Chapter 19

Software for the Scalable Solution of PDEs

Satish Balay
William D. Gropp
Lois Curfman McInnes
Barry F. Smith [1]

19.1 Introduction

The numerical approximation of the solution of partial differential equations (PDEs), which can be used to model physical, chemical, and biological phenomena, is an important application of parallel computers, as we have seen in previous chapters and as is discussed in [Kon00]. Early efforts to build programs to solve PDE problems had to start from scratch, building code for each algorithm used in the solution process. This custom approach has two major drawbacks: it limits the use of parallel computers to a small number of groups that have the resources and expertise to develop these codes, and it hampers the ability to take advantage of developments in parallel algorithms. In conventional, serial programming, both of these drawbacks were partially solved by developing libraries of routines that contained the best numerical analysis and implementation techniques. The same route is being followed for parallel libraries, though parallelism introduces additional complications. Handling these complications has caused many groups to rethink the structure of numerical libraries, leading to better software even for uniprocessor applications. In this chapter, we will cover some of the issues and solutions in the context of the Portable, Extensible Toolkit for Scientific Computation (PETSc), a collection of tools for the numerical solution of PDEs and related problems [BGMS, BGMS00].

[1] Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL 60439-4844. balay@mcs.anl.gov, gropp@mcs.anl.gov, mcinnes@mcs.anl.gov, bsmith@mcs.anl.gov, `http://www.mcs.anl.gov/petsc`.

Many issues arise in designing a parallel program to approximate the solution to a PDE. A key issue is managing *software complexity*, or the interrelationships among code for the various facets of the overall simulation. Three additional critical issues are *numerical algorithms*, *data distribution* and *data access patterns*. Ironically, these are exactly the same issues of importance for sequential solution; only the scale is different.

We begin by asking how we can organize our program to exploit parallelism, manage the complexity of the parallel application, and effectively use the available computing resources. The approach that we take in this chapter is to start at the top, organizing the program around the mathematics of the approximation. As we will see, this single organizing principle not only provides a method for effectively distributing the computation across the processors, but also allows us to change algorithms easily and thereby to incorporate new methods as they become available. Such capabilities enable numerical software developers to better serve the needs of computational scientists, who can leverage expertise encapsulated within existing libraries without needing to commit to a particular solution strategy and to risk making premature choices of data structures and algorithms. Engaging application scientists in library use without requiring excessive commitment on their part is a critical facet of overcoming the all too frequent perception that applications must implement from scratch all facets of modeling to achieve good performance. In fact, the "roll your own" approach is undesirable because it implies that implementation decisions must be made a priori, before experimentation with realistically sized problems can determine a code's most serious bottlenecks. Using abstractions in library design provides the flexibility for application programmers to use library-provided functionality from the beginning of an application's development as well as to inject new algorithms and data structures (which may be written by library developers, the application scientists themselves, or third parties) during the lifetime of the application code.

The remainder of this chapter is organized as follows. Section 19.2 presents an overview of background for the numerical solution of PDEs, while Section 19.3 explains in more detail the challenges in parallel computations for PDE-based models. Section 19.4 overviews various possible possible solution strategies and lays out the territory for the remaining discussion in this chapter. Section 19.5 discusses the approach used within the PETSc software, with emphasis on the use of mathematical abstractions as an organizing principle that can help to address issues in algorithmic flexibility, efficient use of computational resources, and composability with external tools. Section 19.6 provides an overview of recent work throughout the high-performance computing community in parallel PDE software. Finally, we conclude in Section 19.7 with some observations and recommendations.

19.2 PDE Background

Partial differential equations that model scientific applications span the complete range of elliptic, parabolic, and hyperbolic types and combinations thereof.

As discussed in [Hea97], hyperbolic PDEs describe time-dependent physical processes, such as wave motion, that are not evolving toward a steady state; parabolic PDEs describe time-dependent physical processes, such as heat diffusion, that are evolving toward a steady state; and elliptic PDEs describe processes that have already reached a steady state, or equilibrium, and hence are independent of time. In addition, problems can be of mixed type, varying by region or being multicomponent in a single region (e.g., a parabolic system with an elliptic constraint). Generally, elliptic equations are easy to discretize, but challenging to solve because their Green's functions are global: the solution at each point depends upon the data at all other points. Conversely, hyperbolic equations are challenging to discretize because they support discontinuities but are easy to solve when addressed in characteristic form [KKS99].

Many applications are based on replacing an infinite dimensional continuous PDE system with an approximate finite dimensional discrete system that can be solved numerically. A wide range of numerical algorithms can be employed for such problems (see, e.g., [MM94, Hea97]). We often categorize approaches as being explicit or implicit, depending on whether the algorithm computes the solution at a given mesh point using only past iterates or using current information from other mesh points as well. Explicit algorithms update the solution vector simply by using discretization information from neighboring mesh points; no global linear or nonlinear solves are used. Explicit methods are relatively straightforward to implement in parallel, since communication is generally needed only for global reductions (e.g., vector norms) and ghost point transfers for local discretization. In contrast, implicit methods update all (or most) variables in a single global linear or nonlinear solve. Consequently, since they propagate information throughout the global problem domain at each iteration, implicit methods can often converge in fewer time steps than do explicit methods, particularly for large-scale problems. Unfortunately, the challenges in parallel implementations of implicit methods are considerable, due to the inherently global nature of the operators.

Intermediate between these extremes are semi-implicit methods, in which subsets of variables (for example, pressure) are updated with global solves. Most of the remaining discussion in this chapter will focus on issues arising in implicit and semi-implicit methods, since these can be especially effective for large-scale problems and are arguably more difficult to implement in parallel than explicit techniques.

19.3 Challenges in Parallel PDE Computations

A commonly used approach for solving a PDE system is to replace the partial derivatives within the system (for example, spatial and time derivatives) with discrete approximations based on finite differences, volumes, or elements and then to solve numerically the resulting algebraic system of (time-dependent, nonlinear) equations. Without going any further, parallelism already introduces an issue: how is the solution vector distributed among the processors? While this question may at first glance seem straightforward, it immediately leads

to deeper issues regarding data access patterns as well as interrelationships among software for various facets of parallel PDE solution, such as interfaces between partitioning tools and algebraic solvers. We must somehow manage this complexity without sacrificing good performance; these dueling tradeoffs are particularly challenging when using the distributed memory resources and multilevel memory hierarchies of modern architectures.

19.3.1 Software Complexity

We thus recognize immediately that software for parallel numerical PDEs (e.g., tools for time evolution and algebraic nonlinear and linear solution) cannot be developed in isolation, but rather must be considered in relationship to tools that partition the problem domain. Further consideration reveals that typical scientific simulations need many additional capabilities, such as mesh generation, PDE discretization, derivative computations, adaptive mesh refinement and coarsening, optimization, sensitivity analysis, data management, visualization, and parallel performance analysis. Moreover, each computational phase may have a different preferred data representation, so that we must consider tradeoffs in computation time and storage space when transitioning between phases.

In recent years the high-performance computing community have developed a variety of software packages for these phases; however, the combined use of multiple software packages in a given application is a continuing challenge because of incompatibilities in data structures and interfaces. In fact, the situation appears much simpler when considering individual facets of PDE simulations; the more difficult challenges arise when simultaneously considering multiple phases. Understanding the relationships among these phases is critical for the design of efficient software because within the realm of complete PDE-based simulations, no single software component performs in isolation. Moreover, no single research group can expect to encompass the expertise for cutting-edge capabilities in all areas. Composability and interoperability of different tools via well-defined abstract interfaces are critically important, and, as further discussed in Section 19.5.3, this area is now receiving considerable attention throughout the high-performance computing community.

19.3.2 Data Distribution and Access

As we saw in Chapter 3, the performance of CPUs has increased far faster than the performance of the computer's memory. In contemporary systems it can take one hundred clock cycles or more to access main (as opposed to cache) memory. As as result, the performance of many applications is bounded by the performance of the main memory system, not the CPU [GKKS99], even on single-processor systems. Achieving high performance on these systems requires careful attention to the use of memory. For example, it is common in applications to use separate variables for different physical variables, such as `p` for pressure and `v` for velocity. However, code that accesses these variables in a loop over a mesh can suffer significant performance problems. Instead (at

least on RISC-based systems), it is important to *interlace* the variables: define a single variable where the first index (in Fortran) indicates the physical quantity (e.g., pressure or velocity) and the following indices refer to the mesh. In this way, a loop over the mesh accesses memory in a more efficient fashion. Similarly, it is important not to create algorithms that replace a single, multicomponent problem with a collection of single-component (or scalar) problems. While both formulations may involve roughly the same number of floating-point operations, the collection of solvers will often involve far more memory motion and thereby lead to poor efficiencies.

