Chapter 3
Turbulent Combustion Simulations with High-Performance Computing

Hemanth Kolla and Jacqueline H. Chen

Abstract Considering that simulations of turbulent combustion are computationally expensive, this chapter takes a decidedly different perspective, that of high-performance computing (HPC). The cost scaling arguments of non-reacting turbulence simulations are revisited and it is shown that the cost scaling for reacting flows is much more stringent for comparable conditions, making parallel computing and HPC indispensable. Hardware abstractions of typical parallel supercomputers are presented which show that for design of an efficient and optimal program, it is essential to exploit both *distributed memory parallelism* and *shared-memory parallelism*, i.e. *hierarchical parallelism*. Principles of efficient programming at various levels of parallelism are illustrated using archetypal code examples. The vast array of numerical methods, particularly schemes for spatial and temporal discretization, are examined in terms of tradeoffs they present from an HPC perspective. Aspects of data analytics that invariably result from large feature-rich data sets generated by combustion simulations are covered briefly.

Keywords Direct numerical simulation · High performance computing · Parallel computing · Hierarchical parallelism

3.1 Introductory Remarks

Arguably, the *raison d'être* of turbulent combustion simulations has been to inform the design of energy conversion devices, predominantly those in the power generation and transportation sectors (reciprocating and air-breathing engines). While combustion simulations have addressed alternative applications (e.g. atmospheric chemistry, chemical engineering and manufacturing processes), the overwhelming

H. Kolla (✉)
Scalable Modeling and Analysis, Sandia National Laboratories, Livermore, CA, USA
e-mail: hnkolla@sandia.gov

J. H. Chen
Reacting Flow Research, Sandia National Laboratories, Livermore, CA, USA
e-mail: jhchen@sandia.gov

S. De et al. (eds.), *Modeling and Simulation of Turbulent Combustion*, Energy, Environment, and Sustainability, https://doi.org/10.1007/978-981-10-7410-3_3

majority of simulation efforts have been devoted to this endeavour due to the obvious technological incentives and imperatives. Given such a scenario, a question that arises at the outset is

What is the role of direct numerical simulations (DNS) of turbulent combustion?

This question is pertinent since the largest, state-of-art, DNS to date are still in domains that are orders of magnitude smaller in scale (device size) or conditions (Reynolds, Damköhler numbers) compared to real devices. The answer to this question establishes the scope and ambition of combustion DNS and, more importantly, its envelope of feasibility.

This chapter is structured in the hope that the reader will arrive, step-by-step, at a comprehensive and nuanced answer to this question. The first section will discuss, using simple dimensional scaling arguments, the computational cost of combustion DNS. The next section will cover aspects of parallel high-performance computing (HPC) and the choices available to exploit multiple levels of parallelism in a typical supercomputer. These choices are invariably interlinked to the governing equations and numerical aspects, which can guide the design of a high-fidelity combustion code which will be the focus of the following section. The final section will cover aspects of data analytics which are essential in making sense of the large volumes of data generated by combustion DNS.

3.2 Computational Cost of Combustion DNS

It is perhaps prudent to establish what the term "direct numerical simulations" means in the context of turbulent combustion. DNS, as has come to be accepted by convention, may best be described

a simulation methodology where all the relevant scales in the continuum regime are sufficiently resolved on the computational grid.

Note the carefully worded qualifiers. The scales that are resolved are in the 'continuum regime' and not anything smaller such as, for instance, those described by the kinetic theory of gases (e.g. mean free paths). Note also that the scales being resolved on the computational grid implies that the spatial grid resolution is smaller than the smallest (continuum scales) of motion. These thumb rules for what qualifies as DNS can be attributed to early simulations of non-reacting turbulent flow when DNS was pioneered as a means to understand and quantify multiscale statistics of turbulence. For reactive flows, the definition of DNS extends to all scales, or more aptly spatial gradients, relevant to fluid motion (velocity) and reacting scalars (temperature and species mass fractions). It is worth pointing out that many practical devices of interest have flows that defy such an easy characterization. The most prominent examples, relevant to combustion devices, are turbulent spray flames and flames with soot. Mutliphase turbulent flows, whether reacting or non-reacting, introduce scales of interaction between the liquid and gas phases that are prohibitively expensive to resolve. Likewise, the physical processes governing soot straddle the continuum regime. Soot formation occurs at molecular scales with soot precursors understood to be a few aromatic molecules, while they can grow, by agglomeration, to sizes that are larger than the smallest continuum flow scales (Raman and Fox 2016).

With this context, let us revisit some scaling arguments that determine the computational cost of DNS. The cost scaling is simply determined by the ratio of the largest to smallest spatio-temporal scales and the requirement that the computational domain be at least as large as the largest length scale, the so-called integral length scale Λ, while the grid resolution has to be at least as small as the smallest length scale, the Kolmogorov length scale η. This ratio dictates the minimum number of grid points required per spatial dimension, N, which scales with the Reynolds number, Re, as

$$N \sim \frac{\Lambda}{\eta} \sim \mathrm{Re}^{3/4}. \tag{3.1}$$

The total number of grid points for a three-dimensional simulation thus scales as

$$N^3 \sim \mathrm{Re}^{9/4}. \tag{3.2}$$

Combustion exacerbates the cost scaling in two principal ways. First, chemical reactions introduce spatial scales which, under most conditions of practical interest, are finer than turbulent flow scales. Consider, for illustrative purposes, the somewhat benign conditions of an atmospheric pressure methane–air turbulent flame at a Reynolds number of 10,000 and an integral length scale of 0.1 m (10 cm). The Kolmogorov scale $\eta = \Lambda/\mathrm{Re}^{3/4} = 10^{-4}$ m. A reasonable estimate for a chemical length scale, δ, is the ratio of thermal diffusivity to kinematic viscosity which, under these conditions, is $\delta \sim 10^{-5}$ m. The grid points requirement must now be revised to account for the fact that grid resolution must be smaller than δ,

$$N \sim \frac{\Lambda}{\eta}\frac{\eta}{\delta} \implies N^3 \sim \mathrm{Re}^{9/4}\left(\frac{\eta}{\delta}\right)^3. \tag{3.3}$$

Second, combustion considerably increases the number of solution variables and associated number of partial differential equations (PDEs). For non-reacting turbulent flows, the solution state comprises five variables: three components of velocity and any two of energy/enthalpy, pressure and density, and these are obtained as solutions of five PDEs governing conservation of mass, momentum and energy. The thermodynamic equation of state provides the additional constraint. For turbulent reacting flows, the solution state must be expanded to include details of the chemical composition in the form of species mass fractions, Y_i. For the simplest fuel, hydrogen, nine species sufficiently describe the oxidation mechanism (Burke et al. 2012), whereas for hydrocarbons this number is much larger. For the simplest hydrocarbon, methane, the mechanisms can be, depending on the conditions, as small as involving 13 species (Sankaran et al. 2007) or as large as 53 species (GRI-Mech 2017). Hence, for the methane–air turbulent flame example considered above, for the widely used GRI-Mech mechanism, the additional cost factor due to chemical reactions, assuming the cost scales linearly with number of PDEs, is

$$\left(\frac{\eta}{\delta}\right)^3 \times \frac{53+5}{5} \approx 1.16 \times 10^4. \tag{3.4}$$

These cost considerations dictate that turbulent combustion DNS is only feasible with high-performance parallel computing. For the same reasons, combustion DNS have, up to now, only been possible for small ‘postage-stamp-sized’ computational domains and are unable to approach sizes anywhere close to real devices. Only in recent years, thanks to the exponential growth in computing power, have DNS begun to approach laboratory-scale flames. Figure 3.1 shows a historical trend of DNS simulations performed with the Sandia code S3D (Chen et al. 2009) for five representative simulations. Plotted on the y-axis is the computational problem size in logarithmic scale, defined as the product of the total number of grid points and the number of solution variables, and the x-axis shows the year when each simulation was performed. The figure shows a trend where the computational problem size has increased exponentially with time, indicating that state-of-art DNS simulations have kept pace with the increase in computational power. In spite of requiring such massive computational resources, the largest DNS simulations have only been able to approach Re $\sim O(10^4)$, whereas the Re values in practical devices are at least an order of magnitude higher, suggesting that DNS is still quite some way away from approaching real device conditions.

