic flows. The species source terms are written in general form as

$$\dot{\omega}_s = M_s \sum_r \left(\alpha_{sr} - \beta_{sr}\right) \mathcal{R}_r \tag{24}$$

where the sum is over the number of reactions r in the reaction set, α_{sr} are the stoichiometric coefficients of the reactant species s for reaction r, and β_{sr} are the stoichiometric coefficients of the product species s for reaction r. The reaction rate for reaction r, $\mathcal{R}_r$, is the difference between the reverse and forward reaction rates

$$\mathcal{R}_r = \mathcal{R}_{{\rm rev}_r} - \mathcal{R}_{{\rm fwd}_r} \tag{25}$$

where the reverse reaction rate is written as

$$\mathcal{R}_{{\rm rev}_r} = k_{{\rm rev}_r} \prod_s \left(\frac{\rho_s}{M_s}\right)^{\beta_{sr}} \tag{26}$$

and the forward reaction rate is

$$\mathcal{R}_{{\rm fwd}_r} = k_{{\rm fwd}_r} \prod_s \left(\frac{\rho_s}{M_s}\right)^{\alpha_{sr}}. \tag{27}$$

In Equations 26 and 27, $k_{{\rm rev}_r}$ and $k_{{\rm fwd}_r}$ are the forward and backward rate coefficients, respectively. The forward rate coefficient is governed by an Arrhenius rate equation as

$$k_{{\rm fwd}_r} = C_{{\rm fwd}_r} \exp\left(\eta_r \ln T_{\rm eff} - \theta_r / T_{\rm eff}\right) \tag{28}$$

where $T_{\rm eff}$ is a function of the various internal energy temperatures, $C_{{\rm fwd}_r}$ is the reaction rate constant, η_r is Arrhenius pre–exponent factor, and θ_r is the activation energy. For thermal equilibrium, $T_{\rm eff} = \max\left(T, T_{\rm rmin}\right)$, where $T_{\rm rmin}$ is a minimum reaction temperature which helps mitigate spurious behavior of

the reaction model at very low temperatures. For the case of vibrational non–equilibrium, a commonly employed effective temperature is $T_{\text{eff}} = \sqrt{TT_{\text{v}}}$, which states that the reaction rate is a function of the geometric mean of the translational/rotational temperature and the vibrational temperature.

5-Species Air Model The 5-species air model consists of 5 species, N_2, O_2, NO, N and O and 5 reactions

$$\begin{aligned} N_2 + \mathcal{M} &\leftrightharpoons 2N + \mathcal{M} && (\text{r} = 1, \text{dissociation}) \\ O_2 + \mathcal{M} &\leftrightharpoons 2O + \mathcal{M} && (\text{r} = 2, \text{dissociation}) \\ NO + \mathcal{M} &\leftrightharpoons N + O + \mathcal{M} && (\text{r} = 3, \text{dissociation}) \\ N_2 + O &\leftrightharpoons NO + N && (\text{r} = 4, \text{exchange}) \\ NO + O &\leftrightharpoons O_2 + N && (\text{r} = 5, \text{exchange}) \end{aligned} \tag{29}$$

where $\mathcal{M}$ is a collision partner, which may be any of the species. After following the preceding development, the species source terms for the 5-species model are

$$\begin{aligned} \dot{\omega}_{N_2} &= M_{N_2} \left(\mathcal{R}_1 + \mathcal{R}_2 \right) \\ \dot{\omega}_{O_2} &= M_{O_2} \left(\mathcal{R}_2 - \mathcal{R}_5 \right) \\ \dot{\omega}_{NO} &= M_{NO} \left(\mathcal{R}_3 - \mathcal{R}_4 + \mathcal{R}_5 \right) \\ \dot{\omega}_{N} &= M_{N} \left(-2\mathcal{R}_1 - \mathcal{R}_3 - \mathcal{R}_4 - \mathcal{R}_5 \right) \\ \dot{\omega}_{O} &= M_{O} \left(-2\mathcal{R}_2 - \mathcal{R}_3 + \mathcal{R}_4 + \mathcal{R}_5 \right). \end{aligned} \tag{30}$$

For the purposes of the performance analysis of the reacting gas model in this paper, we consider the 5-species air model, with a single energy equation and no turbulence model, giving 9 equations per mesh entity (i.e., cell).

2.3 Spatial Discretization

The conservation equations listed in Equation 1 and Equation 14 are currently discretized within `SPARC` using a cell–centered finite–volume method on both structured and unstructured grids. The basic control volume is a cell (Ω^c), which is composedSS of any number of faces on the boundary of the cell (Γ^c). The number of faces depends on the cell topology (six faces for a hexahedron, in the case of structured and unstructured grids; five faces for a pentahedron, five faces for a pyramid, and four faces for a tetrahedron, in the case of unstructured grids). A domain Ω may contain any number of cell volumes Ω^c that must be integrated over in order to discretize the equations within the domain. Finite volume methods utilize the Divergence Theorem to turn the task of integrating the divergence of a vector field over a volume into integrating the flux of that vector field over a surface. Thus, the integration of these equations is written as

$$\int_{\Omega^c} \frac{\partial \boldsymbol{U}}{\partial t} dV + \int_{\Gamma^c} \left(\boldsymbol{F}_i(U) - \boldsymbol{G}_i(U) \right) dA_i - \int_{\Omega^c} \boldsymbol{S} dV = \boldsymbol{0} \tag{31}$$

where dV and dA_i are the infinitesimal cell volume and face area vector, respectively. The volume and surface integrals in equation 31 are approximated by numerical integration. The volume integrals are approximated for cell c as

$$\int_{\Omega^c} \frac{\partial \boldsymbol{U}}{\partial t} dV \approx \frac{\partial \boldsymbol{U}^c}{\partial t} V^c \quad , \quad \int_{\Omega^c} \boldsymbol{S} dV \approx \boldsymbol{S}^c V^c \tag{32}$$

and the surface integrals are approximated for the faces f enclosing a cell as

$$\int_{\Gamma^c} \left(\boldsymbol{F}_i(U) - \boldsymbol{G}_i(U) \right) dA_i \approx \sum_f \left(\boldsymbol{F}_i^f(U) n_i - \boldsymbol{G}_i^f(U) n_i \right) A^f \tag{33}$$

where n_i is the unit normal face area vector and A^f is the face area.

Thus, the semi-discrete form of Equation 31 can be written as

$$\frac{\partial \boldsymbol{U}^c}{\partial t} V^c + \sum_f \left(\boldsymbol{F}_i^f(U) n_i - \boldsymbol{G}_i^f(U) n_i \right) A^f - \boldsymbol{S}^c V^c = \boldsymbol{0}. \tag{34}$$

Both the Roe and the modified Steger-Warming scheme are used for upwind evaluation of the inviscid fluxes. A variable reconstruction procedure is used to obtain second-order accuracy, along with a slope limiter to control the reconstruction behavior at strong gradients. A simple face averaging of the flow variables is used for evaluation of the viscous fluxes. Gradients are computed by either a Green-Gauss or weighted least-squares (WLS) procedure. These gradients are then interpolated to the face integration point and corrected via a non–orthogonal correction technique. The non–orthogonal correction improves the quality of the interpolated gradients between highly stretched or highly distorted cells by correcting for the difference between the face area vector and the vector between the two neighboring cells.

2.4 Time Integration

Time integration of the discretized governing equations is performed via a four-stage explicit Runge-Kutta integrator or a backward Euler implicit integrator. For implicit solution, Newton's method is used to solve the nonlinear equations at each time step. Several linear equation solvers are available to solve the linearized set of equations, including both relaxation solvers and Krylov solvers. For the performance analysis work in this paper, we focus on implicit integration with a block tri-diagonal (line-implicit) fixed-point iteration solver.