These problems are exacerbated in parallel computers. In addition to the large latency of access to main memory, there is an even larger latency, coupled with significantly lower bandwidth, to the memory on remote nodes or processors. Thus, even greater attention must be paid to both the location of data (data distribution) and the mode by which it is accessed. A simple example of this is given in Section **??**, where different distributions of data to different processes lead to different efficiencies. That example is a case of a more general principle: minimizing the surface to volume ratio of the data distribution. This principle arises because, for PDE calculations, the most common operations involve communicating neighbor data to processes that contain adjacent elements of the mesh. Minimizing the data that must be moved between processes is accomplished by minimizing the area of the joints between adjacent processes, relative to the mesh of unknowns. In practical terms, for a regular two-dimensional mesh, this means that the mesh should be divided into squares (a two-dimensional decomposition) rather than strips (a one-dimensional decomposition). Organizing the numerical algorithm to limit accesses to remote data can also have a significant beneficial effect on performance; for iterative solutions to linear equations, this is often accomplished by choosing a preconditioner that uses only or mostly data local to a process.

The large latency of access to remote memory also has implications for both algorithm and software design. In order to reduce the impact of latency, the simplest approach is to aggregate data transfers so that a single operation moves many data items. This approach encourages a software design that follows a two-phase model: in the first phase, as much data as possible is requested; in the second phase, the computation waits until the data arrives. This technique allows the memory system and interprocess communication system to move the data most efficiently, in contrast to the more common model of requesting a single item and then waiting until it is available. One concrete example of this situation arises in the assembly of a sparse matrix (see [BGMS97] for a detailed discussion). The most obvious approach is to add one element at a time to the matrix, ensuring that as each entry is added, the sparse matrix data structures are updated. However, even for a single process, it is often more efficient to wait to update the sparse matrix data structures until many (possibly all) elements have been added to the matrix. As we have indicated, in the multiprocess case it is even more important to defer updating the matrix data structures until many elements can be communicated with each operation. These considerations apply to both message-passing and thread-based models

of parallelism, since they reflect the costs to access remote memory. Under the thread-based model, smaller aggregates can be used because the latency is lower than in the message-passing model; however, the latency is still large relative both to local memory operations and to floating-point operations.

19.3.3 Portability, Algorithms, and Data Redistribution

If the above were not enough, any significant application must be prepared to evolve over time. Both raw computer speed and the performance of algorithms have grown tremendously over the past thirty years (see Figure 1.1). Hence, an application must be written to exploit both new computing systems and new algorithms.

Portability. To exploit new computers, an application must be portable. At the very least, the application should be written in a standard computer language (such as Fortran or C, without extensions) and be careful in its assumptions about the computing environment (e.g., a C program should not assume that an `int` is a particular length). Parallel programs should use standards such as MPI, OpenMP, or HPF to maintain portability. Even with such standards, the much more difficult goal of *performance portability* (portability without sacrificing performance) can be challenging to achieve, particularly over a wide range of computer architectures [DGK84]. However, the benefits of portability are enormous. Computer performance continues to increase by leaps and bounds; portable applications can quickly take advantage of the fastest computers, independent of any particular vendor.

Algorithms. Algorithmic improvements have been at least as important as advances in computer speed for many applications. Thus, it is important that an application be portable to new algorithms as well as to new hardware. Unfortunately, there are no standards (yet) to which applications can write that will guarantee that the newest algorithm can quickly inserted into an application. Much of the rest of this chapter discusses an approach for this problem based on developing interfaces between the application and the algorithms that it uses. These interfaces reflect the problem being solved, rather than an interface to a specific algorithm. A discussion of particular algorithms for PDEs is beyond the scope of this chapter; various issues are discussed in, for example, [Hea97, KSV97, MM94, QV99, Saa96, SBG96].

Even with the continual improvement in algorithms, in many cases it is not possible to identify the best algorithm in advance. For example, preconditioned iterative methods for linear systems are powerful and effective, but their efficiency can be sensitive to details of the problem. Thus, even for an application that is not expected to be used for many years, it is important to have the ability to experiment with different methods and algorithms. This need also encourages an application design where the code interfaces to techniques that solve problems, rather than to a particular choice of algorithm.

Data Redistribution. The concerns discussed above apply to both sequential and parallel programs. Among the complexities that parallelism adds is that of data redistribution. As often noted, achieving high performance requires pay-

ing close attention to memory locality. In fact, many parallel algorithms have been developed that specify the distribution of the data for maximum efficiency. Unfortunately, the optimal data distribution for one step in an application may not be optimal for the succeeding step. For example, one popular method for solving certain kinds of PDEs is the alternating direction implicit, or ADI, method. In this method, the solution to a three-dimensional PDE is approximated by successively solving one-dimensional problems in each of the three coordinate directions. The fastest algorithms for each of these one-dimensional solves requires that the data be decomposed so that all of the data along the direction being solved is on the same processor. Switching from one coordinate direction to another requires transposing the data (an all-to-all communication). An alternative approach involves the development of more complex algorithms that minimize the time over all three coordinate directions, not just a single direction. Many parallel methods for PDEs suffer from varying degrees of scaling problems due to imperfect data distribution. Algorithms and software must work together to control the cost and complexity of data redistribution.

19.4 Parallel Solution Strategies

Chapter 10 reviews various approaches that deal with these challenges, including parallel languages, parallelizing compilers and compiler directives, computer-assisted parallelization tools, parallel libraries, and problem-solving environments. We briefly discuss issues pertaining to parallel PDE work, mention some recent research in parallel libraries for PDEs, and then explain where our work fits within this spectrum.

The complexity of PDE-based simulations makes automated analysis extremely difficult for distributed-memory parallel systems. While parallel languages and parallel compilers have worked well on shared-memory computers, particular hardware platforms (e.g., CM-5) [Thi93], or specific problems, these approaches have not yet been able to demonstrate general applicability. For example, HPF [KLS+94] is not yet up to the performance of message-passing codes, except in limited settings with much structure to the memory addressing [HKM98]. Hybrid HPF/MPI codes are possible steps along the evolutionary process, with high-level languages automating the expression and compiler detection of structured-address concurrency at lower levels of the PDE modeling. Automated source-to-source parallel translators, such as the University of Greenwich CAPTools project [IJCL96] (which adds MPI calls to a sequential Fortran 77 input) can facilitate the parallelization of legacy applications. Such tools may attain 80–95% of the benefits of the best manual practice, but the result is limited to the concurrency extractable from the original algorithm. In many cases, the legacy algorithm should, itself, be replaced. Similar comments apply to OpenMP and hybrid OpenMP/MPI approaches.

Beyond the capabilities of parallel languages, parallel compilers, and computer-assisted parallelization tools, we still need encapsulation of expertise for parallel PDEs in forms that are usable by the scientific computing community at large. We also must allow application programmers to leverage as much of their ex-

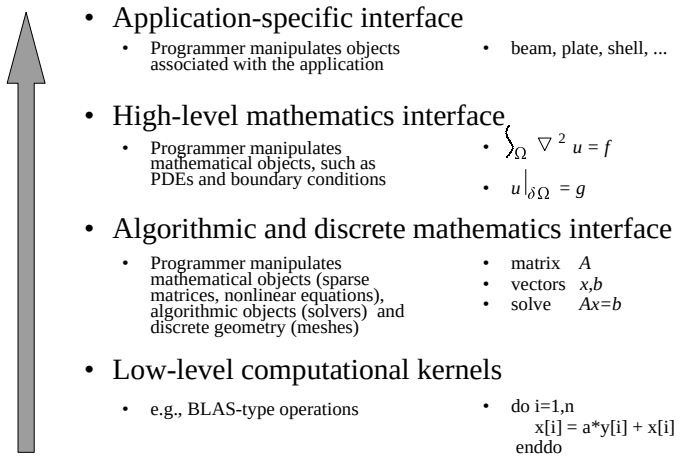

Figure 19.1: Levels of abstraction in mathematical software.

isting legacy code as possible, thereby enabling a gradual transition from the more traditional approach of "the application code does everything" to "the application code uses building blocks within software tools".

As illustrated by Figure 19.1, we can consider numerical libraries and problem solving environments (PSEs) as fitting within a spectrum of different levels of abstraction. At one end, software presents only an application-specific interface to the user, and the software handles all other facets of computation, from mesh generation to discretization to complete solution with numerical methods and data analysis. While this level of abstraction is appealing in the simplicity presented to the application scientist, there is little compile-time flexibility. At the other end of the spectrum are low-level computational kernels, which offer enormous flexibility, although the complexity of interactions is difficult to manage at this level. In the intermediate range we tend to compromise based on the strengths of both ends; no one single abstraction choice is right or wrong, but rather different tradeoffs can be made depending on particular design objectives. PSEs, which are further discussed in Chapter 14, tend to build application-oriented abstraction layers both above and below numerical library levels. Because there is no precise definition of PSEs in use in general practice, the term PSE itself does not convey sufficient information regarding the category of software of a PSE product. Examples can range from lower-level class libraries to complete environments such as engineering tools like Nastran [MSC].