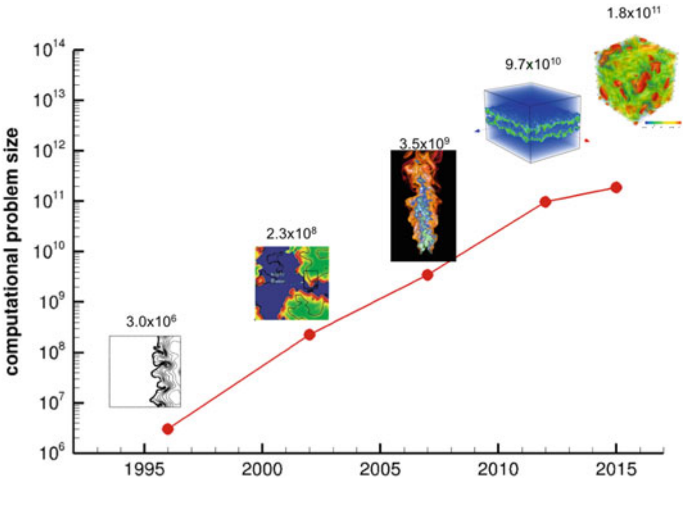

Fig. 3.1 Historical trend showing five representative simulations performed with the DNS code S3D over the last 20 years. The computational problem size is shown on a logarithmic scale on the Y-axis and the year corresponding to each simulation is on the X-axis. From left to right, the simulations correspond to **a** 2D turbulent premixed methane–air flame (Echekki and Chen 1996), **b** 2D turbulent auto-igniting stratified hydrogen–air flame (Echekki and Chen 2002), **c** 3D rectangular slot-jet turbulent Bunsen premixed methane–air flame (Sankaran et al. 2007), **d** 3D temporally evolving rectangular jet turbulent premixed hydrogen–air flame, and **e** a 3D reactivity controlled compression ignition (RCCI) flame of primary reference fuel (Treichler et al. 2018)

3.3 HPC and Hierarchical Parallelism

Having established that turbulent combustion simulations are infeasible without large computational resources, it is necessary to examine the intersection of HPC and CFD simulations. In particular, it is critical to appreciate that achieving efficiency in HPC and parallel computing requires, unfortunately, that one pay attention to numerous aspects of modern parallel computers. In an ideal world, principles governing parallel computing are simple, flexible, robust and mature enough that a programmer requires little or no knowledge of the hardware or software architecture details to write an efficient program, allowing him/her to focus efforts on the algorithmic and physics aspects of the computer program. In reality, however, the situation is the opposite and writing an efficient and performant parallel program involves a careful structuring and organization of the code within the confines and constraints imposed by the system. The payoff, or conversely the penalty, is that a carefully optimized code can often be orders of magnitude faster than a naively written one. It is also worth pointing out that even LES and URANS simulations are tackling problems large enough as to require HPC resources and the principles outlined in this section apply to these as well.

Just by considering an abstract model of typical parallel supercomputer, it becomes apparent that a program has to expose various levels of parallelism, i.e. *hierarchical parallelism*. Without being simplistic, at a high level, a parallel computer can be thought of as a set of inter-connected computing nodes that can communicate with each other through a network. Each node comprises some memory that is private to it and some computing elements that share this memory. Accessing data from the memory of another node requires communicating via the network. This picture establishes the concepts of *distributed* versus *shared-memory parallelism*.[1] The detail to bear in mind is that it is much quicker and more efficient to access shared node-local memory than accessing the memory of another node via network communication.

3.3.1 Distributed Memory Parallelism

At the highest level, utilizing a set of nodes requires decomposing a computational problem into subsets that can each be assigned to a different node. Each node can make progress on its respective subset, communicating with other node(s) to share data as and when necessary. The chief principles guiding the design for distributed memory parallelism are as follows:

1. **Balanced computational load**: It is desirable to decompose the problem such that the computational load is divided as equitably as possible among the nodes.

[1]This distinction is also referred to sometimes as internode versus intra-node or node-level parallelism.

If the load is not balanced, the progress on the overall problem might be limited by the node(s) with the largest load.

2. **Maximize computation-to-communication ratio**: It is desirable to decompose the problem such that the computational load associated with the subset on each node is much larger than the amount of communication that need be performed with other nodes.
3. **Minimize global communication**: It is desirable to devise the algorithms such that the need for global communication, i.e. data aggregation involving all nodes, is minimized or even eliminated if possible.

These principles reflect the constraint that network communication is an expensive proposition and both the number of messages and their sizes must be reduced to the extent possible.

For illustration, consider a general prototype 1D PDE of form

$$\dot{u} = F(u', u''), \tag{3.5}$$

where the dot denotes a time derivative and the single and double primes denote first and second spatial derivatives, respectively. This PDE, with suitable initial and boundary conditions, is to be solved on a discretised 1D computational domain of N grid points. Considering the computational cost arises mainly due to N being large, the most natural choice for decomposing this problem is to divide the grid points among the available nodes (P), an approach commonly referred to as *domain decomposition*. Each node is then responsible for solving the equation over N/P grid points, as shown schematically in Fig. 3.2. The computational work at each grid point involves evaluating the spatial derivatives and assembling them to advance the solution in time. The spatial derivative evaluation will necessarily require information from neighbouring grid points and for grid points at the edges of each subset N/P, these grid points reside on the adjacent node, which have to be exchanged by internode communication. The grid points that need to be exchanged are referred to as *ghost points*, and the number of ghost points and the direction of internode exchange are stipulated by the specifics of the spatial discretization scheme such as

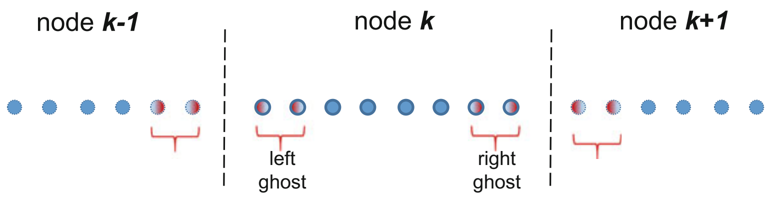

Fig. 3.2 Schematic showing a typical domain decomposition of a 1D computational domain. The dashed line denotes node boundaries and each node is assigned a subset of the computational grid points. The grid points at the edge of the node boundaries, ghost points, need to be communicated. On node k, the left ghost points are required by node $k-1$ while the right ghost points by $k-1$. Conversely, node k requires the right ghost points of $k-1$ and left ghost points of $k+1$

the stencil width and whether it is central or one-sided differencing. The principles listed above translate to the following:

- If the same computations are performed at each grid point, balanced computational load translates to the condition that N is exactly divisible by P. If it is not, then some nodes end up with a larger subset of the domain, and greater load, than others.
- Maximizing computation-to-communication ratio translates to the condition that the ratio of the number of ghost points to the number N/P be minimized. In the example depicted in Fig. 3.2, $N/P = 8$ while the number of ghost points on each node is 4 (2 at either edges). It is easy to conceive of a situation where the number of ghost points is greater than N/P, which is undesirable since each node might have to communicate with more than one adjacent node in each direction, increasing the number of communication messages.
- Minimizing global communication translates to the condition that for advancing the solution on its share of grid points each node needs to communicate only with a small subset of the other nodes, and not all.