2.5 Verification and Validation

`SPARC` is a relatively new code, having formally begun development in mid-2015. Much of its early development was focused primarily on addressing performance portability for the code. In mid-2017 we began a campaign to establish credibility in `SPARC`'s simulation abilities, especially for the use cases that are central to Sandia's mission. We started, and continue, to do this by following a rigorous verification and validation process. This work has focused on solution and code verification for canonical problems and code validation against experimental data. Recent validation work has considered hypersonic double-cone experiments and HIFiRE-1 experiments, both of which are ground tests conducted in Calspan-University of Buffalo Research Center's (CUBRC) shock tunnel. This verification and validation work will be documented in a number of upcoming papers [9, 10, 11, 12].

3 Performance Portability and Multiple Levels of Parallelism in a Compressible CFD Code

Sandia National Labs' primary strategy for achieving performance portability across next-generation computing platforms in application codes centers around a parallel programming model called Kokkos [13, 14]. The Kokkos project provides both a shared-memory parallel programming model, and a C++ library implementation of that model. Applications can use Kokkos to write a single parallel code that builds, runs, and performs well on a variety of computer architectures with minimal architecture-specific code. These architectures include GPUs, "many-core" CPUs like Intel Xeon Phi, and traditional multicore CPUs. Its C++ implementation relies only on standard C++ language features supported by most compiler vendors.

As mentioned in the Introduction, a trend in the new architectures being developed for HPC is to add more parallelism at the node level. This typically means greater numbers of threads per device as well as wider vector units. MPI for parallel communication between nodes is still important, but on-node parallelism is increasingly something that cannot be ignored to fully exploit the potential of these new architectures. This has given rise to the terminology "MPI+X" when referring to the parallel programming paradigms of the future, where "X" refers to the on-node parallel strategy/programming model. We employ multiple levels of parallelism, using MPI for inter-node parallelism, and MPI, threads, and vectorization for on-node

parallelism. This section discusses how we have structured our algorithms and data to use these multiple levels of parallelism.

Throughout this section and the rest of the paper, we will use the following terms. For heterogeneous architectures, we use *host* to refer to the processing units, usually CPUs, that run the main executable code and *dispatch kernels* to the *device(s)*. There may be multiple host CPUs and multiple devices on a single compute node. The *device(s)*, in this context sometimes referred to as 'accelerators', execute *kernels*. A *kernel* is a computational operation that performs some unit of work; in the context of a CFD code, kernels execute flux integration functions, source term functions, and appear in linear equation solvers, among others. On an Nvidia GPU, a kernel is associated with a CUDA global function, and threads are launched to execute the operations on the device. On a CPU or Xeon Phi, a kernel is associated with an OpenMP parallel region, again where threads are launched to execute the operations on the processors. Threads may execute independently or in a hierarchical nature, often called thread teams; the later mode introduces an additional degree of parallelism at the thread level and is distinguished by shared-memory access and team-level synchronization. Within each thread, operations may be *vectorized* via *SIMD* (single-instruction, multiple-data) instructions, introducing yet another level of parallelism to the operations. Lastly, *dispatch* refers to the mechanism by which functions are invocated for execution; this can occur either *dynamically* (i.e., at run-time) or *statically* (i.e., at compile-time).

3.1 High-Level Design

SPARC has the following high-level structure. The `Problem` class is responsible for orchestrating the solution to a problem, where this may be a single-physics forward analysis, a coupled-physics forward analysis, or a more advanced analysis such as sensitivity analysis or solving an optimization problem. The `Problem` class creates one or more `TimeSolver` classes depending on the nature of the analysis. Each `TimeSolver` performs an explicit or implicit time integration. Within each `TimeSolver` is a `NonlinearSolver` which is responsible for solving a set of discretized nonlinear equations. The `NonlinearSolver` calls upon a `Model`, e.g., `AeroModel`, class to compute the discrete residual vector and, if using an implicit time integrator, the discrete linearization of the residual (i.e., the Jacobian matrix). The `Model` calls a set of `Algorithm` classes which compute physical quantities for the discretization being used. For example, one such `Algorithm` computes the inviscid and viscous fluxes and flux Jacobians for the equations outlined in Section 2.1 over all the faces of a mesh that has been discretized with a cell-centered finite volume method. The intent of the `Algorithm` design is to re-use "physics models," that is gas models and flux functions, for any given discretization. The `Algorithm` classes loop over discretization blocks, calling `Kernel` classes, which map the work to be done to an on-node parallel construct. In the simplest case, a `Kernel` implements a `Kokkos::parallel_for`. We use a *kernel composition* approach to pull together a complete set of "physics models" that are dispatched dynamically (less performant) or statically (more performant and required for the GPU in our implementation). The topics of dispatch and kernel composition will be discussed more later in this section. Last in the set of high-level classes is the `LinearSolver` class, which solves the set of linearized equations that have been assembled by the `NonlinearSolver`. As mentioned in Section 2.4, `SPARC` typically uses a block tri-diagonal solver for steady-state hypersonics problems and a block Jacobi solver for unsteady hypersonics problems; these two solvers are available either as natively implemented solvers in `SPARC` or performance-portable implementations are available through the `Trilinos` package [15]. Additionally, `SPARC` has an interface to `Trilinos`' Krylov solvers and more advanced pre-conditioning such as multigrid pre-conditioners. The topic of performance-portable linear solvers will be elaborated on later in this section.

This high-level design description is given to establish context and a nomenclature for the remainder of this paper. The majority of the work in creating performance portability and introducing multiple-levels of parallelism happen at the `Kernel` and `LinearSolver` levels; the rest of the description was given to create an overall view of how the code has been structured to get to the point of calling these two important and fundamental constructs.

3.2 Data Layout

Data layout is critical for achieving high-performance for any single architecture. When trying to achieve high performance for multiple architectures, one must have control over the data layout, since the optimal layout for one architecture (e.g., Xeon Phi) is almost certainly not optimal for another architecture (e.g., GPU). A distinguishing feature of the `Kokkos` programming model is that it integrates data layout (via customizable multi-dimensional arrays called `Views`) with thread-parallel constructs (such as `parallel_for`). With `Kokkos`, we have the flexibility to modify our data layout and parallel strategy for a given architecture and to do this at compile time. The ability to do this is key for achieving performance portability.

`SPARC` supports both unstructured and structured mesh data structures. The unstructured mesh data structures employ a two-dimensional `View` for each mesh block with the first dimension being the number of entities in the block (e.g., cells for a cell-centered discretization) and the second dimension being the number of state-variables for the given set of equations being solved. Each kernel traverses through the discretization blocks and performs its operations. Each kernel class for an unstructured mesh implements a function having the signature

```
void operator() (int entity_id) const;
```

where (*entity_id*) is an index into the entities of an unstructured block, used to retrieve data for the current entities and for indirect addressing of data for neighboring entities. C++ developers call a class that implements `operator()` a *functor*. In this case, `operator()` implements one iteration of a `for` loop. Functors that implement `operator() (IntegerType i) const` for any built-in integer type IntegerType, can serve as the loop body of a `Kokkos::parallel_for`. When it comes time to assemble the result of the operations into numerical linear algebra (NLA) data structures, we must deal with the problem of race conditions inherent to on-node programming models. There are several approaches we can take to avoid race conditions, some of which are (a) thread locks, (b) atomic operations, and (c) graph-coloring. Prior work in the literature and our own investigations on the most performant approach across different architectures led us to choose graph coloring. We perform a 3D graph coloring during initialization and keep this coloring as long as the mesh does not change. Graph coloring ensures that different threads will never attempt to write to the same memory location concurrently in an NLA data structure. This means that we do not need to use atomic operations, which we found to be slower. We also reorder entities within each color to optimize the memory access pattern.

