Charon toolkit for parallel, implicit structured-grid computations: Literature survey and conceptual design

Rob F. Van der Wijngaart
MRJ Technology Solutions
NASA Ames Research Center
Moffett Field, CA 94035

1 Introduction

Writing message-passing codes is a tedious task for all but the simplest applications. Other parallelization tools are becoming available, such as compiler directives for (virtual) shared-memory systems, and parallelizing compilers like SUIF [30], KAP [54], and FORGE Explorer [56], but these are usually not up to the task of large scale parallelization. High Performance Fortran (HPF) [36] holds the promise of efficient parallelization of certain classes of problems, but the language lacks expressivity regarding control structures and data distributions, and writing efficient HPF compilers has proven to be a daunting task. For the majority of scientific applications message passing is still the method of choice.

The advantages of message passing are clear. The user has complete control over exploitation of concurrency and over data distribution. The separation of address spaces between processors and the explicit message passing calls provide a simple programming model. They also enable tuning by the application programmer, since the costly communications are completely managed by the programmer. The major disadvantage, contrary to common belief, is *not* the bookkeeping associated with the placement of message-passing calls. In a typical application program the number of lines of program text involving communications is only a small fraction of the total. What makes message passing truly cumbersome in most scientific computing programs is the explicit management of the domain decomposition, i.e. the restriction of data structures and the associated operations to individual processors[1]. The processor sees only a small 'window' of the entire distributed data structure. Another problem is that message-passing programs cannot be developed gradually by piecemeal conversion of a serial code to a parallel code. Domain decompositions are all-pervasive, and the entire program must be converted all at once. This puts message-passing at a distinct disadvantage compared to the shared-memory paradigm, which allows gradual conversion of legacy code by insertion of parallelizing directives.

Charon provides a vehicle that enables easy development of efficient message-passing programs. It is a toolkit that aids engineers in developing scientific programs for structured-

[1] We will speak of *processors* throughout this document, although sometimes *processes* would be more appropriate.

grid applications to be run on MIMD parallel computers. Both legacy code conversion and development from scratch are supported. Charon is neither language, nor problem-solving environment. It does not provide a compiler, does no dependency analysis. There is no graphical user interface. Instead, Charon provides a small set of user-callable functions that create, manipulate, and interrogate domain decompositions and the distributed variables defined on them. Emphasis is on rapid program development and debugging, and subsequent piecemeal performance tuning. To support this approach, functions that manipulate distributed variables are provided at several levels of abstraction. The highest level is the simplest to use, but also the least efficient. It is a device that emulates serial program execution on distributed data. Salient features are a simulated single program counter, application of the owner-computes rule, and automatic synchronization. The intermediate level is still easy to use, but more efficient. It consists of a collection of communication routines, augmented with functions that control granularity, synchronization, and concurrency, and that allow relaxation of the owner-computes rule. Indexing of variables is global. The lowest level involves local indexing of variables and (possibly) explicit message passing; it offers the highest efficiency. Calls at all three levels can be freely mixed, allowing a gradual transition from low to high efficiency.

Charon is intended to help solve *difficult* structured-grid problems. It offers support for implicit numerical methods that involve recurrences that cannot be expressed in a data-parallel way. It also offers support for very general domain decompositions. Since Charon provides more flexibility than most other systems, it is necessarily more complex. Problems that do not need its power—most notably explicit methods—may be better solved using some of the systems described in Section 2, such as KeLP [22], OVERTURE [9], or PETSc [5].