An example of an algorithm that requires global communication arises in early DNS codes which transformed the Navier–Stokes equations from physical space to the wavenumber space and solved for the Fourier modes of the velocity components. The nonlinear convective term poses a problem since it represents a multiplication in the physical space and hence a convolution in the wavenumber space, which requires an integral over all the Fourier modes, or information over all the grid points.

In many circumstances, it becomes possible to optimize beyond these three principles and, in particular, for each node to completely hide the apparent cost of, or delay due to, communication with a fairly simple rearranging of the computational work. For node k in Fig. 3.2, it is possible to further distinguish the grid points as *interior* (blue) points versus the ghost points (blue-red). Recognizing that computations for the interior points require data that is completely local to node k, an algorithm could start the computations on the interior points without waiting on the messages from the neighbouring nodes to arrive. If N/P is large enough relative to the number of ghost points, these computations take longer than the time it takes for the messages to arrive and the computations on the ghost points can begin as soon as those on the interior points are completed. With such a rearrangement each node is kept busy doing computational work all the time. This principle is generally referred to as *hiding communication latency* and is an aspirational goal of all distributed networked systems, of which parallel computers are just one example. However, from a programmer's perspective, designing for these principles requires a protocol that enables all types of internode communications and, more importantly, a robust implementation of the protocol which can be accessed from within common programming languages (e.g. C, C++ and Fortran). The most popular and widely used communication protocol for parallel scientific computing is the *Message Passing Interface* (MPI) (Gropp et al. 1994). MPI is, strictly speaking, a community standard (MPI Forum 2017) for an interface that provides a communication protocol for parallel computing, but it is not the only one. It just is the most successful standard and has become synonymous with parallel computing and is supported by all major vendors of supercomputers.

3.3.2 Node-Level Parallelism

In some respects, distributed memory parallelism is easy to reason about and design for since the basic architecture of distributed supercomputers at the system level have changed very gradually over the decades. On the other hand, node architectures have been continuously and rapidly changing, making node-level parallelism much harder to expose in a program. Recall that we defined a computational node, rather loosely, as an entity containing some private memory which is shared among certain computational elements. This definition is simplistic and in reality a typical computational node has a rather complex *memory hierarchy*, ranging from *main memory* (also known as *random access memory* or RAM), *cache memory* and *processor registers*. The computational element is typically a processor core that performs the computational operations on data accessed from said memory hierarchy. Registers are smallest in capacity (often large enough to hold only a handful of numbers), closest to the core and fastest to access, whereas main memory is largest in capacity, farthest from the core and the slowest to access. Node architectures have gradually progressed in complexity from containing single-core CPUs, multicore (10s) CPUs, multiple multicore CPUs, to a hybrid multicore CPU together with many-core (100s) CPUs and GPUs. The memory hierarchy, complexity and management have also increased with the growing number of units on each node, although the memory-to-core ratio is decreasing. This makes optimizing for node-level parallelism arduous but necessary.

The performance of a program in a shared-memory environment hinges almost entirely on optimal utilization of the memory hierarchy, so some context is necessary. The purpose of main memory is to provide storage for the entire duration of a running program. Hence, it is usually large, but accessing it is time-consuming (compared to the speed with which arithmetic operations can be performed by the processor). This is ameliorated by cache memory whose purpose is to provide storage for frequently accessed variables by a program. Cache memory is comparatively much faster to access but has limited capacity which is usually not large enough to provide storage for an entire program.[2] In that regard, cache memory acts as an intermediate buffer, but the cache access patterns of a program have an outsized impact on its performance. And finally registers are the memory locations that hold the actual variables feeding the arithmetic operations of the processor and the results. As a consequence, the design of an efficient program must pay careful attention to the following:

[2]Early computer architectures had just a single level of cache between the processor and main memory. However, with the number of processors and their speed, increasing dramatically the current architecture has multiple (upto 3) cache levels. Furthermore, in multicore architectures, some levels of caches are not shared amongst all cores on a node, but subsets of them.

- **Data storage and layout in memory**: Most scientific codes operate on arrays of numbers. Practically, in all programming languages, these arrays are stored in contiguous memory.[3] However, multidimensional arrays are also stored as though they were 1D arrays (vectors) by stretching out the dimensions successively. The scheme depends on whether the arrays are stored in row- or column-major order which can be different for different programming languages, e.g. Fortran is column-major order, while C/C++ are row-major order.
- **Memory access**: Since most scientific programs spend a bulk of the time performing operations on loops over arrays, knowledge of data layout in memory can be leveraged to significantly improve their performance. This arises from a concept known as *locality of reference* (Denning 2005) which establishes that when a memory location, an element `i` of a multidimensional array, is accessed (referenced) in a program, it is highly likely that it will be accessed again in the near future (temporal locality) and highly likely that nearby memory locations—array elements close to `i`—will be accessed in the near future (spatial locality). Hence, aligning the access pattern of the array elements in the program with the underlying memory layout significantly improves performance.

These considerations inform the principles for designing a program for optimal performance. The principles, which are not necessarily mutually exclusive and sometimes even conflicting, can be organized along the various levels of the memory hierarchy as described below. This discussion is kept simple so as to be accessible to someone without a background in computer science. An interested reader will readily find, upon even a cursory search, a wealth of articles that expound these topics in greater depth and nuance.

3.3.2.1 Vectorization

Vectorization pertains to the fact that modern processor architectures are primed to execute an instruction on multiple data elements (vectors) very efficiently, a concept known as *Single Instruction Multiple Data* (SIMD). Consider an example Fortran code that adds two arrays `a` and `b`, and stores the results in array `c`:

```fortran
do i = 1,N
   c(i) = a(i) + b(i)
enddo
```

Behind the scenes, this code is transformed by the compiler to a set of instructions on the processor that comprise a sequence involving reading the elements of `a` and `b` from memory, performing the addition operation and storing the result element `c` in memory. This sequence can be sped up significantly if it is performed on blocks

[3]This is a bit of an illusion. What a program addresses is not directly the physical location of memory but what is known as virtual memory. All arrays are stored contiguously in virtual memory, i.e. the memory addresses of successive elements of an array are contiguous in the virtual address space. The translation of the virtual address space to physical memory addresses is handled by memory management layer of an operating system.

of the arrays being operated on, rather than one element at a time. This is enabled by a combination of processor registers that are large enough to hold multiple data elements and processor instruction sets that support simultaneous execution of the instruction on all elements held in the register.

Modern compiler technology is sophisticated enough that if the loops are straightforward to vectorize, and the corresponding instruction set is supported on a given architecture, compilers can automatically vectorize relevant portions of the code.[4] However, even simple missteps can prevent loops from being vectorized, resulting in significant performance penalty. Consider, instead, the case of adding 2D arrays in the following example:

```
do i = 1,N
  do j = 1,M
    c(i,j) = a(i,j) + b(i,j)
  enddo
enddo
```

Because of the way Fortran arrays are stored in memory, array elements along the first index `i` will be contiguous. However, the inner loop in this example is over the index `j` and successive elements of `j`, not being contiguous in memory, do not constitute blocks of vectors that can be fetched from memory efficiently. A compiler will attempt and fail to vectorize this code. Just by reordering the loops, making the `j` loop as the outer and `i` loop as the inner, the code can be vectorized without affecting correctness. There are other common pitfalls, e.g. placing conditional statements (e.g. `if`, `while`) inside the nested loops, introducing data dependencies on successive elements of the vectors, etc., that prevent a straightforward vectorization. The Intel compiler user guide has a very useful and explanatory page on tips for writing vectorizable codes (Programming Guidelines for Vectorization 2017). Vectorization as a programming practice was pioneered in the early decades of parallel computing (1970s–1990s) when the architecture of supercomputers was dominated by *vector processors*. In subsequent decades, with the changing complexion of computing and the rise of personal desktop computers, vector processors gave way to *scalar processors* and vectorization was not as critical for performance. The architectures are changing once again and with increasing number of computing elements on a given node, it has become necessary once again to optimize for data parallelism through vectorization.