The approach used within PETSc focuses on abstractions for algorithms and discrete mathematics. Such abstractions range throughout a hierarchy, where software for sophisticated PDE algorithms can be designed upon lower-level building blocks of parallel data structures. Various groups have used

similar approaches in leveraging abstractions at the PDE level for the development of parallel PDE software, including DAGH [PB], Diffpack [BL97, Dif], Kelp [FBK96, Bad], Overture [DLBQ99, BHQ], SAMRAI [HK99, KGHS], POOMA [ABC+95, POO], and UG [UG , BBJ+97]. Some of this work is discussed in Chapter 13 within the context of data structure libraries.

19.5 PETSc Approach to Parallel Software for PDEs

Now that we have abstractly discussed the various challenges in PDE solution and possible strategies for tackling them, we present concrete details of one approach. In particular, this section introduces a set of techniques used within the Portable, Extensible Toolkit for Scientific Computation (PETSc) [BGMS, BGMS00] for the development of algorithms and data structures for large-scale PDE-based problems. Paramount goals are managing software complexity and addressing issues in portable, scalable performance across a range of parallel environments, from networks of workstations to traditional massively parallel processors to clusters of symmetric multiprocessors. Our approach uses a distributed-memory (or "shared nothing") model, where we hide within parallel objects the details of communication, and the user orchestrates communication at a higher abstract level than message passing. We note that underneath these layers, data is generally communicated via message passing.

We introduce in Section 19.5.1 some sample motivating applications that lead to discussion in Section 19.5.2 of software design based on their mathematical formulations. Section 19.5.3 discusses issues in interoperability among software tools for the various phases of solving PDE-based systems. Finally, we explain in Section 19.5.4 how the flexibility in both algorithms and data structures afforded by this design enables us to better address issues in achieving high performance.

19.5.1 Sample Applications

A Linear Elliptic Example

To enable concrete discussion of these issues and begin to explain our approach, we consider the linear elliptic PDE, $\nabla^2 u = b$, in a two-dimensional domain Ω with homogeneous Dirichlet boundary conditions; details of this model are discussed in Chapter ??. In subsequent sections, we discuss additional complexities that arise in nonlinear and time-dependent problems. As discussed in Chapter 9, various parallel programming models can be used, and for such a simple model all would be well suited. We focus discussion on an SPMD approach, in which all processes execute essentially the same logic, though on a subset of the global problem domain, because this approach has proven quite effective for more complicated PDE computations and is the approach discussed in Section 19.5.2

The parallel numerical solution process begins with generating a discrete mesh of points that replaces the continuous domain of the equation. We partition the mesh and associated data at runtime across the participating processes so that each process "owns" a unique subset of the mesh and the corresponding

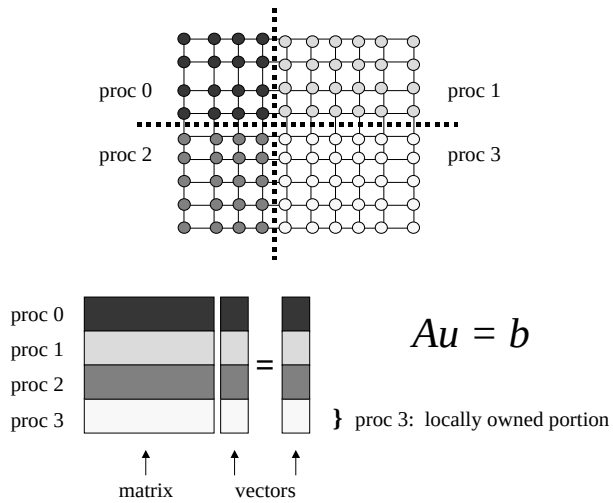

Figure 19.2: Partitioning of a rectangular mesh and a corresponding linear system so that each process "owns" a unique subset of the mesh and the corresponding unknowns of the problem, u. The matrix A and vector b are partitioned accordingly.

unknowns of the problem, as illustrated in Figure 19.2 for a regular rectangular mesh that is distributed across four processes. Partitioning and mesh generation are important phases for practical models beyond this simple example; these issues are discussed in Chapters 16 and 17, respectively. The next phase, discretization of the PDE over the mesh, typically follows the basic philosophy of "owner computes". For efficient distributed-memory computations, the process that stores mesh and associated data for a particular region of the global problem domain calculates most, though not necessarily all, of the entries of the corresponding local part of the discretized linear operator (or matrix), A, and the right-hand-side vector, b, that define the linear system $Au = b$.

As we will further discuss in Section 19.5.2, a natural abstraction for representing this mathematical problem in numerical software libraries follows this form: that is, given the inputs A and b, compute as output the approximate solution u. Note that at this level of abstraction we do not specify details about the internal representation of the matrix A or the vectors u and b. Instead, use of an abstract interface and possibly multiple underlying implementations enables the application code to remain simple — no details about storage formats need to be directly specified or even understood by beginning users, though advanced users can customize these choices. This approach also affords flexibility, because library writers can develop a variety of implementations, each of which may be appropriate in different circumstances (e.g., matrix formats that exploit sparsity and/or special structure). Moreover, when coupled with interoperabil-

ity strategies discussed in Section 19.5.3, such abstractions help to enable the seamless introduction of newly developed implementations into existing code.

A Nonlinear PDE Example

To present the approach used in PETSc, we focus on the discrete framework for an implicit PDE solution algorithm, with pseudo-timestepping to advance toward a steady state. This algorithm has the form

$$\frac{u^l}{\Delta t^l} + F(u^l) = \frac{u^{l-1}}{\Delta t^l}, \tag{19.1}$$

where $\Delta t^l \to \infty$ as $l \to \infty$, u represents a fully coupled vector of unknowns, and the steady-state solution satisfies $F(u) = 0$. We choose this problem because it is often used in large-scale CFD models, including two aerodynamics applications that we will discuss in some detail, namely, a compressible flow over an airplane wing using a structured mesh [GKMT00] and both compressible and incompressible flow using an unstructured mesh [KKS99]. While we will present computational results for these representative large-scale applications, we will illustrate software design and usage via the simpler nonlinear elliptic PDE,

$$F(u) = -\nabla^2 u - \lambda e^u = 0, \tag{19.2}$$

where $u = 0$ on the boundary of the problem domain and λ is a constant. This formulation, which is known as the Bratu problem, is taken from the MINPACK-2 test problem collection [ACM91]. The PETSc software distribution includes parallel implementations of this model that can be used to explore the software functionality.

We will explain the software design used to support pseudo-transient continuation of inexact Newton methods to advance these models toward a steady state. While this discussion will focus on the SNES (Scalable Nonlinear Equations Solvers) component, which provides a level of abstraction that is convenient for these particular applications, these design principles apply equally to the linear solvers and timestepping algorithmic components as well.

19.5.2 Mathematical Formulation

Key considerations when designing user interfaces for algorithmic software components include the following:

- What are the mathematical formulations of the target problem classes?
- What numerical algorithms will we use to solve these problems?

The combination of these two features helps to identify abstractions for components such as solvers and timesteppers as well as the mathematical operators and operands that serve as their primary inputs and outputs. As explained below, sufficiently flexible abstract interfaces can support a variety of implementations of data structures and algorithms and therefore can provide good

models for exploring algorithmic interchangeability and software interoperability among multiple tools developed by different groups. Such capabilities are critical for making high-performance numerical software adaptable to the continual evolution of parallel and distributed architectures and the research community's discovery of new algorithms that exploit their features.

Mathematical Abstractions: Vectors, Matrices, Index Sets, and Solvers

The mathematical formulations for a particular class of models present natural and intuitive abstractions that can be used in software interfaces. PETSc is built around a variety of mathematical and algorithmic objects; the application programmer works directly with these objects rather than concentrating on the underlying (rather complicated) data structures. Two of the basic abstract data objects in PETSc are *vectors* and *matrices*, which were introduced for a linear problem in Section 19.5.1. A PETSc vector (`Vec`) is an abstraction of an array of values that represent a discrete field (e.g., coefficients for the solution of a PDE), and a matrix (`Mat`) represents a discrete linear operator that maps between vector spaces. Each of these abstractions has several representations in PETSc. For example, PETSc currently provides three sequential sparse matrix data formats, four parallel sparse matrix data structures, and a dense representation; each is appropriate for particular classes of problems. In addition, the same matrix interface supports matrix-free approaches, in which matrices need not be explicitly stored, but rather certain functionalities (e.g., the application of the linear operator to a vector) can be provided in an encapsulated form.

While vectors and matrices are rather straightforward mathematical abstractions regardless of parallelism, we introduce the concept of an *index set* to deal with the need for aggregation in efficient distributed-memory computations. An index set (`IS`) is a generalization of a set of integer indices, which can be used for selecting, gathering, and scattering subsets of vector and matrix elements. The index set abstraction provides users complete control to manipulate subsets of matrix and vector elements in aggregation. While one can certainly manipulate individual matrix and vector elements, this approach is not a parallel expression and cannot exploit aggregation and other optimizations.