2 Survey of related projects

We restrict our investigation to scientific-computing problems that are best solved using domain decomposition. There are several ways in which parallelization aids can be classified. First, the data distribution can be explicit or implicit. If it is implicit, all the user does is indicate which program segments can be executed in parallel; interaction is limited to giving hints regarding concurrency. Since the user does not know which statements are executed by which processor, nor where data resides in case of a distributed-memory system, this kind of parallelization aid gives an image of shared memory. Vectorization and multi-tasking compiler directives defined by Cray Research [57], and parallelization directives defined by Advanced Parallel Research (FORGE) [56], Kuck and Associates (KAP/Pro Toolset) [54], Silicon Graphics (MIPSpro Power Fortran (X3H5 compliant) and C (pragma-based directives)) [53], and in the draft report by the Parallel Computing Forum X3H5 committee [42] fit this description. These directives have the form of structured comments, and are ignored by non-parallelizing compilers. If no directives are given, some parallelization or vectorization may still occur if the source code is simple enough to be analyzed by the compiler. Depending on the level of sophistication of the compiler, interprocedural dependency analysis may discover coarse-grain parallelism [56, 30, 54]. Completely automatic parallelization—for shared-memory systems only—is obtained by compilers such as SUIF [30] and KAP [54] by Kuck and Associates.

The advantage of these tools is that they allow quick parallelization of legacy codes. Either nothing needs to be done at all, or only some structured comments are inserted. However, the disadvantages are several. In the case of a non-uniform memory access (NUMA) computer, the lack of control over where data actually resides can lead to severe performance degradation. In many scientific computing programs there is not a single optimal data distribution for the entire code. Under manual control, the user specifies a domain decomposition that is a reasonable compromise. An automatically generated decomposition can be extremely poor for certain parts of the program, leading to frequent remote memory accesses and/or page migrations. Another problem associated with implicit domain decompositions is the lack of control over and expressibility of parallelism. At best, the user can specify that a program structure—typically a loop—can be executed in parallel (`DO PARALLEL` [42]). But often it is more appropriate to express parallelism across loop nests, rather than across simple loops. At worst, there is no control at all, and the compiler may extract parallelism at the wrong level of granularity. Interprocedural analysis may help, but has its limitations, since not all dependencies can be resolved at compile time [29]. Regarding the lack of expressivity of parallelizing compilers, one of the most useful control structures in parallel programming, the pipeline, cannot be expressed without making the domain decomposition explicit. The only directives that a user can supply indicate data parallelism.

But making the domain decomposition explicit is not always sufficient. HPF [36] and the related projects HPC++ [35], Vienna Fortran [14], C* [39, pp. 450–459], Annai [16], CM Fortran [55], PC++/Sage++ [26], Fortran D [25] (augmented with the CHAOS runtime support procedures [43]), Mentat [27], the explicit parallel Fortran syntax bindings from the X3H5 document [42], etc., allow the user to specify or to suggest how to distribute the data. But they do not provide control structures to express parallelism explicitly beyond the `DO PARALLEL` [42], `FORALL, INDEPENDENT` [36], or equivalent constructs. Vienna Fortran, Annai and Fortran D fix some of the deficiencies of HPF, for example by making more explicit which (virtual) processor executes which set of statements in a parallel loop (through the `ON` clause in Fortran D and Vienna Fortran), by providing more general data distributions (through the `BLOCK_GENERAL` distribution in Annai's Parallelization Support Tool (PST), and through user-defined mappings in Fortran D, Vienna Fortran, and PST), by defaulting to private, rather than shared data structures (Fortran D, Annai), and by offering reductions in parallel loops within the language, rather than through the awkward `EXTRINSIC(HPF_LOCAL)` mechanism. But since these languages do not allow the user to specify where and how communications should occur, granularity is often unnecessarily fine, unless hand tuning is applied [31]. Moreover, the programs are strictly SPMD, and any distributed data is distributed over the entire processor set. SAGE, Mentat and HPC++ have the added disadvantage that they require the programmer to use C++, which is not the language of choice for most numerical analysts. HPF, CM Fortran, C* and some other systems provide the device of virtual processors, which are mapped to physical processors by the compiler. While this sometimes constitutes a programming convencience, preventing the user from identifying processors for target data distribution can severely degrade performance, and may even affect correctness (for example when using `EXTRINSIC(HPF_LOCAL)`).