[4]The Intel compiler suite has a very useful compiler option that generates a detailed vectorization report. When turned on, it generates output to a file that lists every portion of the code that was attempted to be vectorized by the compiler, whether the vectorization was successful and what, if anything, prevented a loop from being vectorized.

3.3.2.2 Cache Utilization

As described earlier, caches are memory layers that provide fast access of data to the processor cores to operate on. Most scientific computing applications, and certainly CFD codes, suffer from having low *computational intensity* which is defined as the ratio of number of arithmetic operations performed per units of memory accessed per operation. This would not be an issue if *memory bandwidth*—the rate at which memory can be accessed or transferred—outpaces the rate of performing operations, i.e. FLOPS. Unfortunately, the opposite is true and the performance of virtually all scientific codes is memory bandwidth limited. The purpose of cache is to relieve the bandwidth pressure on main memory by providing an intermediate location that stores the variables frequently accessed by a program. Accordingly, memory access patterns within a program can benefit greatly by optimizing the utilization of cache resulting in improved performance.

Consider a simple example of performing an outer product of two vectors a and b, of lengths N and M, respectively, resulting in an N × M matrix c:

```
do j = 1,M
  do i = 1,N
    c(i,j) = a(i) * b(j)
  enddo
enddo
```

At the point of execution of these loops, all the elements for the innermost-i-loop will be fetched from main memory to cache. Even accounting for the fact that these elements are contiguous in memory, if N is too large, the cache might not be large enough to hold all the elements, resulting in poor *cache reuse*. Each block of a that fits in cache will be fetched from main memory, operated on and purged before moving on to the next block.

Accordingly, optimizing the code for cache reuse requires a modification using a concept called *cache blocking*. Effectively, reorganizing loops and breaking them down further into blocks large enough to fit in cache, and reusing the block sized data as much as possible while it resides in cache, pays dividends. In the above example, let us assume that it is known that the cache is large enough to hold B elements of a vector. With this knowledge, rearranging the loops and having the inner most loop span blocks of size B, as shown below,

```
do j = 1, M, B
  do i = 1, N
    do jj = j, j+B-1
      c(i,jj) = a(i) * b(jj)
    enddo
  enddo
enddo
```

ensures that (1) the chunk of vector b accessed in the inner most jj loop fits in cache and (2) this chunk is reused for each evaluation of the i loop, increasing cache reuse. A careful reader will notice in the above example that, in the process of reordering nested loops for cache blocking, we have reintroduced an inefficiency. The innermost loop index `jj` is not the fastest varying dimension for the matrix `c` and the above code will result in *cache write misses*. A reordering of the loops that respects both contiguous memory access and does cache blocking for this example can be performed as follows:

```
do i = 1, N, B
  do j = 1, M
    do ii = i, i+B-1
      c(ii,j) = a(ii) * b(j)
    enddo
  enddo
enddo
```

This is a somewhat simplistic example illustrating the principle of cache blocking. For operations on multidimensional arrays, e.g. matrix–matrix multiplication, cache blocking can be done in one or two dimensions and the code with reordered loops becomes considerably larger (and less easy to read).

3.3.2.3 Shared-Memory Multiprocessing

At the highest level of a node virtually all computing platforms, and even most desktops, have multicore processors (and often multiples of them). While vectorization and cache utilization can be seen as optimizations necessary at a per-core level, at the node level it becomes necessary to fully utilize the multiple cores for good performance. Usually, multicore architectures have multiple cache levels, with at least one level of cache that is closest to each individual core, and at least one level that is shared by the multiple cores.[5] This allows a programmer to divide computational work among the multiple cores and have them execute in parallel.

By far the most common programming model for shared-memory multiprocess parallelism, and certainly the most accessible for common programming languages, is *Open Multiprocessing* (OpenMP) (Open Multi-Processing 2017). OpenMP provides a simple and easy way to extract parallelism by allowing a programmer to view the multiple cores as a set of *threads* that can each work on independent subsets of a problem concurrently. It extends the concept of *multi-threading* to multicore architectures and provides a core set of constructs that enable the creation of multiple threads, specifying a block of work for each thread, providing access to data variables which may be private to each thread or shared amongst the threads and synchronizing threads. To illustrate the use of OpenMP, consider the earlier example of

[5]The term *multiprocessing* is itself very general, simply referring to a system with multiple processors. The multicore nodes commonly found today may be thought to belong a subset known as *symmetric multiprocessors* (SMP) which strictly means that all the processors share all the memory and I/O resources equally and are orchestrated by one instance of an operating system kernel. The reality may be somewhere in between. Most modern node architectures have multiple sets of multicore CPUs, and they are not all exactly equal since at least one or more layers of memory hierarchy are not equally shared by all the cores. They are better described by a category known as *non-uniform memory access* (NUMA) nodes.

adding two arrays. This example code can be parallelized using OpenMP *directives* as follows:

```
!$OMP PARALLEL DO
do i = 1,N
   c(i) = a(i) + b(i)
enddo
!$OMP END PARALLEL DO
```

To speed up this portion of the code, one compiles the program by providing a compiler flag that ensures the spawning of multiple threads, launching each one on a separate processor at runtime. Each thread/processor executes this loop on the subset of the `i` index range assigned to it, thereby speeding up the program.

While this simple example illustrates the principle, the speedup achievable by such a multi-threading model is often hampered by other aspects. Behind the scenes, OpenMP follows a fork-join model. Typically, a program has a single master thread running which then spawns multiple worker threads upon encountering the directive `OMP PARALLEL DO`. The worker threads execute the block of code that follows and are destroyed at the directive `OMP END PARALLEL DO` with control passing back to the single master thread from thereon. There is a cost associated with the spawning and destroying of multiple threads by the master thread and this cost may not outweigh the benefits if the portion of the multi-threaded code is too lightweight. Moreover, the speedup of an overall larger program might still be limited by the portions of it that cannot be parallelized over multiple threads (so-called serial portion of a code), a phenomenon known as *Amdahl's law*. Also, OpenMP provides constructs for having variables shared amongst all threads versus keeping them private to each thread. The lifetime of a variable is clearly defined by such attributes and ignoring these rules can easily lead to erroneous code. In the above example, the subsets of the arrays `a`, `b` and `c` which fall within the index range for each thread are private to it and no other threads read/modify them. If a variable needs to be accessed by all threads, e.g. a global constant, then it needs to be declared as a shared variable and multiple copies of it are made, one for each thread. If shared variables are created indiscriminately, then this can cause an undue increase in memory footprint. At the same time, one needs to be careful about modifying the values of shared variables inside the multi-threaded portion of the code. Since each thread runs in parallel, the exact order in which the threads finish is undetermined. As a result, if more than one thread modifies the value of a shared variable differently, the final value might depend on whichever threads finishes last, a phenomenon known as *data race*. In general, OpenMP places the heavy burden of ensuring correctness on the application programmer. As a result, getting appreciable speedup in large programs without disrupting correctness proves challenging.