Built on top of this foundation are various classes of solvers, including linear (`SLES`), nonlinear (`SNES`), and timestepping (`TS`) solvers. These solvers encapsulate virtually all information regarding the solution procedure for a particular class of problems, including the local state and various options. Application codes can interface directly to any level of the numerical library hierarchy, as shown in Figure 19.3. In addition, new software tools for other facets of scientific simulations can be built using selected parts of this hierarchy. For example, the Toolkit for Advanced Optimization (TAO) [BMM99, BMM] employs PETSc infrastructure for parallel linear algebra in the construction of parallel optimization software.

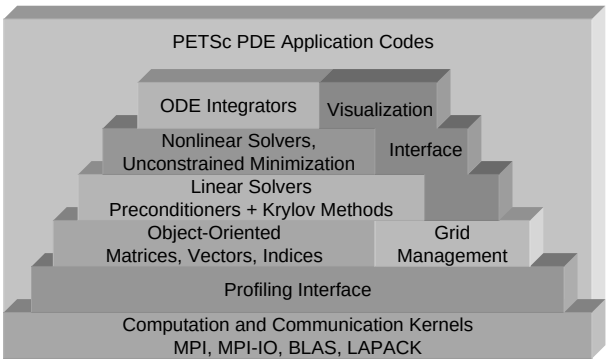

Figure 19.3: Organization of the PETSc libraries. Application codes can interface to whatever levels of abstraction are most appropriate for their needs.

Parallelism

As explained in [BGMS97], we believe that use of the message-passing model within carefully designed and implemented parallel numerical libraries is an effective approach to the problem of efficiently using large-scale distributed-memory, as well as clustered and NUMA (non-uniform memory access) shared-memory computers. This approach enables us to face the explicit tradeoffs that must be made to balance the code's performance (computational efficiency) and ease of use (programmer efficiency). Most important, this combination allows the gradual process of improving performance by the addition of new computational kernels, while retaining the remainder of the correctly working libraries and application code.

The PETSc 2.0 package uses object-oriented programming to conceal the details of the message passing, without concealing the parallelism. Because the details of communication are hidden from the user, other communication approaches besides message passing may be used as well, such as pure OpenMP or an MPI/OpenMP hybrid. A strength of the approach of message passing combined with numerical libraries is that application codes written with this model will also run well on NUMA shared-memory computers—often as well as codes custom written for a particular machine. This translation occurs because even shared-memory machines have a memory hierarchy that message-passing programs inherently respect. For the small number of code locations where taking explicit advantage of the shared memory can lead to improved performance, alternative library routines that bypass the message-passing system may easily be provided, thus retaining a performance-portable library.

In general, the data for any PETSc object (vector, matrix, mesh, linear solver, etc.) is distributed among several processes. The distribution is handled by an MPI communicator (called `MPI_Comm` in MPI syntax), which represents a

group of processes. When an object is created, for example with the commands

C interface:

```
VecCreate(MPI_Comm c,int m,Vec* v);
MatCreate(MPI_Comm c,int m,int n,Mat *A);
SLESCreate(MPI_Comm c,SLES *ls);
```

Fortran interface:

```
call VecCreate(MPI_Comm c,integer m,Vec v,integer ir)
call MatCreate(MPI_Comm c,integer m,int n,Mat A,integer ir)
call SLESCreate(MPI_Comm c,SLES ls,integer ir)
```

the first argument specifies the communicator, thus indicating which processes share the object. The creation routines are collective over all processes in the communicator.

This approach does not attempt to completely conceal parallelism from the application programmer. Rather, the user initiates combinations of sequential and parallel phases of computations, but the library handles the detailed (data-structure-dependent) message passing required during the coordination of the computations. This provides a good balance between ease of use and efficiency of implementation. Six of our main guiding design principles are listed below and discussed in detail in [BGMS97]; the first four focus on allowing the application programmer to achieve high performance, while the last two focus on ease of use of the libraries.

- Performance
  - overlapping communication and computation,
  - determining within the library the details of various repeated communications, and optimizing the resulting message passing code,
  - allowing the user to dictate exactly when certain communication is to occur, and
  - allowing the user to aggregate data for subsequent communication.
- Ease of use
  - allowing the user to work efficiently with parallel objects without specific regard for what portion of the data is stored on each processor, and
  - managing communication whenever possible within the context of higher-level operations on a parallel object or objects instead of working directly with lower-level message-passing routines.

Note that the first four principles are chiefly related to reducing the number of messages, minimizing the amount of data that needs to be communicated, and hiding the latency and limitations of the bandwidth by sending data as soon as possible, *before* it is required by the receiving processor. The six guiding principles, embedded in a carefully designed object-oriented library, enable the development of highly efficient application codes, without requiring a large effort from the application programmer.

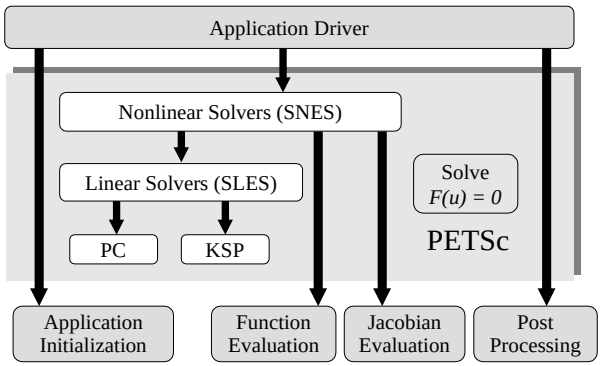

Figure 19.4: Coarsened calling tree of nonlinear PDE application, showing the user-supplied main program and call-back routines for providing the initial nonlinear iterate, computing the nonlinear residual vector at a library-requested state, and evaluating the Jacobian (preconditioner) matrix.

Implicit Solution of Nonlinear PDEs: An Application Code Perspective

The examination of families of algorithms reveals what input and output parameters are needed within abstract interfaces. For example, to solve discretized steady-state nonlinear PDEs of the form $F(u) = 0$, where $F : \Re^n \to \Re^n$ (as given in Equation (19.2)), a variety of algorithms can be used, including explicit, semi-implicit, and fully implicit techniques. We explore the interface of the SNES component of PETSc, which solves systems of this form using implicit Newton-type methods (see, e.g., [DJS83, NW99]), including line search and trust region variants. These methods can be expressed in the form

$$u_{k+1} = u_k - [F'(u_k)]^{-1} F(u_k), \quad k = 0, 1, \ldots,$$

where u_0 is an initial approximation to the solution and $F'(u_k)$ is nonsingular. In practice, the Newton iteration is implemented by the following two steps:

1. (Approximately) solve $F'(u_k)\Delta u_k = -F(u_k)$.
2. Update $u_{k+1} = u_k + \Delta u_k$.

A coarse diagram of the calling tree of a typical nonlinear PDE application appears in Fig. 19.4. The top-level user routine performs I/O related to initialization, restart, and post processing; it also calls PETSc subroutines to create data structures for vectors and matrices and to initiate the nonlinear solver. As shown by this diagram, a basic reason why the design of nonlinear equation solver libraries is fundamentally different from classical numerical linear algebra subroutine libraries such as LINPACK, EISPACK, and LAPACK is that the application code must perform certain operations for the library. The simplest such example is evaluating the nonlinear function $F(u)$ at given state vectors u;

```
SNES   snes;            /* nonlinear solver */
Mat    J;               /* Jacobian matrix */
Vec    x, f;            /* solution and residual vectors */
int    n, its;          /* problem dimension, number of iterations */
AppCtx usercontext;     /* user-defined application context */

...
/* Create matrix and vectors */
MatCreate(MPI_COMM_WORLD,n,n,&J);
VecCreate(MPI_COMM_WORLD,n,&x);
VecDuplicate(x,&f);

/* Create nonlinear solver */
SNESCreate(MPI_COMM_WORLD,SNES_NONLINEAR_EQUATIONS,&snes);

/* Set routines for evaluation of the nonlinear function and Jacobian */
SNESSetFunction(snes,f,EvaluateFunction,usercontext);
SNESSetJacobian(snes,J,EvaluateJacobian,usercontext);

/* Set runtime options */
SNESSetFromOptions(snes);

/* Solve the nonlinear system */
SNESSolve(snes,x,&its);

/* Destroy objects when finished */
SNESDestroy(snes); MatDestroy(J); VecDestroy(x); VecDestroy(f);
```

Figure 19.5: Sample SNES application code interface.

another typical requirement is approximating the associated Jacobian matrix, $F'(u)$. In addition, the software must somehow deal with application-specific data and data structures that are not known and cannot be predicted by the library writers. Auxiliary information required for the evaluation of $F(u)$ and $F'(u)$ that is not carried as part of u is communicated through PETSc via a user-defined "context" that encapsulates application-specific data. (Such information would typically include dimensioning data, mesh geometry data, physical parameters, and quantities that could be derived from the state u but are most conveniently stored instead of recalculated, such as constitutive quantities.)