The problem with all these parallelization tools, as with the parallelizing compilers, is that they do not provide a mechanism to express task parallelism. All processors may 'attack' a `FORALL` construct in HPF, but there is no way of assigning some processors to a certain

task, while others proceed with another. Only data parallelism can be expressed explicitly. The reason for this is that explicit task parallelism requires the placement of communications between interdependent tasks, something that tool writers have traditionally tried to avoid. All task parallelism recognized by parallelizing compilers—and there are several that are capable of extracting some—is implicit, and the communications are inferred by the compiler. As an example we mention the pipeline feature in the Fortran D compiler, which usually produces inefficient code [31], because of the difficulty of automatically choosing the proper pipeline grouping factor. Another parallelization package capable of recognizing recurrences that can be resolved using pipelines is CAPTools [34], which uses a dialogue with the user and sophisticated data dependency analysis to derive the parallel code. CAPTools–like Fortran D and some of the proprietary and public domain HPF (pre-)compilers, for example the ADAPTOR/Bouclettes system [8]—automatically detects certain common data dependencies and inserts the proper control structures and message passing calls. The result is a translated source code program that can be edited by the user for further fine tuning. But CAPTools still has some of the limitations of most other parallelizing tools, namely that data distributions are essentially the same as those of HPF, that the programming model is strictly SPMD, and that the quality of the parallel code produced depends on the capability of the underlying dependency analysis engine to recognize complex data dependencies, as well as on its judicious selection of the proper granularity. Forge Explorer [56] resembles CAPTools in its capability to parallelize programs once the user has indicated interactively how data should be distributed, but is more restricted in its domain decompositions (only one array dimension may be partitioned), and, again, only data parallelism is supported.

Attempts to provide an expression mechanism for task parallelism include Fx [28], Shared Data Abstractions (SDA's) [13], Sisal [11], Fortran M [23], Split C [18], Linda [10], and the HPF/MPI bindings defined by Foster et al. [24]. Split C provides complete expressibility of task parallelism, but poor facilities for data distribution and little user support for manipulation of shared data types (termed *spread arrays*). Fx and Fortran M, and to a lesser extent SDA, the HPF/MPI bindings, Sisal, and Linda, are aimed at multi-disciplinary applications, where certain disjoint tasks can be run concurrently. They set up explicit communications mechanism (input and output mapping directives in Fx, MPI messages between distributed objects in HPF/MPI, channels and ports in Fortran M, stacks in SDA, and objects in tuple space in Linda) that can transfer information between these tasks. Fx, Fortran M and HPF/MPI do so in a tightly coupled fashion, whereas Linda and SDA do not link sender and receiver directly. Sisal is the only language of those investigated here that is a purely functional language [2] (others are mixtures of functional and imperative languages). It contains no explicit expression mechanism for concurrency. Rather, parallelizing Sisal compilers make use of the guaranteed absence of side effects in function calls and the language distinction between serial and independent (data parallel) loops to derive parallel code. The problem with the above approaches is that there is no support for global (shared) data types within the tasks, other than in the data-parallel sense. Just as in the 'bare' message-passing environment, the user is responsible for interpreting the meaning of arrays owned by the individual processors when using Fx, Fortran M or Linda (tags in Linda can help alleviate this task, but creating and maintaining those is still the programmer's responsibility). SDA,

[2] for a biased discussion of the merits of functional languages, see reference [12].

Sisal and HPF/MPI do support data types globally known to (sets of) tasks, but allow only data parallel or serialized operations on such data types.

Apparently, the user must choose between either task parallelism and private, non-shared data types, or data parallelism and shared data types. What we want is shared data types for programming convenience—where sharing may be among a subset of the processors in the system—and task parallelism for flexibility. One way of accomplishing this is encapsulation through parallel libraries; tasks are issued as parallel, atomic tasks on globally defined, shared data types, and the library implementation, which may involve task parallelism, is hidden from the user. Opting for libraries represents a compromise, since no library can be completely general-purpose. The art of the library designer consists of choosing a system that is small enough that it can be mastered fairly easily, and large and flexible enough to solve more than the particular problem for which it was invented. Some of the interesting projects in this area are ScaLAPACK [7], KeLP [22], PETSc [5], OVERTURE [9], Global Arrays toolkit [41], //ELLPACK [32], PINEAPL [38], Telluride [37], PARTI [49], DAME [17].