Of course no discussion on multiprocessing is complete without covering *Graphics Processing Units* (GPUs), the latest class of computing processors that are having a large impact on scientific computing. GPUs are best described as an array of *streaming multiprocessors* where each multiprocessor is designed to efficiently execute a large number of threads. Each multiprocessor further organizes the large number of threads into smaller thread groups that can execute one common instruction.[6] By efficiently switching context between multiple thread groups, a multiprocessor can ensure apparent concurrent progress on a rather large number of threads. Conceptually, this may be seen as extending the principle of SIMD to *Single Instruction Multiple Thread* (SIMT), although from the perspective of a programmer this is purely a matter of nuance. For efficient use of GPUs, it is desirable to organize the computational kernels into a hierarchy of thread groups such that at the finest level of this hierarchy, all the threads are performing the same computation. Often the starting point for such an organization is to transform loop computations into thread groups/blocks. However, the level of parallelism required to get the most out of a GPU is rather large ($\sim$1000s of threads) and one might be hard pressed to express such large loops for a majority of a program. Moreover, the available memory on a GPU is smaller on a per-thread basis making the efficient use of its capacity difficult. This requires going beyond simple ‘data-level parallelism’, which is the subject of the next subsection.

Finally, while the concepts listed above are illustrative, using them in concert for a complex program requires considerable effort, often using trial and error. A pragmatic approach is to identify the portions of a code that may be the most time-consuming, and reasoning about how they could be improved for a given platform. The solution could involve some combination of the approaches above and at different levels. For instance, one could employ multi-threading at a high level and get further speed up by vectorizing or cache blocking the lower levels of nested loops. This effort is made further challenging by the fact that a solution that works best for one platform might not carry over easily to a different platform due to a slight difference in the node-level architecture. The best programs parametrize their codes for the various levels of parallelism such that the parameters reflect the specifics of the architecture. But such a straightforward solution might elude all but the simplest codes.

3.3.3 Data, Task and Hybrid Parallelism

The entire preceding discussion, starting from distributed memory parallelism using domain decomposition to shared-memory parallelism using multiprocessing, was illustrated using examples of *data-level parallelism*, i.e. computations that can be performed in parallel on multiple data elements, which in the case of CFD invariably become solution variables at the computational grid points. As the discussion on GPUs illustrated, with the increasingly complex node-level architectures, data parallelism is no longer sufficient to fully utilize HPC computing resources. An alter-

[6]NVIDIA, which has pioneered use of GPUs for scientific computing, has developed a full-fledged programming model—CUDA—that provides constructs for the thread hierarchy. The smallest group of threads that execute a common instruction is called a *warp* and from a performance perspective having all threads in a warp do the same computation without diverging is critical.

native paradigm, *task parallelism*, is becoming increasingly important. Simply put, task parallelism is orthogonal to data parallelism and can be thought of as independent sets of computation that can be performed on the same data element. The term independent here refers to concept that the sets of computation have inputs and outputs that do not depend on each other. An example from a combustion perspective would be the computations of viscosity, thermal conductivity and species diffusivities. These quantities are required for different conservation equations, and their computations are independent and can be performed in parallel. Yet, it is fairly common for programmers to express these operations in sequence in a code since that is what all the widely used programming languages enable. Programming languages/models that allow expressing task parallelism are far less common. As the GPU discussion illustrates, to get the best performance, one might be required to exploit both data and task parallelism, i.e. *hybrid parallelism*. For the GPU example, this maybe accomplished by launching, on each multiprocessor, separate thread groups for independent tasks but have each thread within a group perform the same task on multiple data elements.

A fairly recent, and radical, development in parallel computing is the concept of *asynchronous many-task* (AMT) programing models and runtimes (Bennett et al. 2015). In this paradigm, a programmer is not required to manually reason about, and order, computational kernels in a program for parallel execution of independent tasks. Rather, the programmer is required only to specify tasks and their inputs and output dependencies. The *runtime* does the analysis of determining when a certain task is ready to be executed, based on whether its input dependencies are satisfied, and issues the task for execution on the next available resource. In such a paradigm, there is no notion of synchronous or 'in-order' execution, as would happen in common programming languages like C/C++/Fortran, and the actual order of execution only respects the data dependencies of the tasks as specified by the programmer. Treichler et al. (2018) describe the implementation of a combustion DNS code in one such AMT runtime "legion" (Legion 2017). Bennett et al. (2015) report a systematic comparative study of a few state-of-art AMT runtimes.

3.4 Physics and Numerical Aspects

A discussion on the physics and numerical aspects of computational combustion was deliberately set to follow the section on HPC aspects. Historically, combustion codes have been developed based on the classes of problems one was interested in solving, which establishes the framework for the set of physics and associated numerics. Considerations of HPC usually follow later. In inverting the perspective, we hope to give an appreciation for what implications the physics/numerics choices have on parallel computing and in particular which choices are conducive for HPC and which might be inhibiting. It is not the intent of this section to be a comprehensive survey of numerical methods for CFD of reacting flows, which would be a vast undertaking. Rather, we intend to present step-by-step the choices confronted as viewed through the lens of HPC. In the discussion to follow, we limit ourselves to gas-phase turbulent reacting flows and do not consider aspects of multiphase reacting flows.

3.4.1 Governing Equations and Constitutive Laws

As mentioned in Sect. 3.2, a reacting flow system is described by a set of variables that are governed by conservation laws in the form of PDEs. These are typically the conservation of mass, momentum, energy, species concentrations and a thermodynamic equation of state. The governing equations are complemented by constitutive laws for molecular transport (mass diffusion for species concentrations, thermal diffusion and viscosity), thermodynamic quantities (specific heat at constant pressure/volume) and chemical kinetics. Within this framework, there is some flexibility, depending on the conditions, in choosing the form of governing equations one wishes to solve. For instance, the energy equation could be transformed into an equation for enthalpy or temperature. Likewise, for species concentrations one could choose mass fractions or mole fractions. Poinsot and Veynante (2012) give an excellent overview of the governing equations for reacting flows, various equivalent forms and simplifications.

The first choice to be made is the size of the chemical system. As mentioned in Sect. 3.2, this has a direct bearing on the size of the resulting system of equations and the computational cost. For higher hydrocarbons, as may be relevant for transportation systems, the number of species in a mechanism can be extremely large, e.g. 2885 species (11754 elementary reaction steps) for a diesel surrogate mechanism (Pei et al. 2015), so as to be prohibitively expensive. Large chemical mechanisms also involve a relatively large fraction of intermediate species that are extremely shortlived (fast chemical timescales) which makes the PDEs very stiff, compounding the problem. Hence, some sort of mechanism reduction is almost always necessary, and a reduced mechanism must be judiciously chosen such that it includes the chemical pathways relevant for the phenomenon under study, and it remains valid for the conditions (pressure, temperature and equivalence ratios) of the simulation. As an example, the RCCI simulation (Treichler et al. 2018) of a primary reference fuel (a blend of iso-octane and n-heptane) targeted the study of ignition timing in a turbulent mixture undergoing piston compression. The mechanism, chosen to be valid under the range of pressures and mixture stratifications expected in the simulation and retain the key elementary steps governing ignition chemistry, contained 116 transported species, 55 species treated with a quasi-steady-state assumption (QSSA) and 861 elementary reaction steps (Luong et al. 2013), itself reduced from a much larger detailed mechanism with 874 species and 3796 elementary steps (Curran et al. 2002).

For engineering simulations—LES or RANS—such detailed mechanisms may be unnecessary and they could be significantly reduced while still preserving the fidelity required to predict engineering quantities of interest. An excellent example of this is a two-step reduced six species chemical mechanism for kerosene–air premixed combustion by Franzelli et al. (2010). Even for DNS, the purpose is sometimes to formulate, test and validate models for *turbulence–chemistry interactions* and a simple description of the combustion chemistry suffices. For premixed combustion, under certain assumptions (Bray and Libby 1976), global single-step irreversible reaction mechanism; unity Lewis numbers of reactant and product species; and adiabatic flow and constant pressure combustion, it is possible to represent the entire thermochemical system by a single reacting scalar, a progress variable. DNS of turbulent premixed flames using just a progress variable-based description have made major contributions to the understanding and modelling of turbulence–chemistry interactions, as detailed in chapter [refer Prof. Nilanjan Chakraborty].