Figure 19.5 illustrates the basic SNES user interface, which is both simple to use and inherently flexible. In particular, this single interface is identical for the uniprocessor and parallel cases, serves both real and complex numbers, and supports a range of different algorithms. The primary phases of solver usage are (1) instantiating the solver via the routine `SNESCreate()`; (2) specifying a vector data structure and call-back routine for evaluation of the nonlinear function $F(u)$ via `SNESSetFunction()` (and optionally the matrix data structure and associated routine for evaluation of the Jacobian $F'(u)$ via `SNESSetJacobian()`); (3) selecting various runtime options via `SNESSetFromOptions()`; (4) solving the nonlinear system via `SNESSolve()`; and (5) destroying the solver and freeing associated memory via the routine `SNESDestroy()`.

Note that the SNES user interface employs abstractions for vectors (`Vec`), matrices (`Mat`), and nonlinear solver algorithms (`SNES`). This interface reveals nothing about the particular data structures that may be used at runtime, and in fact the actual algorithms, including linesearch and trust regions variants of inexact Newton methods, are implemented in a data-structure-neutral format using these same abstractions. This data-structure-neutral approach [SG96] allows the natural storage formats for vectors and matrices to be dictated by the user's application. Since issues regarding the selection of storage formats for parallel, sparse linear algebra are usually quite complicated, this feature is critical to the software's performance.

Figure 19.6 presents sample code to evaluate the nonlinear function within Equation (19.2) in parallel on a two-dimensional regular mesh with a finite difference discretization. The problem is partitioned according to Figure 19.2, where each process owns a unique subset of the mesh and the corresponding data objects. The approach for parallel computation of the nonlinear function and Jacobian is "owner computes," with message merging and overlapping communication with computation where possible via split transactions. Each processor "ghosts" its stencil dependencies on its neighbors' data. Grid functions are mapped from a global (user-defined) ordering into contiguous local orderings, which may be designed to maximize spatial locality for cache line reuse. Scatter/gather operations are created between local sequential vectors and global distributed vectors. This example uses distributed arrays (`DA`) within PETSc to handle ghost point communication; the more general `VecScatter` tool could be used for unstructured meshes. Alternatively, one could employ tools that provide parallel discretization capabilities at higher levels of abstraction, such as Overture [BHQ]. In fact, we have recently develped "object wrappers" that allow all Overture and PETSc objects to coexist and interoperate in the same application.

Both a procedural interface (i.e., routine calls) and a command-line interface (i.e., `argc/argv` program input parameters) may be used to specify particular choices for algorithms, parameters, and data structures. The procedural interface provides a great deal of control on a usage by usage basis within a single application. For example, one can select a linesearch or trust region variant of Newton's method by calling `SNESSetType(snes,ls)` or `SNESSetType(snes,tr)`, respectively. Alternatively, these choices can be specified by the corresponding runtime option (e.g., `-snes_type [ls,tr]`); the runtime option approach applies the same rules to all queries via a database and thereby enables the user to have complete control at runtime with no extra coding. A typical usage scenario is to employ the procedural interface to indicate defaults (that may be different from those specified by the library) within a given application code, and then to use the command-line interface to override these defaults for experimentation with a variety of alternatives.

```
/*  FormFunction - Evaluates nonlinear function, F(X).

    Input Parameters:                          Output Parameter:
    snes - the SNES context                    F - vector containing
    X    - input vector                            newly evaluated
    ptr  - optional user-defined context           nonlinear function
*/
int FormFunction(SNES snes,Vec X,Vec F,void *ptr)
{
 AppCtx  *a = (AppCtx *) ptr;
 int     ierr, i, j, row, mx, my, xs, ys, xm, ym, gxs, gys, gxm, gym;
 double  two = 2.0, one = 1.0, lambda, hx, hy, hxdhy, hydhx, sc;
 Scalar  u, uxx, uyy, *x, *f;
 Vec     localX = a->localX, localF = a->localF;

 mx = a->mx;               my = a->my;               lambda = a->param;
 hx = one/(double)(mx-1);  hy = one/(double)(my-1);
 sc = hx*hy*lambda;        hxdhy = hx/hy;            hydhx = hy/hx;

 /* Scatter ghost points to local vector */
 ierr = DAGlobalToLocalBegin(a->da,X,INSERT_VALUES,localX); CHKERRQ(ierr);
 ierr = DAGlobalToLocalEnd(a->da,X,INSERT_VALUES,localX); CHKERRQ(ierr);

 /* Get pointers to vector data */
 ierr = VecGetArray(localX,&x); CHKERRQ(ierr);
 ierr = VecGetArray(localF,&f); CHKERRQ(ierr);

 /* Get local grid boundaries */
 ierr = DAGetCorners(a->da,&xs,&ys,PETSC_NULL,&xm,&ym,PETSC_NULL); CHKERRQ(ierr);
 ierr = DAGetGhostCorners(a->da,&gxs,&gys,PETSC_NULL,&gxm,&gym,
        PETSC_NULL); CHKERRQ(ierr);

 /* Compute function over the locally owned part of the grid */
 for (j=ys; j<ys+ym; j++) {
   row = (j - gys)*gxm + xs - gxs - 1;
   for (i=xs; i<xs+xm; i++) {
     row++;
     if (i == 0 || j == 0 || i == mx-1 || j == my-1) {f[row] = x[row]; continue;}
     u = x[row];
     uxx = (two*u - x[row-1] - x[row+1])*hydhx;
     uyy = (two*u - x[row-gxm] - x[row+gxm])*hxdhy;
     f[row] = uxx + uyy - sc*exp(u);
   }
 }

 /* Restore vectors */
 ierr = VecRestoreArray(localX,&x); CHKERRQ(ierr);
 ierr = VecRestoreArray(localF,&f); CHKERRQ(ierr);

 /* Insert values into global vector */
 ierr = DALocalToGlobal(a->da,localF,INSERT_VALUES,F); CHKERRQ(ierr);
 return 0;
}
```

Figure 19.6: Sample parallel nonlinear function evaluation code for equation (19.2), using a finite difference discretization on a two-dimensional regular mesh and distributed arrays for ghost point communication.

19.5.3 Composability and Interoperability

As discussed in Section 19.3.1, the high-fidelity multiphysics applications of interest within high-performance scientific computing often require the combined use of software tools that encapsulate the expertise of multidisciplinary research teams. Current-generation software tools have demonstrated good success in direct pairwise interfaces, whereby one tool directly calls another by using well-defined interfaces that are known at compile time. For example, we have developed two-way interfaces between PETSc and PVODE, which provides higher-order, adaptive ODE schemes and robust nonlinear solvers [Hin]. However, more flexible and dynamic capabilities are needed than predefined interfaces that use a succession of subroutine calls. This is especially important because we must support incremental shifts in parallel algorithms and programming paradigms that inevitably occur during the lifetimes of scientific application codes.

Consequently, various research groups within the high-performance computing community are exploring the ideas of *component programming*, based on encapsulating units of functionality and providing a meta-language specification of their interfaces (see, e.g., [Szy98, BDH+98]). Component-based software development can be considered an evolutionary step beyond object-oriented design. Object-oriented techniques have been quite successful in managing the complexity of modern software, but they have not resulted in significant amounts of cross-project code reuse. Sharing object-oriented code is difficult because of language incompatibilities, the lack of standardization for interobject communication, and the need for compile-time coupling of interfaces. Component-based software development addresses issues of language independence—seamlessly combining components written in different programming languages—and component frameworks define standards for communication among components.

The Common Component Architecture (CCA) Forum, whose current membership is drawn from various Department of Energy national laboratories and collaborating academic institutions, is working to specify a component architecture for high-performance scientific computing [Com, AGG+99]. We are currently incorporating new features within the PETSc software to enable compliance with this evolving specification.

19.5.4 Performance Issues

As discussed by [AGKS99], achieving high sustained performance for PDE-based simulations involves three aspects. The first is a scalable algorithm in the sense of convergence rate; the second is good per-processor performance on contemporary cache-based microprocessors; and the third is a scalable implementation, in the sense of time per iteration as the number of processors increases. This section demonstrates that the flexible software design presented in this chapter enables application codes to address all three of these issues and to avoid premature optimization for particular algorithmic and data structure choices by experimenting with a range of options for realistic problems.