The structured mesh data structures employ a four-dimensional `View` for each mesh block. The first three dimensions denote the index triplet of the structured block entities and the fourth dimension provides the state-variable storage. Each kernel class for a structured mesh implements a function having the signature

```
void operator() (int k, int j, int i) const;
```

where (k, j, i) is an index into a structured block. Traversing a structure grid calls for a triply nested loop. In our current implementation, we handle this with an outer `Kokkos::parallel_for` over a single index, and manually implement a loop-collapse construct for the two inner loops. Recent versions of `Kokkos` provide multidimensional `parallel_for` and `parallel_reduce` that handle loop collapse automatically, and also let users easily control optimizations like tiling and loop order. We plan to explore these Kokkos features in future work. Similar to assembly on an unstructured mesh, we use graph coloring for assembly into the NLA data structures for a structured mesh. At present, we use a simple even-odd graph coloring, meaning we traverse through the mesh, one i, j, and k line at a time, with the kernel performing its operations on the odd-numbered entities first and then traversing through the mesh a second time, with the kernel operating on the even-numbered entities second. In this case, memory access is stride-1 for the inner-most i index dimension, stride-i for the middle j index dimension, and stride-ij for the outer-most k index dimension. As the stride size increases, memory access becomes more costly. This observation will be recalled later in Section 4 and is a motivation for pursuing a tile-based data layout for structured grids which breaks data into smaller structured chunks to improve memory access.

For GPU architectures, the customizability of `Kokkos`' `View` data layout is important; we observe best performance when the four-dimensional arrays used for structured grids takes the layout (k, j, v, i). This layout places the variable data storage in the second innermost access position and gives a strided or "coalesced" memory access pattern. This same memory access pattern applies to the two-dimensional arrays

used for unstructured grids, taking the form (v, i). A typical (k, j, i, v) layout is preferred and used for CPU and KNL architectures.

3.3 Dispatch

The topic of function dispatch within a kernel was discussed in detail in prior work by the authors [16]. We'll summarize this topic here because it is an important one when considering design for performance portability with GPU architectures. Dispatch of a function can occur either *dynamically* (i.e., at run-time) or *statically* (i.e., at compile-time). The next concern is at what level do functions need to be dispatched; they may be called at a higher level and operate on "worksets" of data or at a lower-level and operate on individual entities and their data.

We have found it more natural to write our functions at the entity level, such that the evaluation of a flux across a face or the source term of a cell is considered one at a time. This decision dictated our choice of dispatch. Regarding the GPU, we needed a strategy to dispatch an object's function calls from the device. Dynamic dispatch (i.e., making a virtual function call) on a GPU is possible by constructing the object on the GPU. This is a necessity because constructing it on the host and copying to the GPU means that it will exist in a different address space and it will have invalid memory access. Dynamic function dispatch on the GPU in inner loops is costly, so we avoided this approach. Instead, we opted for static dispatch via kernel composition. We do this by composing the full object/function hierarchy via `C++` templates and the complete kernel at compile time. Then at runtime, the appropriate kernel is selected and run. While kernel composition and static dispatch lead to higher-performing code, it comes at the cost of increased build time. This is an area of concern that has to be continually managed; we've opted for an explicit template instantiation (ETI) approach to manage the combinations of objects/functions that must be chained together into only the necessary set. This has worked well but compile time and executable size can still get large. To handle this, on CPU platforms we allow for a mix of static and dynamic dispatch, choosing to use dynamic dispatch for boundary condition codes where we are less concerned with performance and where the sheer number of kernels to be composed is large. On GPU platforms, we must use static dispatch and we are investigating using relocatable device code (RDC) to reduce compile time and executable size; the performance cost of this has yet to be determined. Given the possibility of mixing static and dynamic dispatch, we use a `Dispatcher` class that handles either run-time or compile-time function invocation. Consider that we want to call the `computeFlux()` method on a derived class `RoeFlux` which inherits from the base `Flux` class, we have a pointer to `RoeFlux`'s base class `Flux` called `object` and we want to call the method either statically or dynamically. An example `Dispatcher` for dynamic dispatch is as follows

```
template <bool is_dynamic> struct Dispatcher
{
  template <typename Type>
  static void computeFlux (Type* object) {
     object->computeFlux();
  }
};
```

where the `is_dynamic` template parameter controls static or dynamic dispatch and for this example `Type` is the `Flux` base class. This first dispatcher simply provides a call through to `RoeFlux`'s virtual method `computeFlux` via the base class object. This is then called with

```
Dispatcher<true>::computeFlux<Flux>(object);
```

An example `Dispatcher` for static dispatch is as follows

```
template <> struct Dispatcher<false>
{
  template <typename Type>
  static void computeFlux (Flux* object) {
     static_cast<Type*>(object)->Type::computeFlux();
  }
};
```

where we now have a template specialization for `is_dynamic = false`. This is called as follows

```
Dispatcher<false>::computeFlux<RoeFlux>(object);
```

This statically casts our base class object to a `RoeFlux` and calls the `computeFlux()` method directly, thus avoiding the virtual function call. This former case is more performant on CPU platforms and is our design pattern for enabling lower-level polymorphism on the GPU.

3.4 Kernels

In `SPARC`, kernels are the construct that pull together data layout and mesh traversal to dispatch a function using an on-node parallel pattern (such as `parallel_for` or `parallel_scan`). The kernel is given access to the data arrays and the functions that use that data to do its operations. The parallel pattern being used ultimately calls the kernel's `operator()` functor mentioned in Section 3.2. On CPU architectures, Kokkos calls OpenMP's parallel patterns. On GPU architectures, Kokkos calls CUDA global functions. Several important kernels in `SPARC` are `ComputeResidualVolume` and `ComputeResidualBC`, which compute inviscid and viscous fluxes and flux Jacobians over the discretization blocks and assemble the results in the MPI-distributed NLA residual vector and Jacobian matrix.

An example kernel for a structured grid is given in the listing below, which shows the salient features of the previous discussion.

```
template <typename Scalar, typename GasModelType, typename KokkosExeSpace>
class ComputeResidualVolumeKernel :
   public GasModelKernel<Scalar, GasModelType, KokkosExeSpace>,
   public MeshTraverserKernel<ComputeResidualVolumeKernel<Scalar, GasModelType,
       KokkosExeSpace>, KokkosExeSpace>
{
private:
   typedef MeshTraverserKernel<ComputeResidualVolumeKernel<Scalar, GasModelType,
       KokkosExeSpace>, KokkosExeSpace> MeshTraverser;
   const Array4D<Scalar> cell_primitive_vars;
   ...

public:
   ComputeResidualVolumeKernel(const StructuredBlock& blk, const
       MaterialCompFluid<Scalar>& gasmodel, const Array4D<Scalar>& cell_V, ...) :
      GasModelKernel<Scalar, KokkosExeSpace, GasModelType> (gasmodel),
      MeshTraverser(blk.kmin, blk.kmax, blk.jmin, blk.jmax, blk.imin, blk.imax),
      cell_primitive_vars(cell_V),
      ...
   { /* constructor code */ }

   KOKKOS_FORCEINLINE_FUNCTION
   void compute(const Index& k, const Index& j, const Index& i) const
   { /* kernel functor code */ }

   void Run() const
   {
      this->MeshTraverser::Run(); // traverse through the current mesh block,
          running the above kernel functor code for each mesh entity
   }
}
```

In this listing, the `ComputeResidualVolumeKernel` class derives from a `GasModelKernel` class, which provides functionality to ensure that data associated with the perfect or reacting gas models is accessible from the current memory space (host memory or device/GPU memory). `ComputeResidualVolumeKernel` also derives from the `MeshTraverserKernel`, which as previously discussed, provides an architecture optimized means of employing a parallel pattern to perform the `compute()` functor operations on a structured mesh block. The class constructor initializes its base classes and stores references to needed data arrays; in

this case an Array4D, which is a wrapper for a four-dimensional Kokkos::DualView. The compute() functor contains the bulk of the computational operations, possibly making calls to the gas model, computing fluxes, flux Jacobians, etc. This method is decorated with the KOKKOS_FORCEINLINE_FUNCTION macros, which annotates the method for Kokkos to call in parallel. Lastly, the Run() method invokes MeshTraverser::Run(), which employs a loop-collapsed parallel_for to call the compute() functor for each mesh entity.