For thermodynamic quantities, the widely accepted practice, established by the authors of the CHEMKIN package (Kee et al. 1990), is to evaluate them as polynomials of temperature. This may seem like the more expensive approach compared to the simplification of assuming these quantities to be temperature-independent. However, this is one example where, from the perspective of computational cost, such a simplification may be unnecessary. Univariate polynomial evaluations have a high computational intensity, i.e. they require a lot of FLOPS per byte of data accessed and they are generally favourable from the perspective of easing memory bandwidth pressure. As a result, opting for the more compute-intensive option of temperature polynomials might incur no penalty on computational performance. The same argument applies to transport properties. Evaluating them as polynomials of temperature, as established by the TRANSPORT package (Kee et al. 1986), might not incur significant penalty on the computational cost. However, for multi-species mixtures, there still is a choice between evaluating the transport properties using a mixture-averaged or a full multicomponent formulation for the molecular transport coefficients. The mixture-averaged formulation is effectively a weighted sum of the transport coefficients of the individual components, evaluated as polynomials of temperature, weighted by their concentrations. The full multicomponent formulation requires inverting a matrix of dimensions equal to the number of components, which can be extremely expensive. While it is well known that for laminar flames the two formulations yield considerably different results, a recent DNS of a temporally evolving turbulent stratified jet flame (Bruno et al. 2015) observed that, statistically, the quantitative differences between a mixture-averaged and multicomponent diffusion formulation were negligible in the turbulent flame. While this may be encouraging, it is not conclusive and it is fair to expect the differences between the two formulations to be regime dependent and greater for systems with a large number of species.

3.4.2 Compressible Versus Low-Mach Formulations

A large subset of turbulent combustion applications occur in low-speed subsonic flows. Under such conditions, the timescales pertaining to advection are much larger compared to that of acoustic propagation and yet stability considerations of time advancement schemes will dictate that the time step be limited by the acoustic timescale. Even though the density is varying in the domain, its change arises to leading order due to the temperature change from combustion and not due to thermodynamic compression/expansion. Representing such a system by a conventional compressible formulation will require needlessly small time steps.

An elegant solution that takes advantage of the separation of the acoustic and advective timescales is the low-Mach formulation for reactive flows (Tomboulides et al. 1997; Najm and Knio 2005; Nonaka et al. 2012), which filters out the acoustic waves by decomposing pressure into a thermodynamic and a hydrodynamic part

$$p(x,t) = p_{therm}(t) + p_{hydro}(x,t). \tag{3.6}$$

The equation of state involves only the thermodynamic pressure, p_{therm}, which is assumed to equilibrate in the whole domain instantly and hence is a function only of time, while the momentum equation involves only the gradient of the hydrodynamic pressure, p_{hydro}. In terms of the formulation, this decomposition transforms the continuity equation and the equation of state into a constraint on velocity divergence that requires solving an elliptic equation with spatially variable coefficients. From an HPC perspective, this maybe seen as a drawback of this approach. An elliptic PDE has the attribute that the solution at every point in the domain is influenced by every other point. Solving such a PDE, in a distributed memory setting, requires an all-to-all exchange of information which is communication intensive. In practice, sophisticated algorithms are used to solve elliptic PDEs in an efficient way but effectively the global information exchange has to happen in one form or another. In contrast, this is avoided in a compressible formulation since the flow of information by acoustic propagation is resolved and the domain of influence for the PDE solution at any grid point is localized. The choice then is between a compressible formulation that restricts the time steps and increases the time to solution but with a simpler algorithm (and code) versus the low-Mach formulation that allows large time steps but at the expense of a much more complicated numerical algorithm and code.

3.4.3 Spatial and Temporal Discretizations

The most consequential design choice for any CFD program, the one that establishes the entire numerical, algorithmic and computational framework for a code, is the choice of spatial and associated temporal discretizations for the PDEs. Here too, the landscape is vast and we do not attempt an exhaustive discussion. Rather, we focus on the prominent classes of numerical methods that have emerged for reactive flow problems and assess their pros and cons from an HPC perspective.

The choice of the spatial discretization is guided first and foremost by what kind of simulations one wishes to perform and what accuracy is required. For simulations that are fundamentaly of academic interest and whose aim to investigate and quantify the multiscale nature of turbulence–chemistry interactions, the computational domains can be kept simple but a high accuracy is desired for spatial derivatives.

Finite-difference schemes are a natural choice. Early DNS codes for incompressible turbulence were based on *spectral* or *pseudo-spectral* methods[7] which solve the Fourier modes of velocity. The transformation of Navier–Stokes equations to the Fourier space allows one to precisely control the range of wavenumbers that can be resolved and hence the Reynolds numbers that are affordable, but these methods are limited to periodic spatial domains. High-order accurate spatial finite-difference schemes (Lele 1992; Kennedy and Carpenter 1994) emerged from this need to perform DNS in physical space but with targeted spectral-like accuracy. These schemes are simple to implement and offer a high order of accuracy at relatively modest computational expense and combustion DNS codes have successfully implemented schemes that are up to tenth-order accurate (Jenkins and Cant 1999). But the main concern is that the formulation of these schemes is not mathematically conservative and choosing a scheme of modest-order accuracy might violate domain-wide global conservation laws. Finite-volume discretization, on the other hand, is conservative by construction and preserves domain-wide conservation. However, finite-volume schemes become increasingly difficult to formulate for higher order accuracy and require large stencil widths increasing the computational cost. Of a secondary concern is the ability to handle irregular geometry domains. The finite-difference approach, being of the method of lines mould, is restricted to only simple computational domains, purely rectangular domains with structured meshes. The finite-volume method, in principle, can handle complex geometries with tetrahedral mesh elements, but the difficulties in formulating high-order accurate schemes for tetrahedral elements become more severe. Multi-block body-fitted curvilinear meshes also allow one to handle modestly complex geometries but these are usually preferred for engineering, as opposed to academic simulations and finite-volume methods are a better choice since ensuring conservation is more important than resolving spatial gradients with high fidelity.

From an HPC perspective, the choice of temporal discretization and the overall time advancement scheme has a much larger bearing. The main issue confronting turbulent combustion simulations is that there are four relevant timescales governed by the distinct physical processes: convection, acoustics, diffusion and chemistry, and they can be different by orders of magnitude. The choice of which timescales to resolve explicitly, which to handle implicitly and how to couple them all consistently determines the overall temporal discretization framework. The previous section already elucidated the choice between resolving and filtering acoustic timescales when they are much smaller than convective timescales. For convective–diffusive–reactive flows, three classes of temporal schemes are apparent: fully explicit schemes, coupled implicit–explicit (IMEX) schemes and decoupled operator splitting schemes:

- Fully explicit schemes represent the brute force approach and the time step is chosen to be smaller than the smallest relevant timescale. These schemes are relatively

[7]The difference between *spectral* and *pseudo-spectral* methods lie in how the nonlinear convective term was handled. In the spectral method, all computations were in the wavenumber space, but pseudo-spectral methods use an intermediate step to transform the velocity Fourier modes to physical space, compute the convective term in the physical space and transform it back to the Fourier space.

straightforward to implement and the spatial domain of influence is nearest neighbour, i.e. advancing the solution in time at a grid point requires information only from the grid points in the immediate vicinity. From a distributed memory perspective, this is extremely attractive since the time advancement algorithm for a set of grid points assigned to a node requires communication only with a handful of other nodes that contain the adjacent grid points. Widely used classes of explicit methods (e.g. Runge–Kutta and Adams–Bashforth) differ in terms of the number of stages involved in an overall time step and the storage required for the intermediate solution. The former determines the computational cost of the scheme and the latter determines the memory footprint.