Algorithmic Experimentation

Now that we have covered the basic principles of design and seen what some of the issues are for parallel PDE computations, we examine a specific application to demonstrate how this approach enables investigation of open research issues. In particular, we explore the standard three-dimensional aerodynamics test case of transonic flow over an ONERA M6 wing using the frequently studied parameter combination of a freestream Mach number of 0.84 with an angle of attack of 3.06^o. The robustness of solution strategies is particularly important for this model because of the so-called λ-shock that develops on the upper wing surface. The basis for our implementation, as discussed in [GKMT00], is a legacy sequential Fortran 77 code by Whitfield and Taylor [WT91] that uses a mapped structured C-H mesh. This application demonstrates the use of the nonlinear solvers within SNES in the legacy context, where we retain the original code's discretization as embodied in flux balance routines for steady-state residual construction and finite-difference Jacobian construction. The function evaluations are undertaken to second order in the upwinding scheme, and the Jacobian matrix (used mainly as a preconditioner) is evaluated to first order. We parallelize the logically regular, mapped mesh using the distributed array tools of PETSc.

We consider Newton-Krylov-Schwarz methods, which combine a Newton-Krylov method with a Schwarz-based preconditioner. From a computational point of view, one of the most important characteristics of a Krylov method for the linear system $Ax = b$ is that information about the matrix A needs to be accessed only in the form of matrix-vector products in a relatively small number of carefully chosen directions. Newton-Krylov methods are suited for nonlinear problems in which it is unreasonable to compute or store a true, full Jacobian, where the action of A can be approximated by discrete directional derivatives. However, if the Jacobian A is ill-conditioned, the Krylov method will require an unacceptably large number of iterations. The system can be transformed into the equivalent form $B^{-1}Ax = B^{-1}b$ through the action of a preconditioner, B, whose inverse action approximates that of A, but at smaller cost. It is in the choice of preconditioning where the battle for low computational cost and scalable parallelism is usually won or lost. In Schwarz preconditioning methods (see, e.g., [SBG96]), the operator is introduced on a subdomain-by-subdomain basis through a conveniently computable approximation to a local Jacobian. Such Schwarz-type preconditioning provides good data locality for parallel implementations over a range of parallel granularities, allowing significant architectural adaptability.

Figure 19.7 shows a sample script that can be used to automate experimentation with this hierarchy of tunable algorithms. The script demonstrates the use of both linesearch and trust region variants of Newton's method on various numbers of processors. Several Krylov methods are considered, including GMRES, BiCGStab, and transpose-free QMR, in conjunction with additive Schwarz preconditioners with various degrees of overlap. This script facilitates the investigation of which preconditioning and Krylov methods are most effective

for particular problem sizes and processor configurations. Additional runtime options could also be invoked to investigate a range of other issues, including linear and nonlinear convergence parameters, blocked matrix data structures, and derivative computations via sparse finite differences and automatic differentiation.

```
#! /bin/csh
#
# Sample script: Experimenting with nonlinear solver options.
# Can be used with, e.g., petsc/src/snes/examples/tutorials/ex5.c
#
foreach np (8 16 32 64)                      # number of processors
  foreach snestype (ls tr)                   # nonlinear solver
    foreach ksptype (gmres bcgs tfqmr)       # Krylov solver
       foreach overlap (1 2 3 4)             # level of overlap for ASM
          echo '****** Beginning new run ******'
          mpirun -np $np ex2 -snes_type $snestype -ksp_type $ksptype \
             -pc_type asm -pc_asm_overlap $overlap
       end
    end
  end
end
```

Figure 19.7: Sample script for Newton-Krylov-Schwarz algorithmic experimentation.

Preconditioner quality dramatically affects the overall efficiency of the parallel Newton-Krylov-Schwarz methodology, as demonstrated in Figure 19.8 for various degrees of overlap for the restricted additive Schwarz method (RASM) [CS99], which eliminates interprocess communication during the interpolation phase of the additive Schwarz technique. The graphs within these figures compare convergence rate (in terms of relative residual norm) with both nonlinear iteration number (left-hand graph) and time (right-hand graph) for a mesh of dimension $98 \times 18 \times 18$ with five degrees of freedom per node, on 16 processors of an IBM SP2. All runs plotted in this figure use preconditioned restarted GMRES with a Krylov subspace of maximum dimension 30 and a fixed relative convergence tolerance of 10^{-2}; each processor hosts a single preconditioner block, which is solved via point-block ILU(0). We see that for this model, two-cell overlap provides a good balance in terms of power and cost. Less overlap trades off cheaper cost per iteration for a preconditioner that does not allow the nonlinear iterations to converge as rapidly, while more overlap is costly to apply and does not contribute to faster nonlinear convergence. Similar behavior was observed for other problem sizes and processor configurations, even when using different criteria to determine linear inner iteration convergence.

Data Structures and Orderings for Fast Local Performance

A key consideration in algorithms and data structures is the management of multilevel memory hierarchies. To demonstrate some of these issues, we consider another application, FUN3D, which is a tetrahedral vertex-centered unstruc-

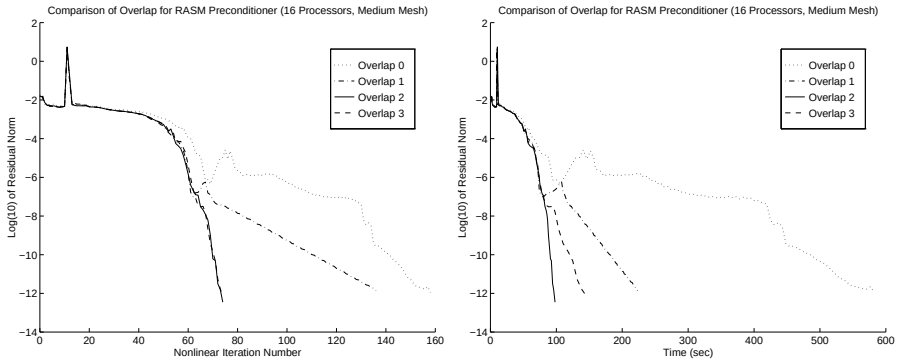

Figure 19.8: Comparison of four domain-decomposed preconditioners: subdomain-block Jacobi and restricted additive Schwarz with overlap of 1, 2, and 3 cells. All methods solve point-block ILU(0) on 16 subdomains on an IBM SP2.

tured mesh code originally developed for uniprocessors by W. K. Anderson of the NASA Langley Research Center for compressible and incompressible Euler and Navier-Stokes equations [AB94]. FUN3D uses a control volume discretization with variable-order Roe schemes for approximating the convective fluxes and a Galerkin discretization for the viscous terms. The application was parallelized using the `VecScatter` tools within PETSc for ghost point communication and the nonlinear solvers within SNES [KKS97].

We can view PDE computations predominantly as a mix of loads and stores with embedded floating-point operations (flops) [AGKS99]. Since flops are cheap relative to memory references, we concentrate on minimizing the memory references and emphasize strong sequential performance as one of the factors needed for efficient aggregate performance. Data storage patterns for primary and auxiliary fields should adapt to hierarchical memory through (1) interlacing, (2) structural blocking degrees of freedom that are defined at the same point in point-block operations, and (3) reordering of edges for reuse of vertex data. Interlacing allows efficient reuse of cached operands, since components at the same point interact more intensely with each other than do the same fields at other points. Similarly, blocking reduces the number of loads significantly and enhances reuse of data items in registers. Also, edge-reordering for vertex reuse reflects the fact that nearby points interact more intensely than distant points. Applying these techniques within FUN3D required whole-program transformations of certain loops of the original vector-oriented application but, as shown in Figure 19.9, raised the per-processor performance by a factor of between 2.5 and 7, depending on the microprocessor and optimizing compiler. We note that the use of the abstract interface for PETSc matrix assembly enabled the change from a compressed, sparse row point storage format to the block variant without changing a single line of the matrix assembly code.

Because of the cost and difficulty of architectural tuning for new environ-

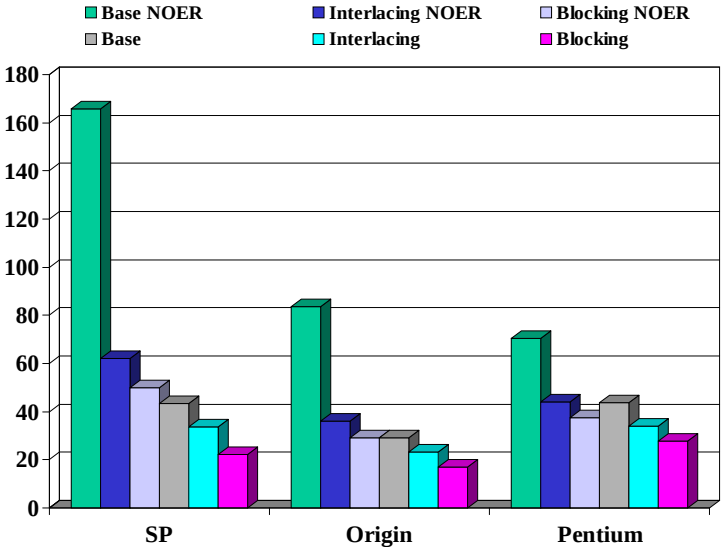

Figure 19.9: Effect of cache optimizations of the average execution time for one nonlinear iteration of the FUN3D application. *Base* denotes the case without any optimizations, and *NOER* denotes no edge reordering. The performance improves by a factor of about 2.5 on the Pentium and 7.5 on the IBM SP. The processor details are 120 MHz IBM SP (P2SC "thin", 128 KB L1), 250 MHz Origin2000 (R1000, 32 KB L1, and 4 MB L2), and 400 MHz Pentium II (running Windows NT 4.0, 16 KB L1, and 512 KB L2).

ments, some recent efforts have focused on automating this process for numerical kernels. In particular, ATLAS (Automatically Tuned Linear Algebra Software) [WD] and PHiPAC (Portable High Performance ANSI C) [BAV+] are packages for automatically producing high-performance BLAS, in particular matrix-matrix-multiplication routines, for machines with complicated memory hierarchies and functional units.