ScaLAPACK is a distributed-memory version of the linear-algebra library LAPACK. It allows matrix distributions that are a subset of the HPF array distributions, namely two-dimensional block-cyclic distributions. The library is built on top of lower-level serial (BLAS; Basic Linear Algebra Subprograms) and parallel (PBLAS; Parallel BLAS, and BLACS; Basic Linear Algebra Communication Subprograms) libraries. The efficiency of the library derives from the implementation of the fairly small set of machine-dependent routines of PBLAS and BLACS. Those problems that can be cast completely as numerical linear-algebra problems can be solved readily using ScaLAPACK, with almost the some ease—and a very similar user interface—as in LAPACK. Unfortunately, numerical linear algebra problems are often embedded in larger applications, which may lead to incompatibilities. For instance, a finite-element code may generate the stiffness matrix using a three-dimensional block decomposition, but the resulting matrix equation can only be solved in ScaLAPACK using a two-dimensional decomposition. Remapping may be done, but usually at the substantial cost of a global exchange operation. At NASA Ames, most structured-grid applications do not construct system matrices explicitly, so they do not benefit from ScaLAPACK parallelization.

ScaLAPACK is part of the larger project "A Scalable Library for Numerical Linear Algebra" [52], funded mainly by the Advanced Research Projects Agency. The project's aim is to produce a software library for performing dense and sparse linear algebra computations on massively parallel computers. The work has three main components based on matrix type: dense and banded matrix computations; direct methods for sparse symmetric matrices; iterative methods for sparse nonsymmetric matrices. Some other linear-algebra libraries available or currently being developed are:

- JTpack90 [50], provides iterative methods for sparse problems. It makes use of the the Parallel Gather-Scatter Library PGSLib [21]. The package, written in and callable from Fortran 90, contains a number of Krylov solvers that can operate on matrix as well as matrix-free problems, as long as a residual can be computed.

- Aztec [33], a library written in ANSI C, covers essentially the same ground as JTpack90, but more tools are provided for assessment and control of numerical accuracy.

- LINSOL [47] aims at accommodating both vector and parallel architectures with its Krylov solvers. It emphasizes node code efficiency by providing several storage formats that can be combined for optimal memory access.

- PLAPACK [2] is a software library for dense linear algebra applications.

- PSPARSLIB [45], a Krylov library for parallel sparse iterative solvers, again has aims similar to those of JTpack90, Aztec, and LINSOL.

- The Global Arrays toolkit [41] targets dense matrix operations. It is different from all the other libraries in that it allows both task parallelism and data parallelism, a property that Charon also seeks to provide. Matrices are created and distributed in collective operations, but methods that operate on the matrices can be collective (in the vein of PBLAS [7] operations) as well as private (e.g. processor 1 may fetch a submatrix and perform an operation on it, with all other processors idling, or engaging in other tasks). However, distributions are limited to those in HPF, and are restricted to two-dimensional arrays.

In addition to general-purpose linear-algebra libraries, a plethora of special-purpose parallel packages are being developed, some of which utilize the above linear-algebra libraries. These derive their utility from their execution efficiency, combined with their ease of use. However, most such libraries have specialized too much, sacrificing generality and expandability for efficiency and simplicity.

- Telluride [37], built on top of JTpack90, is a finite-volume unstructured-mesh library for the solution of metal casting problems. It implements a two-phase incompressible Navier-Stokes solver, supplemented with several interface tracking algorithms and solidification models.