This basic kernel pattern is replicated for each significant computational operation for which we wish to dispatch operations in a thread-parallel fashion. The majority of SPARC is kernelized in this way and is the key to portably and performantly executing the core assembly and solve operations on various architectures.

3.5 Linear Solvers

Design for performance portability across architectures in a linear solver follows many of the same principles as it does for an application code such as SPARC; this requires wise data layout and employing on-node parallel constructs to effectively use threads and vectorization in a portable way. There are several linear solver options available to SPARC. Some are implemented natively to SPARC (block Jacobi and block tri-diagonal solvers) and use OpenMP only. Others are available through the Trilinos package, such as block Jacobi and block tri-diagonal solvers as well as Krylov solvers. The solver we will focus on here is the block tri-diagonal solver available through Trilinos' Ifpack2 [17], which uses BLAS routines from KokkosKernels. KokkosKernels [18] implements computational kernels for linear algebra and graph operations using the Kokkos parallel programming model.

On a CPU or KNL, the KokkosKernels block tridiagonal solver uses a single level of threading and a SIMD-friendly data layout to achieve optimal performance on these devices; the reader is referred to the paper on this work for further details of the implementation [19]. The key piece of this solver is a compact data layout that allows for efficient SIMD vectorization on these architectures. On the GPU, the best performance achieved to date has employed a hierarchical threading strategy and also uses SIMD-friendly data layout. Work on optimizing for the GPU is ongoing and currently undocumented.

4 Performance Analysis

In this section we present and discuss a performance analysis of SPARC on multiple architectures: traditional CPUs, many-core CPUs, and GPUs. ARM-based systems are potential alternatives to these three primary architectures, and NNSA has recently announced the procurement of the world's largest ARM-based HPC system, which will be sited at Sandia [20]. However, an ARM system of reasonable scale is not yet available to us and is not included in this work. Two cases will be considered, both using the "Generic Reentry Vehicle" (GRV) geometry in a high Mach number flow. The GRV has evolved as useful test problem which is representative of the aerodynamic and aerothermodynamic analysis use cases Sandia supports. Due to the sensitive nature of Sandia's work in aerothermodynamics for reentry, we give no specific details of this problem other than that we use the finite-volume discretization in SPARC, and we perform a steady-state analysis using implicit time integration and a block tri-diagonal solver. Additionally, we restrict this work to consideration of the structured grid discretization; prior performance analysis revealed that the unstructured implementation performs 10-30% better than the structured implementation [16]. This was attributed to the linear memory access created during the graph coloring and initialization. As previously discussed, memory access for the structured implementation is strided in the j and k dimensions which is less optimal; this has motivated current work on tile-based data layouts for structured grids. The remaining details of the problem are largely irrelevant to this study since the purpose is to assess the performance portability strategy we have employed. The first performance analysis case utilizes the perfect gas model discussed in Section 2.1 and the second case utilizes the 5-species air model presented in Section 2.2.

Both a strong and weak scaling study was performed for each of the above two cases. Analysis on traditional CPU architectures was performed on a Intel Xeon Broadwell (denoted CTS-1/BDW) and on an Intel Xeon Haswell system (denoted ATS-1/HSW). Analysis on Intel's Xeon Phi architecture was performed on an Intel Xeon Phi Knights Landing system (denoted ATS-1/KNL). Analysis on Nvidia GPU architectures was performed on a multiple testbed systems. The first GPU system, denoted CTS-1/P100, has compute nodes containing a two Intel Xeon Broadwell processor and two Nvidia Pascal 100 GPUs per host CPU. The

second GPU system, denoted ATS-2/P100, is similar to actual ATS-2 hardware but is a generation prior; it has compute nodes with two IBM Power8 CPUs and two P100 GPUs per host CPU. The third GPU system, denoted ATS-2/V100, has actual ATS-2 nodes, which have two IBM Power9 CPUs and two Nvidia Volta 100 (V100) GPUs per host CPU. All of these systems are a good representation of the HPC landscape available to the NNSA labs for capacity (moderate scale) and capability (highest scale) computing for the next several years.

In each plot that follows, the abscissa is the number of compute nodes or GPUs. For CTS-1/BDW and ATS-1/HSW, each node has two CPUs. For ATS-1/KNL, each node has a single KNL, which were always run in "quad_cache" mode. This mode treats the HBM as a cache for the main DDR memory. If the problem fits entirely into HBM, "quad_flat" mode performs slightly better but requires that the nodes be rebooted into this mode and for this reason is less practical unless the memory requirements for any given problem are known ahead of time. For any GPU node, we consider the unit of measure to be a single GPU. The ordinate in these plots is $\log_2$ time per time step in seconds. Ideal strong scaling should follow a constant slope line on these plots. Ideal weak scaling should follow a line of slope 1 (i.e., a horizontal line). The GPU testbed platforms are fairly limited in size, with the ATS-2 systems having no more than 40 GPUs and the CTS-1 system having 120 GPUs. For CPU and KNL, the legend in each plot has "x threads," where x is used to denote the number of threads per rank. For Broadwell and Haswell nodes, we employ 32 ranks per node. For KNL nodes, we employ 64, 32, and 16 ranks per node with 4, 8, and 16 threads per rank, respectively.

For each scaling analysis, six timings are reported. The first, "Compute Residual: Boundary Terms," is the on-node only (i.e., no MPI time included) time taken to compute fluxes and flux Jacobians on the mesh boundaries and assemble the result into the distributed NLA objects. This is generally a very small fraction of the total time taken for each time step. The second, "Compute Residual: Interior Terms," is the on-node time taken to compute fluxes and flux Jacobians for the interior of the mesh and to assemble the result. For the assembly phase, this is by far the most compute intensive kernel. The third, "Post Solve Update," is the time taken to compute all quantities needed for the "Compute Residual" routines. This includes computing primitive variables from conserved variables, quantities that are derived from the the primitive variables and gradients of a subset of these variables, in addition to other small operations. "Post Solve Update" includes time spent doing MPI communication for primitive variables and gradients. The fourth timing is "Equation Assembly," which is the total time taken for a time step minus the linear equation solve time. This captures any remaining operations not a part of the first three timers. The fifth timing is "Linear Equation Solver," which is simply the time taken to solve the set of assembled equations at each time step. Generally speaking, for the GRV cases considered here, 10-40% more time is spent solving the equations than time is spent assembling the equations. The sixth and last timer is "Total Problem Solve," which is the start-to-end time for each individual time step.

4.1 GRV - Perfect Gas Model

This analysis is similar to the analysis we reported in prior work [16] but with the following important differences. Prior work did not have all aspects of the code yet implemented and kernelized. This is important because we now have nearly all physics models (RANS turbulence, reacting gas models) implemented, which adds complexity to the kernels; at the time the prior work was conducted the code had a limited physics simulation capability. Additionally, prior work did not include linear equation solve times for GPU systems. At the time we did not have a reasonably performant block tri-diagonal solver for GPUs. The `KokkosKernels` solver we have now for GPUs is still under active development, and in many regards the performance shown here for this solver is preliminary.

4.1.1 Strong Scaling

A strong scaling analysis was performed on a 32 million cell multi-block structured grid. Figure 1 shows the performance for several architectures and MPI rank/thread configurations. The commentary below will be discussed relative to Haswell and Broadwell results.