Scalability

Having first assured attention to good per-processor performance for the FUN3D application, we are now ready to discuss the scalability of this aerodynamic model. In Figure 19.10 we demonstrate several metrics of the code's parallel scalability, which uses pseudo-timestepping and the Newton-Krylov-Schwarz implementations in PETSc, for a fixed-size mesh with 2.8 million vertices running on up to 1024 Cray T3E processors. We see that the implementation efficiency of parallelization (i.e., the efficiency on a per-iteration basis) is 82 percent in going from 128 to 1024 processors. The number of nonlinear iterations is also

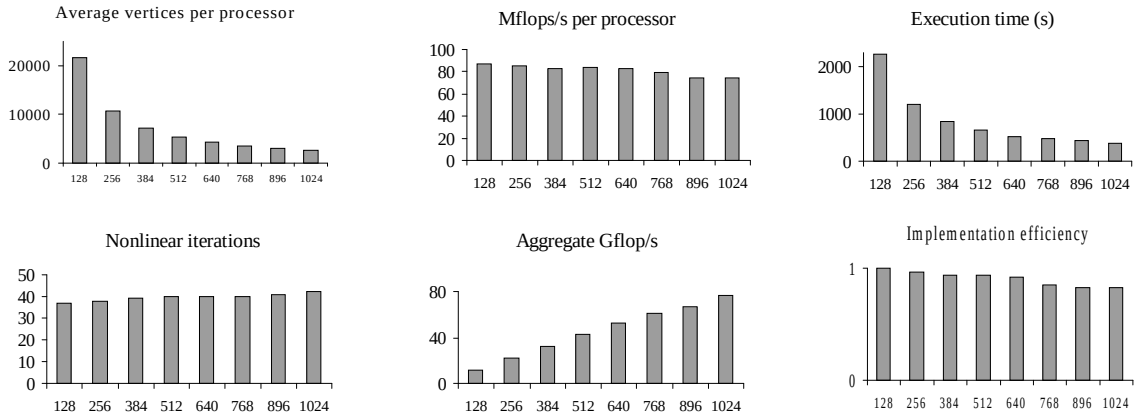

Figure 19.10: Parallel performance of the FUN3D application for a fixed-size mesh of 2.8 million vertices (over 11 million unknowns) run on up to 1024 Cray T3E 600 MHz processors.

fairly flat over the same eightfold range of processors (rising from 37 to 42), reflecting reasonable algorithmic scalability. This is much less serious degradation than predicted by the linear elliptic theory (see [SBG96]); pseudo-timestepping, required by the nonlinearity, is responsible. The overall efficiency is the product of the implementation efficiency and the algorithmic efficiency. The computational rates per processor are also close to flat over this range, even though the relevant working sets in each subdomain vary by nearly a factor of eight. This emphasizes the requirement of good serial performance for good parallel performance.

19.6 Software for PDEs

We now provide a brief overview of various software tools for the scalable solution of partial differential equations.

- **DAGH** [PB] - DAGH (Distributed Adaptive Grid Hierarchy) provides a program development infrastructure for the implementation of solutions of partial differential equations using adaptive mesh refinement algorithms.
- **Diffpack** [Dif, BL97] - Diffpack is an object-oriented framework for solving PDEs.
- **DOUG** [HS] - DOUG (Domain decomposition On Unstructured Grids) is a black box parallel iterative solver for finite element systems arising from elliptic partial differential equations.

- **FFTW** [Fft] - FFTW is a collection of fast Fourier transform routines, including routines for parallel computers. FFTs are often used in solving certain classes of linear PDEs, and can be used as preconditioners for more general PDEs.

- **KeLP** [Bad, FBK96] - KeLP (Kernel Lattice Parallelism) is a framework for implementing portable scientific applications on distributed-memory parallel computers. It is intended for applications with special needs, in particular, that adapt to data-dependent or hardware-dependent conditions at run time.

- **MUDPACK** [Ada] - MUDPACK includes a suite of portable Fortran programs that automatically discretize and use multigrid techniques to generate second- and fourth-order approximations to elliptic PDEs on rectangular regions.

- **Overture** [BHQ, DLBQ99] - Overture is an object-oriented code framework for solving PDEs; it provides a portable, flexible software development environment for applications that involve the simulation of physical processes in complex moving geometry.

- **Parallel ELLPACK** [HRH, HKM+95] - Parallel ELLPACK is a problem solving environment for PDE-based applications.

- **PARASOL** [PAR] - PARASOL is an integrated environment for parallel sparse matrix solvers. PARASOL is written in Fortran 90 and uses MPI for communication.

- **PETSc** [BGMS97, BGMS] - PETSc (Portable, Extensible Toolkit for Scientific computing) is a collection of tools for the parallel numerical solution of PDEs and related problems.

- **POOMA** [ABC+95, POO] POOMA (Parallel Object-Oriented Methods and Applications) is an object-oriented framework for applications in computational science requiring high-performance parallel computers.

- **SAMRAI** [KGHS, HK99] - SAMRAI is an object-oriented code framework that provides general and extensible software support for rapid prototyping and development of parallel structured adaptive mesh refinement applications.

- **UG** [UG , BBJ+97] - UG (Unstructured Grids) is a flexible software tool for the numerical solution of partial differential equations on unstructured meshes in two and three space dimensions using multigrid methods.

- **VECFEM** [Gro] - VECFEM is a package for the solution of nonlinear boundary value problems by the finite element method.

Additional pointers may be available through the following on-line resources:

- **MGNet** [Dou] - MGNet is a repository for information related to multigrid, multilevel, multiscale, aggregation, defect correction, and domain decomposition methods, including links to software packages.
- **NHSE** [Nat] - The National High-Performance Software Exchange is a distributed collection of software, documents, data, and information of interest to the high performance and parallel computing community.

19.7 Observations and Recommendations

As discussed in Chapter 3, future computing technology will likely be characterized by highly parallel, hierarchical designs. This trend in design is a fairly straightforward consequence of two other trends: a desire to work with increasingly large data sets at increasing speeds and the imperative of cost-effectiveness. Fortunately, data use in most PDE-based applications has sufficient temporal and spatial locality to map reasonably well to distributed- and hierarchical-memory systems. To achieve good performance, this locality can be exploited by a combination of the application programmer at the algorithmic level, the system software at the compiler and runtime levels, and the hardware.

This chapter presented some ideas for addressing these issues in PDE software at the level of numerical library writers and application programmers. In particular, we discussed how organizing applications around the mathematics of models enables the writing of applications that can be run without change with a wide variety of different algorithms and data structures. This facilitates exploiting parallelism, managing complexity within the application, and effectively using the available computing resources. Using these techniques, applications have run scalably on thousands of processors, achieving performance in the teraflop range. We conclude with a few additional recommendations for application scientists.

- Design application codes around abstract concepts, not particular algorithms or data structures. Expect the best algorithms to change over the lifetime of an application.
- Take advantage of modern programming languages; e.g., use features of Fortran 90 rather than minimalistic Fortran 77.
- Use programming models that offer portable performance, such as MPI or OpenMP. Use vendor-specific features or extensions only when the benefit clearly outweighs the loss of portability.
- Communicate and compute on aggregates, not individual elements.
- Use libraries whenever possible; and when libraries do not provide the needed functionality, contact the authors with suggestions and recommendations.
- Give the largest possible problem to the numerical library. For example, if the library offers suitable nonlinear solvers as well as linear solvers, use the

nonlinear solvers rather than building a simple nonlinear iteration yourself and using the library's linear solvers. This approach gives the library the best opportunity to maximize performance (see Section 19.3.3).

Acknowledgments

We were supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Bibliography

[AB94] W. K. Anderson and D. L. Bonhaus. An implicit upwind algorithm for computing turbulent flows on unstructured grids. *Computers and Fluids*, 23:1–21, 1994.

[ABC+95] S. Atlas, S. Banerjee, J. C. Cummings, P. J. Hinker, M. Srikant, J. V. W. Reynders, and M. Tholburn. POOMA: A high-performance distributed simulation environment for scientific applications. In *Supercomputing '95 Proceedings*, December 1995.