- Coupled implicit–explicit schemes (IMEX) schemes allow one to choose much larger time steps, by treating the fast processes implicitly. The low-Mach formulation can be construed as being explicit for convective and implicit for acoustic processes. The term coupled refers to the fact that all the physical processes are considered together in time advancing each PDE, which makes it easier to reason about the temporal order of accuracy, compared to the decoupled schemes (see below). However, the implicit treatment incurs the cost that the spatial domain of influence is no longer nearest neighbour and this manifests itself as a global linear system of form $\mathrm{A}\phi^{n+1} = \mathrm{B}\phi^{n} + \mathrm{d}$, where ϕ is the solution vector at all the grid points with superscript n denoting the time, A and B are matrices (usually sparse) that contain the coefficients of the discretization, and d is the vector containing the spatial boundary conditions. Solving such systems requires sophisticated linear system solvers (e.g. multi-grid methods and Krylov methods) whose implementation is fairly involved.
- Operator split methods (Strang 1968) also use a combination of implicit and explicit schemes, but they decouple physical processes and consider the time advancement due to the various processes one at a time. A typical implementation for reacting flow (Najm and Knio 2005) involves advancing the solution state by the diffusion terms for one half time step, followed by a full time step advancement by the reaction terms and finally another half step advancement by the diffusion terms. This offers the advantage that a different solver, best suited for the relevant terms, can be used for each stage. The terms treated implicitly will still require sophisticated solvers and in addition operator split methods introduce splitting errors requiring special numerical treatment to preserve consistency across the physical processes. The end result is a temporal scheme that requires a fairly complicated algorithm even for guaranteeing modest orders of temporal accuracy (Descombes et al. 2014).

On a concluding note, the choice of temporal scheme should be guided by the separation of the timescales, if any, in the target simulation. If all the timescales are comparable, it is best to use fully explicit methods due to their simplicity, ease of implementation and desirability from a distributed memory perspective. If the timescales are disparate, the combined implicit–explicit schemes may be a better option for reasons described above. Moreover, a fairly elaborate set of linear system solvers, which are usually required for the implicit schemes, are available as open libraries which have been designed and specifically optimized for HPC (PETSC 2017; Trilinos Home Page 2017).

3.4.4 An Exemplar Combustion DNS Code: S3D

The preceding discussion highlights the various factors, by no means exhaustive, that guide the design of a combustion code. To illustrate how they are all put together we present an exemplar code S3D (Chen et al. 2009), developed at the Combustion Research Facility, Sandia National Laboratories. S3D is a massively parallel solver developed for performing DNS of multi-species gas-phase turbulent reacting flows. It employs explicit high-order accurate central difference schemes for spatial discretization and explicit multi-stage low storage Runge–Kutta scheme for temporal discretization (Kennedy and Carpenter 1994). It interfaces with the CHEMKIN and TRANSPORT library routines for incorporating detailed finite-rate chemical kinetics and mixture-averaged molecular transport properties. The longstanding version of the code, written in Fortran 90, uses MPI-based distributed domain decomposition. More recently, it has been refactored for heterogeneous architectures using a hybrid MPI+OpenACC implementation and also a radical task-based programing model called Legion (Treichler et al. 2018). Over the years, S3D has performed turbulent combustion DNS simulations of remarkable scale and chemical complexity. Due to its simplicity and ease of use, it has been ported to a variety of supercomputer architectures with excellent parallel scalability and performance. Moreover, it has served as a vehicle that enabled various other facets of HPC research: data analytics and visualization (Ye et al. 2016), data compression (Austin et al. 2016), large-scale parallel I/O (Schendel et al. 2012) and HPC resilience (Gamell et al. 2017).

3.5 Data Analyses

While performing massive turbulent combustion simulations is challenging in its own right, the effort does not end there. Turbulent combustion is the classic multiscale multi-physics scientific application and accordingly its simulations generate large volumes of data rich in information. Extracting insight buried in these data sets is also often a large computational undertaking. The recent S3D simulation shown in Fig. 3.1 has a computational problem size of 1.8×10^{11}. If the data are represented by double-precision floating-point numbers (8 bytes), this translates to each snapshot of this data set having a file size of 1.44 TB! Considering a few hundred time snapshots are required for capturing the temporal evolution, the total data size approaches PetaBytes, which is well beyond the capacity of even modest computing clusters, let alone desktop computers.

The only pragmatic approach for analysing such volumes of data is to perform the analysis *in place*, i.e. the same computing resource where the data is generated. This dictates the use parallel computing for the analyses as well and domain decomposition serves as a good strategy for distributed parallelism. Below are some commonly recurring motifs for combustion data analyses:

- **Descriptive statistics**: Statistical moments (mean, variance, covariance, etc.) gathered over ensembles in the 'homogeneous' dimensions. For temporally evolving simulations, homogeneous dimensions are the spatial dimensions that are periodic while for statistically stationary simulations time is a homogeneous dimension. In either case, gathering statistics requires global communication and aggregation of data in space and/or time.
- **Feature-based analyses**: Often, it is of interest to hone in on specific features of the turbulent flames (ignition kernels, extinction regions, iso-surfaces/iso-volumes of reacting scalars, strain-/vorticity-dominated regions, etc.). Feature identification, extraction and tracking is a broad topic requiring specialized algorithms for different types of features (e.g. Marching cubes algorithm for iso-surface construction) and these need to be further implemented in a parallel setting. An example of an iso-surface based analysis is the geometric flame thickness studied in Chaudhuri et al. (2017).
- **Phase-space analyses**: It is sometimes required to transform from a space–time domain to the solution variables domain (phase-space). Examples of analyses in such transformed domains include constructing joint/conditional/marginal probability distribution functions (PDFs), trajectories in phase space, empirical low-dimensional chemical manifolds, etc.
- **Fourier spectral analyses**: Assessing classical hypotheses borne out of the spectral view of the turbulence energy cascade requires constructing energy (Kolla et al. 2014) and dissipation spectra (Kolla et al. 2016) of velocities and reactive scalars. Fast Fourier transforms are integral to this which are challenging in a distributed data set since they are communication intensive and extremely expensive.
- **Filtering and multiscale analyses**: In multiscale settings, it is sometimes desirable to extract scale-specific features and statistics. Examples of such analyses include wavelet transforms, bandpass filtering, explicit spatial filtering in the vein of LES, etc.

It is obvious that the space for analyses is rather broad and in some sense more diverse, in terms of algorithmic needs and implementations, compared to the PDE solution algorithms. Another concept, which takes the idea of in-place analyses even further, is in-situ analysis which refers to a paradigm where the analyses and solver are much more tightly integrated and the data is being analysed as it is generated. Looking towards the future, and extrapolating the rate at which the scale of combustion simulations is growing, generating and saving such volumes of data seems unsustainable. It appears that in-situ analyses will play a much bigger role in the overall workflow and the final data sets generated will have to be reduced by orders of magnitude. A good example of what such a workflow might look like is presented by Bennett et al. (2012).

Acknowledgements This work was supported by Sandia National Laboratories, a multi-mission 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 Energys National Nuclear Security Administration under contract DE-NA-0003525. The authors acknowledge the support of the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations—the Office of Science and the National Nuclear Security Administration—responsible for the planning and preparation of a capable exascale ecosystem including software, applications, hardware, advanced system engineering and early testbed platforms to support the nations exascale computing imperative. Helpful comments and inputs from Dr. Xinyu Zhao (University of Connecticut), Dr. Giulio Borghesi (Sandia National Labs) and Dr. Aditya Konduri (Sandia National Labs) are gratefully acknowledged.