Figure 1a shows the boundary term evaluation scaling. We note here that this kernel scales well for traditional CPUs, does not scale well for many-core CPUs and performs slower and does not scales well for GPUs. There is very little work to be done for this kernel and the kernel launch time, especially for the

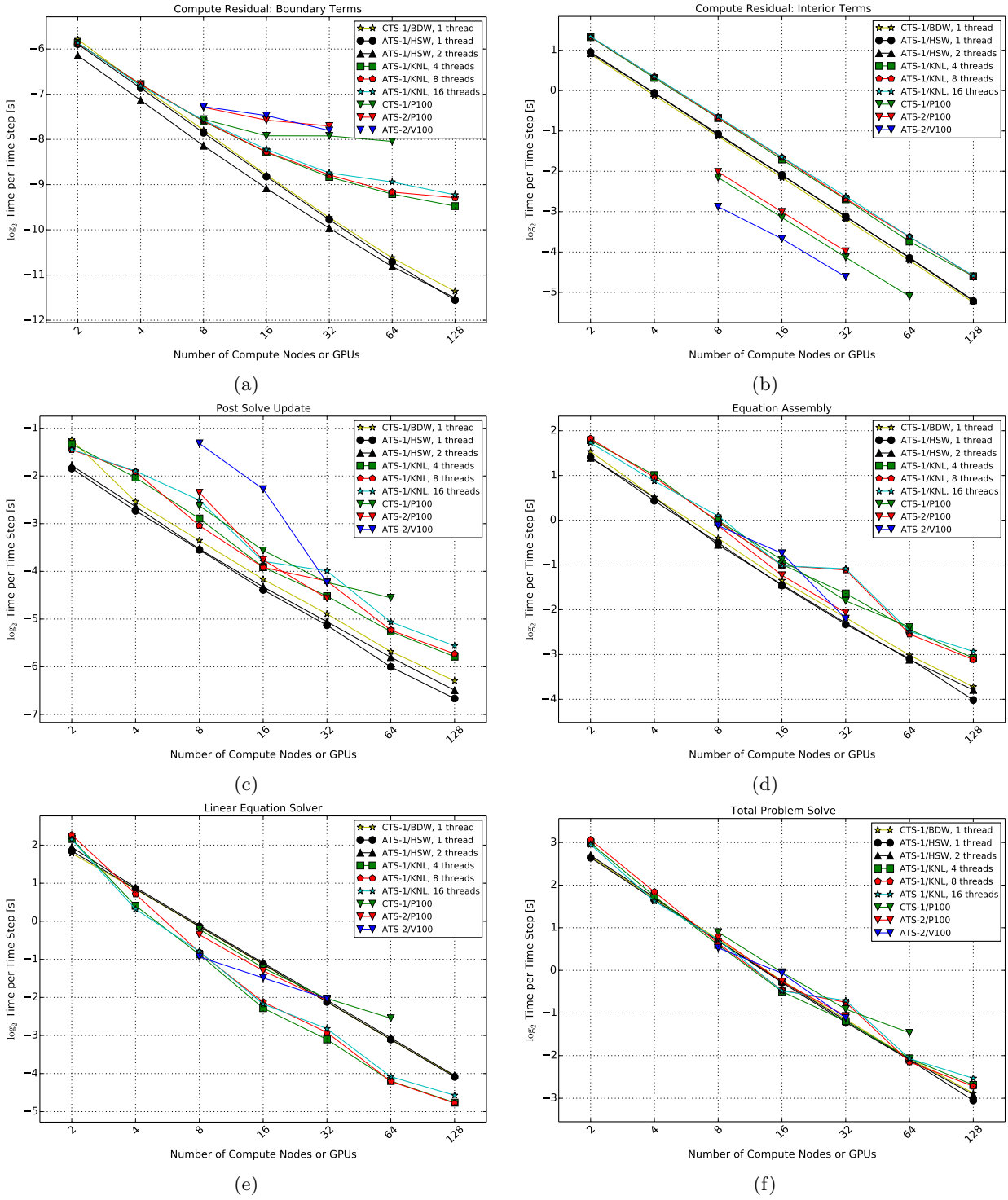

Figure 1: GRV, perfect gas model, implicit line solver, strong scaling

GPU, is not insignificant; it is evident that spreading the work to more devices and launching more kernels has diminishing returns. This kernel is insignificant in terms of the overall equation assembly time but it remains to be seen what impact this has at very high node/GPU counts.

Figure 1b shows the interior term evaluation scaling. All architectures are scaling quite well. KNL, for any threading count, performs $\sim 1.3\times$ slower than HSW/BDW. Other experiences have indicated that the assembly performance of PDE codes running on for KNL processors benefits more from SIMD vectorization than it does from threading [21]. For P100, this kernel is $\sim 2\times$ faster than HSW/BDW and for V100, is $\sim 4\times$ faster. This is a rather nice result for this significant kernel running on GPU architectures.

Figure 1c shows combined time of set of kernels that execute after the linear equation solve has taken place. This is grouped in the total equation assembly time because this is a necessary operation for computing the linear system to be solved. Because kernel execution and MPI communication time are grouped together here we cannot delineate the relative timings for each. We can say that this operation is performing $\sim 1.2\times$ - $1.5\times$ slower on KNL and P100 systems. For KNL, prior work has shown that MPI communication time goes as increase the thread count (and decrease the MPI rank count), however, gradient computation time takes $\sim 1.3\times$ longer on KNL than on HSW/BDW. The P100 systems are comparable to KNL, which is likely results from faster gradient kernel operations but generally slower MPI communication. The V100 result is $\sim 2\times$ - $4\times$ slower than HSW/BDW. We attribute this to the immaturity of the MPI software stack on this very new hardware.

Figure 1e shows the linear equation solve time. Here we note really good performance and strong scaling as the node count increase when running solver running on KNL relative to HSW/BDW. The nonlinearity and poorer scaling at the lower node counts is attributed to the fact that problem does not fit entirely into HBM, so there is no memory bandwidth benefit. At 16 nodes, this problem fits entirely into HBM and a significant speedup ($\sim 2\times$) over HSW/BDW is observed. This makes intuitive sense because the solver is memory bandwidth bound, unlike most assembly operations which are compute bound, an benefit greatly from HBM. The inclusion of SIMD vectorization also increases the performance of the solver on KNL, which has wider vector processing units than HSW/BDW. Relative to prior work without vectorization, we have observed an additional 20-30% increase in performance with vectorization.

Figure 1f shows the total time taken for a time step (the sum of equation assembly time and linear equation solve time). It is quite interesting, and somewhat unfortunate, that the end-to-end time step timing ends up being very similar across all of the architectures. For HSW/BDW systems, we feel like the performance is good overall and nearing optimal performance for the hardware. This statement is fitting since traditional CPUs have been around for decades and we collectively have been writing our software to perform well on for these systems. For the newer hardware, the strengths of each end up being offset by the weaknesses. In the case of KNLs, excellent solver performance is offset by equation assembly time that is about equally slower than HSW/BDW. The net effect is that end-to-end time on KNL, for this particular problem, is not noticeably any better than HSW/BDW. We will note that a more solve-dominated problem will give better performance than HSW/BDW by taking advantage of the fast solves on KNL. SIMD vectorization of the interior assembly kernels could make a big difference in the overall performance on KNL. In the case of GPUs, the excellent interior flux evaluation performance is offset by poor scaling of very small kernels and poor MPI performance. Coupled with modest linear equation solve performance, the net effect is that end-to-end time on GPUs is modestly slower than the other architectures. Given a completely optimized GPU solver and better MPI performance, this architecture possibly has the most potential.

4.1.2 Weak Scaling

An assessment of weak scaling (fixed grid size per node or GPU) was performed on the GRV using a sequence of nested multi-block structured grids, ranging from ~ 2 million cells to ~ 2 billion cells. Each grid is an $8\times$ refinement of the previous grid in the sequence. A weak scaling analysis for V100 was not possible; it was not possible to run the weak scaling problem with 64 GPUs (our system only has 40 V100s) and we encountered technical difficulties in getting a single V100 case to run. A single V100 data point is shown for only for the sake of reference.

Figure 2a shows the boundary term evaluation weak scaling. This kernel clearly is not scaling well for any architecture but this kernel is quite small and overall insignificant.