- The Kernel Lattice Parallelism (KeLP [22]) project offers a convenient user interface for the solution of partial differential equations using structured, adaptively-refined grids. It provides functions for the creation, movement, reshaping and destruction of the refinements. However, grids and refinement are limited to aligned, Cartesian blocks. More seriously, KELP only provides coarse-level parallelism, and does not allow individual blocks to be further distributed among processors. Numerical operations performed on blocks must be data-parallel. This restricts numerical methods to explicit schemes, Jacobi- or colored-Gauss-Seidel-type point-relaxation, or Krylov-based solvers, all of which compute updates (or residuals, in the case of Krylov subspace methods) on a pointwise basis.

- The University of New Hampshire C* compiler [15] offers support for stencil computations on structured grids. Most notably, staging communications with neighboring processors in the case of so-called box-shaped difference stencils (see below), as described, for example, by Scherr [46], reduces the latency in massively parallel computations. However, since the support is provided within the context of the C* language [39, pp. 450–459], computations can only be performed in a data-parallel fashion.

- OVERTURE [9] is a C++ library for solving partial differential equations on serial and parallel computers. It provides a high-level specification and solution mechanism for partial differential equations on (collections of overlapping) structured grids, with provisions for adaptive refinement. OVERTURE contains procedures for stencil operations and a library of boundary conditions and integrators. All operations are specified as data parallel grid functions.

- //ELLPACK (pronounced "parallel ellpack") [32] offers three mechanisms for the parallel solution of PDE problems. The first (termed the M^+ approach) makes use of sequential legacy software for off-line discretization of the target PDE and expects on input a linear system and a partitioning of the associated matrix. M^+ suffers from memory and bandwidth bottlenecks, especially for nonlinear problems that require rediscretizations at every iteration.
  The second parallelization approach (termed D^+) again uses legacy software for linearization and discretization of the PDE, but now the problem is assembled and solved using the coarse granularity of decomposition of the domain into as many subdomains as there are processors, and of independent solution within each subdomain, supplemented with coupling software.
  The third approach is the only truly parallel problem solver in //ELLPACK, but it is limited to templates describing elliptic equations.

- The Parallel Automated Runtime Toolkit at ICASE (PARTI, [49]) consists of two sets of library functions, namely PARTI proper, and multiblock PARTI. PARTI provides an interface to manipulate data structures related to unstructured meshes and general sparse matrices. It employs the celebrated inspector/executor model, in which loops over irregular data structures are preprocessed to determine their remote data requirements, and the pertinent communications and calls to gather/scatter routines are automatically inserted. This model assumes that all loops are data parallel, so that once remote data is fetched, processors can execute their own segments of the loop independently. This restricts numerical methods to those expressible as data parallel loops, for example those that compute the forces between molecules in molecular dynamics codes like CHARMM [19].
  Multiblock PARTI accommodates sets of interfacing structured grids ("blocks"). Blocks are assigned to sets of processors, and are updated independently, after which irregular communication takes place to update interface values. Within blocks fine-grain parallelism may be exploited through the use of Fortran-D-conforming data distributions and loops. Thus, multiblock PARTI extends the use of Fortran D by allowing task parallelism among the various blocks, but is restricted to the data parallelism expressible in Fortran D within individual blocks.

- The principal aim of DAME (DAta Migration Environment [17]) is to provide a homogeneous distributed virtual machine with a regular virtual topology to the application programmer, hiding the details of the irregularly connected, temporally and architecturally heterogeneous environment on which the application is actually run. Like the Global Arrays project [41], its most important target is dense matrix operations. One of the interesting facets of DAME is that it provides explicit index conversion functions

that translate global indices into local ones, and also local data extractor functions that extract from a specified data domain the part contained in the address space of the calling node. However, operations on distributed data sets are restricted to data parallel functions, and distributions are restricted to 2-dimensional block decompositions.