References

GRI-Mech 3.0. http://www.me.berkeley.edu/gri_mech
Legion: a data-centric parallel programming system. http://legion.stanford.edu
MPI forum. http://mpi-forum.org
Open multi-processing. http://www.openmp.org
PETSc web page. http://www.mcs.anl.gov/petsc
Programming guidelines for vectorization. https://software.intel.com/en-us/node/695829
Trilinos home page. https://trilinos.org
Austin W, Ballard G, Kolda T (2016) Parallel tensor compression for large-scale scientific data. In: IPDPS'16: Proceedings of the 30th IEEE international parallel and distributed processing symposium, pp 912–922. https://doi.org/10.1109/IPDPS.2016.67
Bennett J, Abbasi H, Bremer PT, Grout R, Gyulassy A, Jin T, Klasky S, Kolla H, Parashar M, Pascucci V, Pebay P, Thompson D, Yu H, Zhang F, Chen J (2012) Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In: International conference on high performance computing, networking, storage and analysis (SC)
Bennett J, Clay R, Baker G, Gamell M, Hollman D, Knight S, Kolla H, Sjaardema G, Slattengren N, Teranishi K, Wilke J, Bettencourt M, Bova S, Franko K, Lin P, Grant R, Hammond S, Olivier S, Kale L, Jain N, Mikida E, Aiken A, Bauer M, Lee W, Slaughter E, Treichler S, Berzins M, Harman T, Humphrey A, Schmidt J, Sunderland D, McCormick P, Gutierrez S, Schulz M, Bhatele A, Boehme D, Bremer PT, Gamblin T (2015) ASC ATDM Level 2 Milestone #5325: asynchronous many-task runtime system analysis and assessment for next generation platforms. Technical report SAND2015-8312, Sandia National Laboratories, Albuquerque, NM
Bray K, Libby PA (1976) Interaction effects in turbulent premixed flames. Phys Fluids 19(11):1687–1701
Bruno C, Sankaran V, Kolla H, Chen J (2015) Impact of multi-component diffusion in turbulent combustion using direct numerical simulations. Combust Flame 162:4313–4330
Burke M, Chaos M, Ju Y, Dryer F, Klippenstein S (2012) Comprehensive H_2/O_2 kinetic model for high-pressure combustion. Int J Chem Kinet 44:444–474
Chaudhuri S, Kolla H, Dave H, Hawkes E, Chen J, Law C (2017) Flame thickness and conditional scalar dissipation rate in a premixed temporal turbulent reacting jet. Combust Flame 184:273–285
Chen JH, Choudhary A, de Supinski B, DeVries M, Hawkes ER, Klasky S, Liao WK, Ma KL, Mellor-Crummey J, Podhorszki N, Sankaran R, Shende S, Yoo CS (2009) Terascale direct numerical simulations of turbulent combustion using S3D. Comput Sci Discov 2:015001
Curran H, Gaffuri P, Pitz W, Westbrook C (2002) A comprehensive modeling study of iso-octane oxidation. Combust Flame 129:253–280
Denning P (2005) The locality principle. Commun ACM 48(7):19–24

Descombes S, Duarte M, Dumont T, Laurent F, Louvet V, Massot M (2014) Analysis of operator splitting in the non-asymptotic regime for nonlinear reaction-diffusion equations. Application to the dynamics of premixed flames. SIAM J Numer Anal 52:1311–1334

Echekki T, Chen J (1996) Unsteady strain rate and curvature effects in turbulent premixed methane-air flames. Combust Flame 106:184–202

Echekki T, Chen J (2002) High-temperature combustion in autoigniting non-homogeneous hydrogen/air mixtures. Proc Combust Inst 29:2061–2068

Franzelli B, Riber E, Sanjose M, Poinsot T (2010) A two-step chemical scheme for kerosene-air premixed flames. Combust Flame 157:1364–1373

Gamell M, Teranishi K, Mayo J, Kolla H, Heroux M, Chen J, Parashar M (2017) Modeling and simulating multiple failure masking enabled by local recovery for stencil-based applications at extreme scales. IEEE Trans Parallel Distrib Syst

Gropp W, Lusk E, Skjellum A (1994) Using MPI: portable parallel programming with the message-passing interface. Scientific and engineering computation series. MIT Press

Jenkins K, Cant R (1999) Direct numerical simulation of turbulent flame kernels. In: Knight D, Sakell L (eds) Proceedings of the second AFOSR conference on DNS and LES. Kluwer

Kee R, Rupley F, Miller J (1990) The CHEMKIN thermodynamic data base. Technical report SAND-87-8215B, Sandia National Laboratories

Kee RJ, Dixon-Lewis G, Warnatz J, Coltrin ME, Miller JA (1986) A Fortran computer code package for the evaluation of gas-phase multicomponent transport properties. Technical report SAND86-8246, Sandia National Laboratories

Kennedy C, Carpenter M (1994) Several new numerical methods for compressible shear-layer simulations. Appl Numer Math 14:397–433

Kolla H, Hawkes E, Kerstein A, Chen J (2014) On velocity and reactive scalar spectra in turbulent premixed flames. J Fluid Mech 754:456–487

Kolla H, Zhao XY, Chen J, Swaminathan N (2016) Velocity and reactive scalar dissipation spectra in turbulent premixed flames. Combust Sci Technol 188:1424–1439

Lele S (1992) Compact finite-difference schemes with spectral like resolution. J Comput Phys 103:16–42

Luong M, Luo Z, Lu T, Chung S, Yoo C (2013) Direct numerical simulations of the ignition of lean primary reference fuel/air mixtures under HCCI condition. Combust Flame 160:2038–2047

Najm H, Knio O (2005) Modeling low Mach number reacting flow with detailed chemistry and transport. J Sci Comput 25:263–287

Nonaka A, Bell J, Day M, Gilet C, Almgren A, Minion M (2012) A deferred correction coupling strategy for low Mach number flow with complex chemistry. Combust Theory Model

Pei Y, Mehl M, Liu W, Lu T, Pitz W, Som S (2015) A multi-component blend as a diesel fuel surrogate for compression ignition engine applications. J Eng Gas Turbines Power (GTP-15-1057)

Poinsot T, Veynante D (2012) Theoretical and numerical combustion, chapter 1. Edwards

Raman V, Fox RO (2016) Modeling of fine-particle formation in turbulent flames. Annu Rev Fluid Mech 48:159–190

Sankaran R, Hawkes E, Chen J, Lu T, Law C (2007) Structure of a spatially developing turbulent lean methane-air Bunsen flame. Proc Combust Inst 31:1291–1298

Schendel ER, Pendse SV, Jenkins J, Boyuka II DA, Gong Z, Lakshminarasimhan S, Liu Q, Kolla H, Chen J, Klasky S, Ross R, Samatova NF (2012) ISOBAR Hybrid compression-I/O interleaving for large-scale parallel I/O optimization. In: Proceedings of the 21st international symposium on high-performance parallel and distributed computing, HPDC '12. ACM, New York, NY, USA, pp 61–72. https://doi.org/10.1145/2287076.2287086. http://doi.acm.org/10.1145/2287076.2287086

Strang G (1968) On the construction and comparison of difference schemes. SIAM J Numer Anal 5:503–517

Tomboulides A, Lee J, Orszag S (1997) Numerical simulation of Low-Mach number reactive flows. J Sci Comput 12:139–167

Treichler S, Bauer M, Bhagatwala A, Borghesi G, Sankaran R, Kolla H, McCormick P, Slaughter E, Lee W, Aiken A, Chen J (2018) S3D-Legion: an exascale software for direct numerical simulation (DNS) of turbulent combustion with complex multicomponent chemistry. In: Straatsma T, Williams KAT (eds) Exascale scientific applications scalability and performance portability, chapter 12. CRC Press

Ye Y, Neuroth T, Sauer F, Ma KL, Borghesi G, Konduri A, Kolla H, Chen J (2016) In situ generated probability distribution functions for interactive post hoc visualization and analysis. In: IEEE sixth symposium on large data analysis and visualization (LDAV)