Figure 2b shows the interior term evaluation weak scaling. Performance relative to each architecture shows

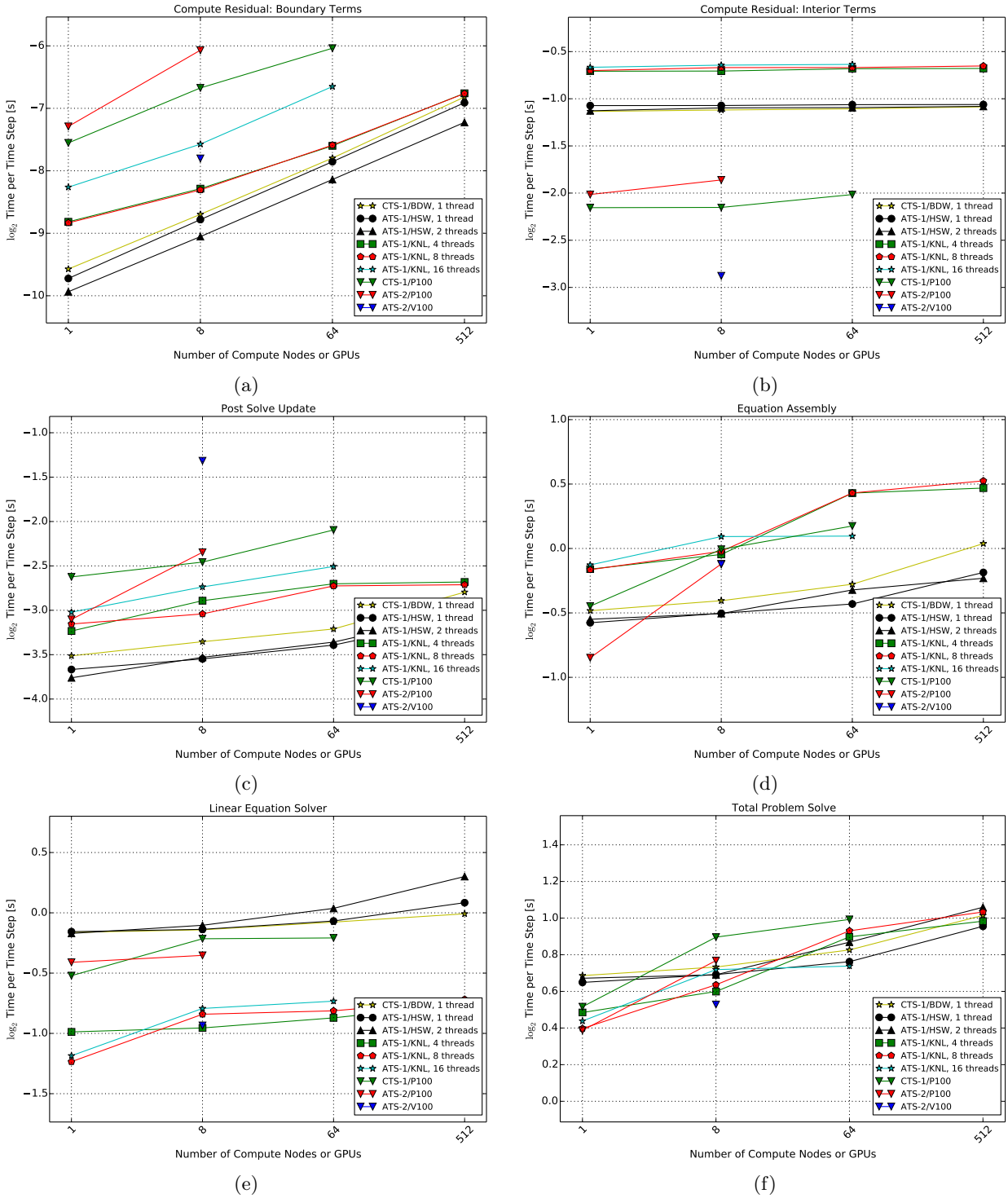

Figure 2: GRV, perfect gas model, implicit line solver, weak scaling

the same behavior as in the strong scaling, and the same rationale given there applies here. HSW/BDW and KNL weak scale very well while the GPU platforms show slightly less than ideal parallel efficiency.

Figure 2c shows the post solver kernels aggregate weak scaling. HSW/BDW performs and scales the best and the explanation given for the strong scaling of this operation is the same here.

Figure 2e shows the linear equation solve weak scaling. KNL performs best but the architecture that scales the best here is not evident.

Lastly, Figure 2f shows weak scaling for the end-to-end time step time. Similar to the conclusion given in for the strong scaling of this problem, all architectures perform about the same. Again, for KNL and GPU, the strengths and weaknesses roughly offset. It does appear as though the weak scaling parallel efficiency of HSW/BDW is the best, but bring this problem up to a much higher weak scaling limit is required before we can definitively make this assertion.

4.2 GRV - 5-species Air Model

This analysis focuses on the performance of the reacting gas model that has been implemented in `SPARC`. We use the 5-species air model discussed in Section 2.2 with a single energy equation and no turbulence model. While the machinery around mesh traversal and data layout is the same as with the perfect gas model, the computational kernels are considerably more complex.

For this study we reduce the number configurations for each architecture as follows: a) only one thread per rank for HSW/BDW (the previous study shows now significant impact of using two threads), b) 8 and 16 threads per rank for KNL (these configurations have proven to be the most performant for our code), and c) only P100 results (unfortunately, problems with the MPI installation prevented us from running this case on the V100 system).

4.2.1 Strong Scaling

A strong scaling analysis for the reacting gas model was conducted on the same 32 million cell multi-block structured grid as in Section 4.1.1. We begin with some preliminary comments for all of the plots.

First, for this problem, GPU memory becomes completely saturated at 16 devices. This has a very dramatic and adverse effect in each of the scaling curves. The reason for this is not entirely understood at this time.

Second, much of commentary given in Section 4.1.1 for the relative performance between architectures applies here. We will thus limit the commentary here to only the changes in performance and scaling behavior between the perfect and reacting gas models.

Figure 3b shows the strong scaling of the interior flux evaluation kernel. The difference between HSW/BDW and KNL is larger the in the perfect gas model, being $\sim 1.8\times$ slower for KNL. This kernel is compute-bound even more so than the perfect gas kernel and the slower cores on KNL are likely the reason for the slower performance. As previously mentioned, SIMD vectorization on KNL may improve the performance. On P100, this kernel is no faster than HSW/BDW, where as it was $\sim 2\times$ for the perfect gas model. This model is much more complex and requires the use of temporary variables/arrays. Temporary variables/arrays do not significantly affect performance on traditional CPUs or many-core CPUs but do have an affect on GPUs, which are more susceptible to register pressure. We attribute this to reduced performance of this model on P100. We expect performance on V100 to be much better than P100, as it was in Figure 1b, however we were unable to run the strong scaling for this case.

Figure 3e shows the solver performance for the GPU to be between that of HSW/BDW and KNL, as was case in Figure 1e. For the complete time step time, shown in Figure 3f, also we draw the same conclusion as we did for Figure 1f; the end-to-end time is roughly the same across architectures, owing to the fact that for KNL and GPU the strengths are almost equally offset by the weaknesses.