- The Parallel Industrial NumErical Applications and Portable Libraries (PINEAPL) project [38], spearheaded by the Numerical Algorithms Group (NAG), has created the NAG Parallel Library, which provides an extension of the traditional NAG Fortran 77, Fortran 90, and C libraries. In the style of NAG, support for the solution of partial differential equations (PDEs) in the Parallel Library consists of a set of templates, for the specification of (unstructured) grid and equation to be solved. Communication, which is shielded from the user, is based mainly on the BLACS [7] routines to ensure portability and efficiency. While reported scalability is good [38], functionality is rather limited due to the template nature of the library. In the PDE area only scalar Helmholtz and Poisson solvers are provided at present.

- The Portable, Extensible Toolkit for Scientific computation (PETSc, pronounced "pet-see" [5]) is the most extensive and versatile of the parallelization support packages available today. Rather than providing a (necessarily restricted) template for the parallel formulation and solution of PDEs, it offers a set of functions for the creation, manipulation and destruction of high level distributed data types, such as vectors and matrices, and a collection of general-purpose linear and nonlinear equation solvers.
  Like in the Global Arrays [41] project, distributed data types are created collectively, but may be manipulated collectively (using, for example, PETSc vector routines) as well as individually. PETSc vectors are linear arrays that primarily support irregular grid/graph computations. Collective vector routines (such as `VecAXPY` [6], which is the equivalent of an overwriting DAXPY) are implemented atomically, like HPF's `FORALL`. Individual elements can also be assigned values (`VecSetValues`), but since this incurs the overhead of a function call and possibly communication, such assignments are best grouped together. PETSc provides a vehicle for such aggregation through the `VecAssemblyBegin/End` functions. Vectors are distributed in contiguous segments over the processors in a PETSc context (implemented as an MPI communicator).
  One-, two- and three-dimensional distributed arrays (DAs) are used to support structured-grid computations. Their elements can be accessed using global (i.e. with respect to the global grid) or local (with respect to the local on-processor segment of the array) indexing. Provisions are made for overlap zones (ghost points) that can act as buffers for copies of data elements on geometrically neighboring processors. Elements of DAs, like those of distributed vectors, can be set collectively and individually.
  With the proper use of the assembly routines it is possible, in principle, to program pipeline control structures explicitly, which has the advantage that the grouping factor is under the control of the user. However, PETSc allows only one type of distribution for its vectors and DAs, namely blocking (i.e. uni-partitioning). There is no support for more advanced domain decompositions, such as multi-partitioning. Nor is there support for dynamic decompositions, such as those required by transpose-based parallel algorithms. Finally, there are several other data accesses in DAs that are required in

complex CFD production codes and that are not supported in PETSc, such as fetches of data from remote processors at points other than ghost points.

We also mention the problem solving frameworks of Linda Program Builder [1], Multi-agent Environment for MPSEs [20], and POOMA [44]. These differ from the parallel libraries above in that they provide effectively just an empty shell, with little explicit support for the difficult task of discretizing and solving partial differential equations or other important engineering problems. We will not consider those environments any further.

3 Charon conceptual design

Based on the survey of projects on parallelization aids for scientific computing in the previous section, and on experience developing advanced parallel programs from scratch, we arrive at the following basic design guidelines for Charon.

1. Every control structure and data access expressible in a serial code should also be expressible in a parallel code. Common control structures should be easy to express;

2. Domain decompositions should be flexible, potentially dynamic, and under complete control of the user. Common domain decompositions should be easy to specify;

3. The user should always be able to get efficient access (i.e. without the need to copy) to memory locations where the (distributed) data is actually stored;

4. Programming in Charon should not have to be done exclusively through explicit function calls;

5. Converting a serial program or design to a parallel program should be easy;

6. Parallel I/O should be straightforward and efficient;

Criteria 1–4 are largely satisfied by the message-passing model, which we adopt for this work. With its wide acceptance and proven efficiency on a large number of platforms, the Message Passing Interface (MPI [48]) is the proper choice. Easy expressibility of common control structures and domain decompositions requires a layer of functions on top of lower-level constructs. Criterion 4 is a corollary to 1; in systems that force the programmer to use library calls alone for accomplishing tasks, functionality is inherently limited, and too much 'foreign language' is required.[3] User control in the user's language (mostly Fortran and C) should be explicitly recognized and supported, not merely allowed. Criteria 5-6 are generally at odds with the low-level functionality and data distribution support of the message-passing model; flexibility and ease of use have been found incompatible in virtually all of the systems surveyed in Section 2. This is because most systems provide only one level of programming support, which also needs to be *efficient* to qualify as a useful instrument. Charon's design includes a hierarchy of control structures, all of which have complete expressibility, but trade ease of use against efficiency. Data distribution and control are orthogonal design features of