[ACM91] Brett M. Averick, Richard G. Carter, and Jorge J. Moré. The MINPACK-2 test problem collection. Technical Report ANL/MCS-TM-150, Argonne National Laboratory, 1991.

[Ada] John C. Adams. MUDPACK Web page. See `http://-www.scd.ucar.edu/css/software/mudpack`.

[AGG+99] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. C. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of High Performance Distributed Computing*, 1999. To appear (also Argonne National Laboratory Mathematics and Computer Science Division preprint P759-0699).

[AGKS99] W. K. Anderson, W. D. Gropp, D. K. Kaushik D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh CFD application. In *Proceedings of SC 99*, 1999. To appear.

[Bad] Scott Baden. KeLP Web page. See `http://www-cse.ucsd.edu/-groups/hpcl/scg/kelp/`.

[BAV+] J. A. Bilmes, K. Asanovic, R. Vudoc, S. Iyer, J. Demmel, C. Chin, and D. Lam. PHiPAC Web Page. See `http://-www.icsi.berkeley.edu/~bilmes/phipac/`.

[BBJ+97] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuss, H. Rentz-Reichert, and C. Wieners. UG – a flexible software toolbox for

solving partial differential equations. *Computing and Visualization in Science*, 1997.

[BDH+98] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, František Plášil, Gustave Pomberger, Wolfgang Pree, Michael Stal, and Clemens Szyperski. What characterizes a (software) component? *Software – Concepts and Tools*, 19:49–56, 1998.

[BGMS] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc Web page. See `http://www.mcs.anl.gov/petsc`.

[BGMS97] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[BGMS00] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.28, Argonne National Laboratory, March 2000.

[BHQ] David L. Brown, William D. Henshaw, and Daniel J. Quinlan. Overture Web page. See `http://www.llnl.gov/CASC/Overture`.

[BL97] A. M. Bruaset and H. P. Langtangen. A comprehensive set of tools for solving partial differential equations: Diffpack. In *Numerical Methods and Software Tools in Industrial Mathematics*, pages 61–90. Birkhauser Press, 1997.

[BMM] Steve Benson, Lois Curfman McInnes, and Jorge Moré. Toolkit for Advanced Optimization (TAO) Web page. See `http://www.mcs.anl.gov/tao`.

[BMM99] Steve Benson, Lois Curfman McInnes, and Jorge Moré. GPCG: A case study in the performance and scalability of optimization algorithms. Technical Report ANL/MCS-P768-0799, Mathematics and Computer Science Division, Argonne National Laboratory, 1999.

[Com] Common Component Architecture Forum. See `http://-www.acl.lanl.gov/cca-forum`.

[CS99] X.-C. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM J. Scientific Computing*, 21:792–797, 1999.

[DGK84] J. J. Dongarra, F. G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, January 1984.

[Dif] Diffpack Web page. See `http://www.nobjects.com/Diffpack/`.

[DJS83] J. E. Dennis Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

[DLBQ99] William D. Henshaw D. L. Brown and Daniel J. Quinlan. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pages 215–224. SIAM, 1999.

[Dou] Craig C. Douglas. MGNet Web page. See `http://www.mgnet.org`.

[FBK96] Stephan J. Fink, Scott B. Baden, and Scott R. Kohn. Flexible communication mechanisms for dynamic structured applications. In *Irregular '96*, 1996.

[Fft] FFTW Web page. `http://www.fftw.org/`.

[GKKS99] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Toward realistic performance bounds for implicit CFD codes. In A. Ecer et al., editor, *Proceedings of Parallel CFD'99*. Elsevier, 1999. To appear.

[GKMT00] William D. Gropp, David E. Keyes, Lois Curfman McInnes, and M. D. Tidriri. Globalized Newton-Krylov-Schwarz algorithms and software for parallel implicit CFD. *Int. J. High Performance Computing Applications*, 2000. To appear (also ICASE Technical Report 98-24).

[Gro] Lutz Grosz. VECFEM Web Page. See `http://-wwwmaths.anu.edu.au/~vecfem/`.

[Hea97] Michael T. Heath. *Scientific Computing: An Introductory Survey*. McGraw Hill, 1997.

[Hin] A. Hindmarsh et al. PVODE Web page. See `http://www.llnl.gov/CASC/PVODE`, Lawrence Livermore National Laboratory.

[HK99] Richard Hornung and Scott Kohn. The use of object-oriented design patterns in the SAMRAI structured AMR framework. In *Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*, pages 235–244, 1999.

[HKM+95] E. N. Houstis, S. B. Kim, S. Markus, P. Wu, N. E. Houstis, A.C. Catlin, S. Weerawarana, and T.S. Papatheodorou. Parallel ELLPACK elliptic PDE solvers. In *Proceedings of INTEL Supercomputer User's Group Conference*, Albuquerque, NM, 1995.

[HKM98] M. E. Hayder, D. E. Keyes, and P. Mehrotra. A comparison of the PETSc library and HPF implementations of an archetypal PDE computation. *Advances in Engineering Software*, 29:415–424, 1998.

[HRH] Elias Houstis, John Rice, and Apostolos Hadjidimos. Parallel ELLPACK Web page. See `http://www.cs.purdue.edu/research/-cse/pellpack/`.

[HS] Mark J. Hagger and Linda Stals. DOUG Web Page. See `http://-www.maths.bath.ac.uk/~parsoft/doug/`.

[IJCL96] C. S. Ierotheou, S. P Johnson, M. Cross, and P. F. Leggett. Computer aided parallelization tools (CAPTools) - conceptual overview and performance on the parallelization of structured mesh codes. *Parallel Computing*, 22:197–226, 1996.

[KGHS] Scott Kohn, Xabier Garaiza, Rich Hornung, and Steve Smith. SAMRAI Web page. See `http://www.llnl.gov/CASC/SAMRAI`.

[KKS97] D. K. Kaushik, D. E. Keyes, and B. F. Smith. On the interaction of architecture and algorithm in the domain-based parallelization of an unstructured grid incompressible flow code. In J. Mandel et al., editor, *Proceedings of the 10th International Conference on Domain Decomposition Methods*, pages 311–319. Wiley, 1997.

[KKS99] D. K. Kaushik, D. E. Keyes, and B. F. Smith. Newton-Krylov-Schwarz methods for aerodynamic problems: Compressible and incompressible flows on unstructured grids. In C.-H. Lai et al., editor, *Proceedings of the 11th International Conference on Domain Decomposition Methods*. Domain Decomposition Press, Bergen, 1999. To appear.

[KLS+94] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, and M. E. Zosel. *The High Performance Fotran Handbook*. MIT Press, 1994.

[Kon00] Alice E. Koniges, editor. *Industrial Strength Parallel Computing*. Morgan Kaufmann Publishers, San Francisco, 2000.

[KSV97] David E. Keyes, Ahmed Sameh, and V. Venkatakrishnan, editors. *Parallel Numerical Algorithms*. Kluwer Academic Publishers, the Netherlands, 1997.

[MM94] K. W. Morton and D. F. Mayers. *Numerical Solution of Partial Differential Equations*. Press Syndicate of the University of Cambridge, 1994.

[MSC] MSC Software Corporation. NASTRAN Web page. See `http://www.mechsolutions.com/products/nastran/`.

[Nat] National High-Performance Software Exchange Web page. See `http://www.nhse.org`.

[NW99] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer-Verlag, New York, 1999.

[PAR] PARASOL Web page. See `http://www.genias.de/projects/-parasol`.

[PB] M. Parashar and J. C. Browne. DAGH Web page. See `http://-www.caip.rutgers.edu/~parashar/DAGH/`.

[POO] POOMA Web page. See `http://www.acl.lanl.gov/pooma`.

[QV99] Alfio Quarteroni and Alberto Valli. *Domain Decomposition Methods for Partial Differential Equations*. Oxford Science Publications, Oxford, 1999.

[Saa96] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston, 1996.

[SBG96] Barry F. Smith, Petter Bjørstad, and William Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[SG96] Barry F. Smith and William D. Gropp. The design of data-structure-neutral libraries for the iterative solution of sparse linear systems. *Scientific Programming*, 5:329–336, 1996.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, New York, 1998.

[Thi93] Thinking Machines Corporation. *Users Manual for CM-Fortran*. Thinking Machines Corporation, 1993.

[UG] UG Web Page. See `http://cox.iwr.uni-heidelberg.de/~ug/`.

[WD] R. Clint Whaley and Jack Dongarra. ATLAS Web Page. See `http://www.netlib.org/atlas/`.

[WT91] D. L. Whitfield and L. K. Taylor. Discretized Newton-relaxation solution of high resolution flux-difference split schemes. In *Proceedings of the AIAA Tenth Annual Computational Fluid Dynamics Conference*, pages 134–145, 1991.