4.2.2 Weak Scaling

A weak scaling analysis of the reacting gas model was performed for multi-block structured grids ranging from ~ 2 million cells to ~ 2 billion cells. Figures 4a - 4f show very similar trends as in Figures 2a - 2f. The performance differences between the perfect gas model and the reacting gas model discussed in the previous section apply to the weak scaling of this problem. For these reasons, not much new commentary is

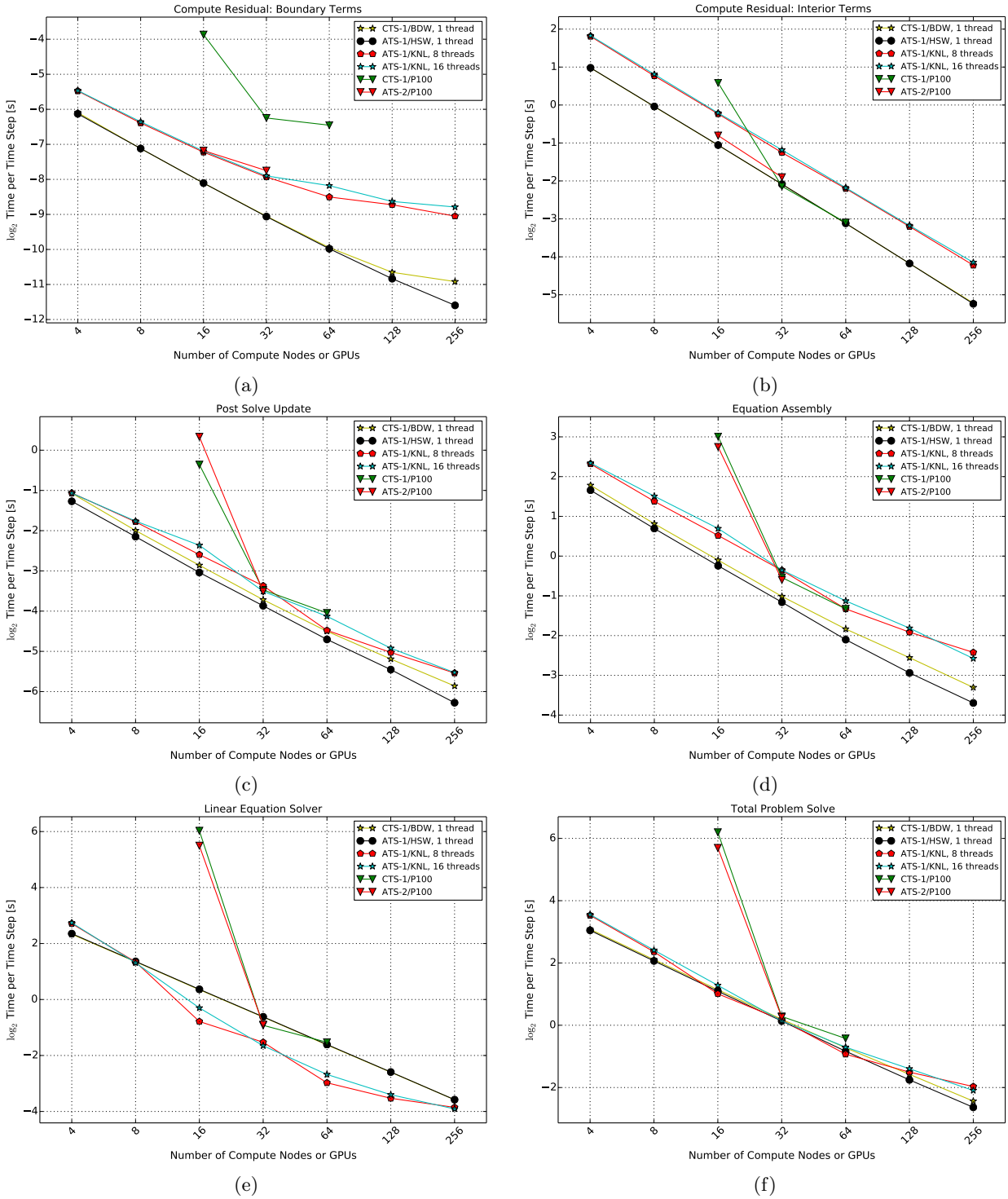

Figure 3: GRV, 5-species/1-temperature gas model, implicit tri-diagonal solver, strong scaling

warranted. We did get a single V100 run for the coarsest weak scaling grid with 4 GPUs, shown as single, blue triangle in the plots. Not much can be concluded from this sole data point, however for Figure 4b it seems to indicate the same observation from the perfect gas model analysis that V100 is $\sim 2\times$ faster than P100.

5 Conclusion

SPARC is being developed as a performance-portable compressible CFD code to do Sandia's aerodynamic and aerothermodynamic modeling and simulation work and to address the challenges posed by next-generation computing platforms. In this paper, we presented the software design patterns we have adopted to enable the code to run performantly on present day CPU-based systems (Intel's Xeon Haswell and Broadwell processors) as well as many-core architectures (Intel's Xeon Phi Knights Landing processors) and GPU architectures (Nvidia's Tesla P100 and V100). Multiple levels of parallelism are employed by SPARC; MPI is used for inter-node parallelism and MPI, threads, and SIMD vectorizations (to some degree) are used for on-node parallelism. Kokkos enables our on-node parallelism strategies to be portable among architectures. At this point in time, we do not claim to have optimized code for any given architecture and our effort on truly achieving performance portability is still a work in progress. However, we feel that what we have accomplished to date is promising. We have a single code base that is running reasonably efficiently on CPUs, KNLs and GPUs, with some notable highlights: 1) all significant computational kernels for the perfect and reacting gas models in SPARC have been written with Kokkos and are portable across all architectures, 2) linear equation solve performance is up to $2\times$ faster for threaded and vectorized KNL than for traditional CPU architectures and of comparable performance for GPU architectures, and 3) the most expensive equation assembly kernels are up to $2\times$ faster for P100 GPUs and up to $2\times$ faster for V100 GPUs than for traditional CPU and KNL architectures. Ongoing and future work focuses on the following: 1) use of SIMD vectorization to improve the performance of equation assembly and solve for KNL, 2) improvement and optimization of the linear equation solver for GPU systems, and 3) strategies for improved assembly on GPU systems.

Acknowledgments

The development of this code is a rather large effort at Sandia and there have been many contributors to its design and development. We would like to thank all of the contributors and collaborators, past and present, that have worked on this effort. The Kokkos team, including Christian Trott, for their development of a package the enables application code performance portability. The Application Performance Team for performance analysis support, including Simon Hammond, for many useful discussions regarding performance on next-generation computing platform and for the help and support of Sandia's testbed computing systems. Ross Bartlett, Joe Frye and Sam Browne for DevOps support and Trilinos building and installation across many computing platforms. And Matt Bopp, Brian Carnes, Jeff Fike, and Ross Wagnild, who round out the core SPARC development team.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

[1] Manuel R. Lopez-Morales, Jonathan Bull, Jacob Crabill, Thomas D. Economon, David Manosalvas, Joshua Romero, Abhishek Sheshadri, Jerry E. Watkins II, David M. Williams, Francisco Palacios, and Antony Jameson. Verification and Validation of HiFiLES: a High-Order LES Unstructured Solver on Multi-GPU Platforms. In *Proceedings of the 32nd AIAA Applied Aerodynamics Conference*, 16–20 June 2014.

[2] Dylan Jude and James Baeder. Extending a Three-Dimensional GPU RANS Solver for Unsteady Grid Motion and Free-Wake Coupling. In *Proceedings of the 54th AIAA Aerospace Sciences Meeting*, 4–8 January 2016.