[3] Compare ordering food in a French restaurant; an English-speaking customer may be willing to learn a few French words to order a special meal, but will cancel the reservation if the whole dinner conversation has to be conducted in French as well.

Charon: functions at each level of control can operate on any domain decomposition. As a consequence, top-performing parallel codes derived using Charon will often look very similar to message-passing codes, although many of the common manipulations will not have to be coded explicitly by the user, but will be provided as tools in the toolkit. The important difference is that Charon codes will have been created in a piecemeal fashion, with support for rapid prototyping and validation. I/O is usually a data-parallel operation, and support for this is easily, efficiently, and transparently provided in Charon, using the collective I/O operations defined in the MPI 2 standard [40].

3.1 Creating domain decompositions

Charon is intended for structured-grid parallel scientific computing. More specifically, it supports the solution of partial differential equations. Such problems often involve stencil operations that require gathering of data from nearby points in the grid. Hence, most useful domain decompositions attempt to assign contiguous blocks of the grid to individual processors, which reduces the amount of communication necessary to fetch nonlocal data. An additional convenience in the manipulation of domain decompositions is obtained by defining them in terms of *Cartesian sections*. These are regular tessellations created by cutting the grid along coordinate planes. In order to retain full flexibility, we separate the construction of Cartesian sections from the assignment of cells or *partitions* to individual processors. Completely arbitrary decomposition of a three-dimensional grid of $n_x \times n_y \times n_z$ points is possible by applying $n_x - 1$, $n_y - 1$, and $n_z - 1$ cuts in the three respective coordinate directions, thus creating $n_x \cdot n_y \cdot n_z$ partitions of a single grid point each, which can then be assigned to individual processors. The crucial element in obtaining this flexibility is the divorce of the tessellation from the processor assignment process. The resulting data structure, called *decomposition*, is fundamental to Charon. It can be created easily and in a single step for the most common data distributions, such as uni- and multi-partition [51], and block-cyclic distributions [36]. Decompositions can be tailored to specific needs by modifying them after creation. An example of the use of Cartesian sections of a two-dimensional grid to define decompositions is shown in Figure 1.

3.2 Defining distributed variables

By default, all variables declared in Charon are private, which means that they are accessible by only one processor. This is consistent with the MIMD programming model. When a distributed grid variable, or *distribution*, is created, it is associated with a certain domain decomposition. The variable can have arbitrary tensor rank and vector dimensions. Buffer regions for the accommodation of data copied from neighboring partitions (so-called ghost points) are specified at the time of creation of the distributed variable. Distributed variables can allocate their own space (Fortran 90, C, C++), or take a pointer to previously allocated space (Fortran 77/90, C, C++). The latter operation of memory association is important in cases where the user assumes total responsibility for memory management, for example by the use of common blocks and/or work arrays. Furthermore, it is the only reasonable way to give the user access to the location where the grid-related data actually resides in memory when using Fortran 77. Handles restrict operations on data to those foreseen by the library writer, and the access mechanism provided by PETSc [6, p. 89] (see below), apart