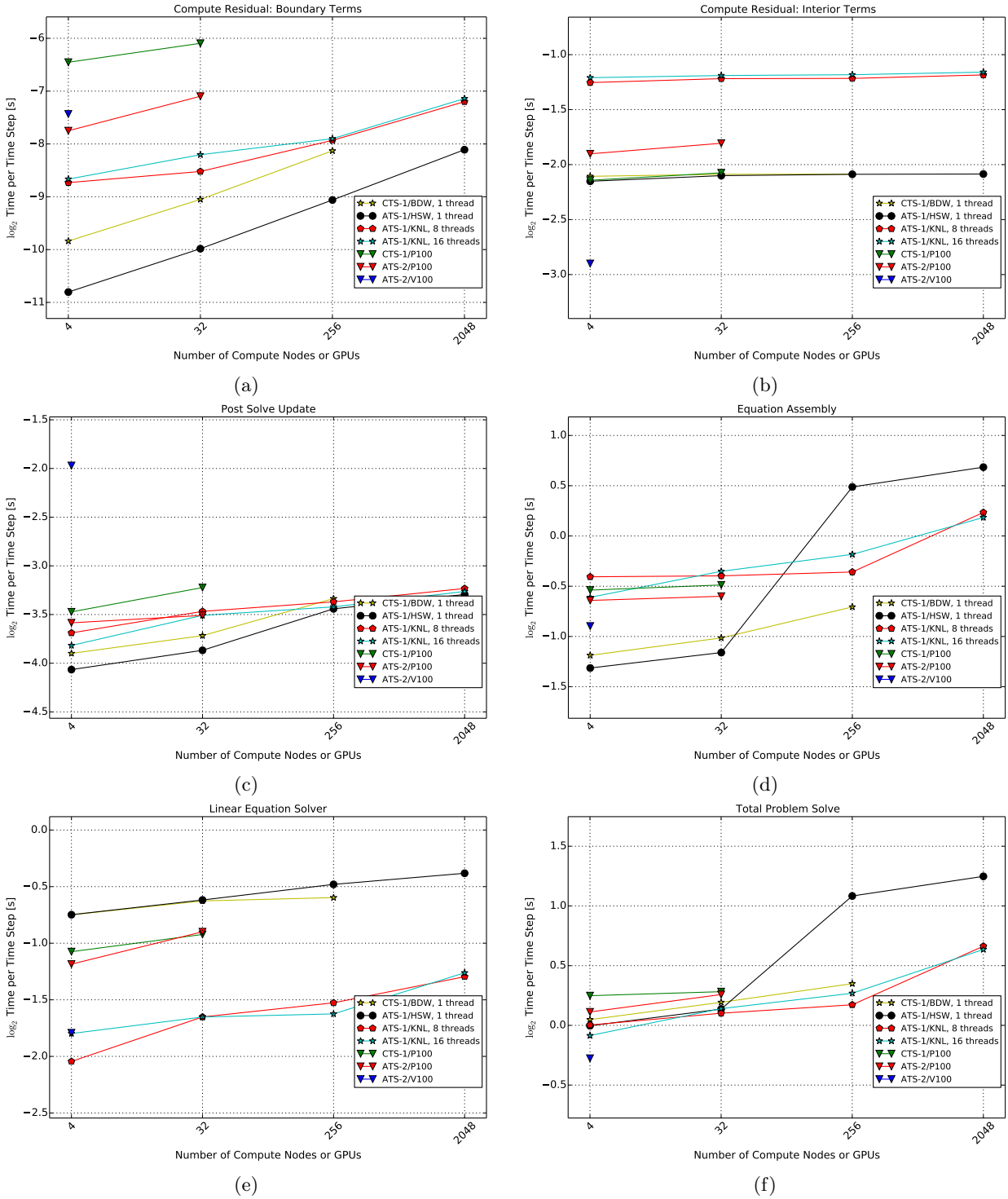

Figure 4: GRV, 5-species/1-temperature gas model, implicit tri-diagonal solver, weak scaling

[3] Jialin Lou, Yidong Xia, Lixiang Luo, Hong Luo, Jack Edwards, and Frank Mueller. OpenACC Directive-based GPU Acceleration of an Implicit Reconstruction Discontinuous Galerkin Method for Compressible Flows on 3D Unstructured Grids. In *Proceedings of the 54th AIAA Aerospace Sciences Meeting*, 4–8 January 2016.

[4] Jerry Watkins, Joshua Romero, and Antony Jameson. Multi-GPU, Implicit Time Stepping for High-order Methods on Unstructured Grids. In *Proceedings of the 46th AIAA Fluid Dynamics Conference*, 13–17 June 2016.

[5] Thomas D. Economon, Francisco Palacios, Juan J. Alonso, Gaurav Bansal, Dheevatsa Mudigere, Alexander Heinecke, and Mikhail Smelyanskiy. Towards High-Performance Optimization of the Unstructured Open-Source SU2 Suite. In *Proceedings of AIAA Infotech @ Aerospace*, 5–9 January 2015.

[6] Yi Lu, Kai Liu, and W.N. Dawes. Efficient and Affordable High Order, High Fidelity Large Eddy Simulations for Industrial Level Problems. In *Proceedings of 55th AIAA Aerospace Sciences Meeting*, 9–13 January 2017.

[7] Kenneth J. Franko, Travis C. Fisher, Paul T. Lin, and Steven W. Bova. CFD for Next Generation Hardware: Experiences with Proxy Applications. In *Proceedings of the 22nd AIAA Computational Fluid Dynamics Conference*, 22-26 June 2015.

[8] D. Curran, C.B. Allen, and S. McIntosh-Smith. Towards Portability for Structured Grid CFD Codes. In *Proceedings of the 54th AIAA Aerospace Sciences Meeting*, 4–8 January 2016.

[9] B. Carnes, V. G. Weirs, and T. Smith. Code Verification and Numerical Error Estimation for Use in Model Validation of Laminar, Hypersonic Double-Cone Flows. In *Submitted to AIAA SciTech*, 2019.

[10] B. A. Freno and B. R. Carnes. An Overview of Preliminary Code Verification Efforts for the Sandia Parallel Aerodynamics and Reentry Code (SPARC). In *Submitted to AIAA SciTech*, 2019.

[11] S. Kieweg, J. Ray, V. G. Weirs, B. Carnes, D. Dinzl, B. Freno, M. Howard, W. J. Rider, and T. Smith. Uncertainty Quantification, Sensitivity Analysis, and Validation Assessment of Laminar Hypersonic Double-Cone Flow Simulations. In *Submitted to AIAA SciTech*, 2019.

[12] J. Ray, S. Kieweg, D. Dinzl, B. Carnes, V. G. Weirs, B. Freno, D. Dinzl, M. Howard, and T. Smith. Estimation of Inflow Uncertainties in Laminar Hypersonic Double-Cone Experiments. In *Submitted to AIAA SciTech*, 2019.

[13] Christopher G Baker, Michael A Heroux, H Carter Edwards, and Alan B Williams. A Light-weight API for Portable Multicore Programming. In *2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 601–606. IEEE, 2010.

[14] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns. *J. of Parallel and Distr. Comp.*, 74:3202–3216, 2014.

[15] Michael A Heroux and James M Willenbring. A New Overview of the Trilinos Project. *Scientific Programming*, 20(2):83–88, 2012.

[16] Micah Howard, Andrew M. Bradley, Steven W. Bova, James R. Overfelt, Ross M. Wagnild, Derek D. Dinzl, Mark F. Hoemmen, and Alicia M. Klinvex. Towards a Performance Portable Compressible CFD Code. In *Proceedings of the 23rd AIAA Computational Fluid Dynamics Conference*, 5–9 June 2017.

[17] Andrey Prokopenko, Christopher Siefert, Jonathan J Hu, Mark Frederick Hoemmen, and Alicia Marie Klinvex. Ifpack2 User's Guide 1.0. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States); Sandia National Laboratories, Livermore, CA, 2016.

[18] *KokkosKernels*, https://github.com/kokkos/kokkos-kernels.

[19] Kyungjoo Kim, Timothy B Cost, Mehmet Deveci, Andrew M Bradley, Simon D Hammond, Murat E Guney, Sarah Knepper, Shane Story, and Sivasankaran Rajamanickam. Designing Vector-Friendly Compact BLAS and LAPACK Kernels. In *Supercomputing 2017, Denver, CO*. ACM, 2017.

[20] *Sandia to Install First Petascale Supercomputer Powered by ARM Processors*, https://www.top500.org/news/sandia-to-install-first-petascale-supercomputer-powered-by-arm-processors.

[21] Mahesh Rajan, Doug Doerfler, Mike Tupek, and Simon Hammond. An Investigation of Compiler Vectorization on Current and Next-generation Intel Processors Using Benchmarks and Sandia's Sierra Applications. *Cray User Group Meeting*, 2015.