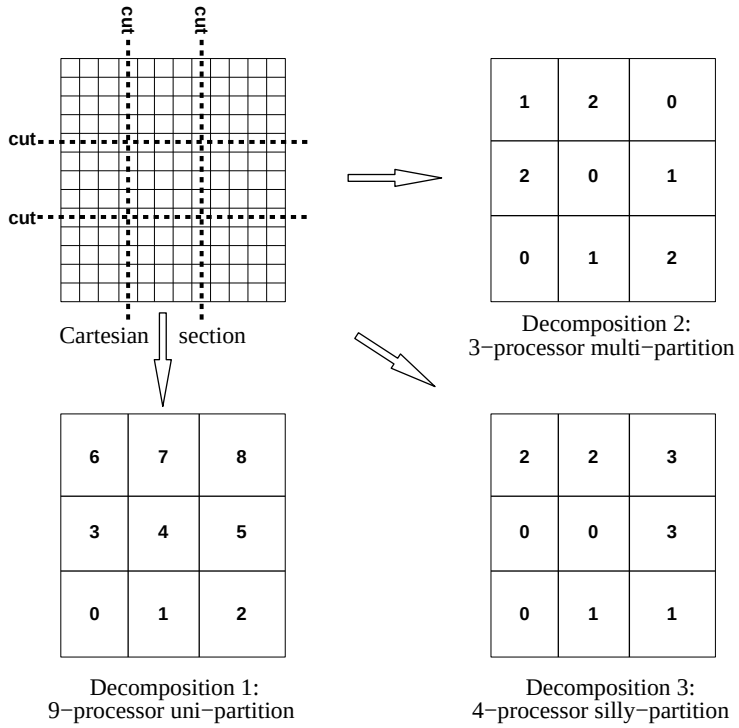


Figure 1: A Cartesian section can be used to define several different domain decompositions. Numbers inside partitions indicate processor ownership.

from precluding reusing scratch space, is awkward and unnatural.

Accessing data associated with a vector in PETSc requires use of an auxiliary array (`xx_v`), whose starting address is used as an 'anchor' for referring to the actual data, and whose size is ignored. In the example below, which sets the values of PETSc vector `x` directly, the offset `xx_i` signifies the distance of the data from the anchor. A preprocessor statement is used to ease index computations. Note that `x` is a handle, whose Fortran 77 type is integer.

```
#define xx_a(ib)  xx_v(xx_i + (ib))

      double precision xx_v(1)
      integer          x, xx_i, i, ierr, n

      call VecGetArray    (x, xx_v, xx_i, ierr)
      call VecGetLocalSize(x, n,          ierr)
      do   i = 1, n
         xx_a(i) = 3.0d0*i + 1.0d0
      end do
      call VecRestoreArray(x, xx_v, xx_i, ierr)
```

In Charon the user has immediate access to local data that is part of a distributed variable if it is preallocated. The distributed variable merely provides a structuring interpretation of that space. Whereas the advantage is clear, the disadvantage is that there is less protection against memory errors due to insufficient space allocated for the data. Therefore, Charon provides interrogation routines that return for a created data structure the minimum data storage size required.

3.3 Manipulating distributed variables

One of the most useful and common operations on distributed variables is `copy_faces`, which fetches values from neighboring processors and stores them in the ghost point locations. This facilitates standard stencil operations without the need for frequent remote data accesses. As in PETSc [6], support is provided for so-called box- and star-shaped difference stencils. This controls whether or not corner ghost points need to be copied. However, rather than making this an attribute of the distributed variable, it is specified at the time of the `copy_faces` operation, as is the update of array values on periodic boundaries. A special form of the copy operation is the update of the ghost point values of (part of) a particular face of an individual partition. This is a useful and convenient operation in applications that involve recurrences.

Semantically, `copy_faces` is a data-parallel operation (although it is actually staged for efficiency reasons in the case of box-shaped difference stencils), and it is executed atomically by the collection of processors in the *communicator* used for defining the distributed variable. Another data-parallel communication operation is the *redistribution* (possibly *in situ*) of a distributed variable from one domain decomposition to another. This action, which may involve a global exchange, is useful for supporting transpose-based algorithms, such as the Fast Fourier Transform program FT in the NAS Parallel Benchmarks 2 program suite [4]. The last data parallel communication in Charon is `copy_tile`, which is issued in terms of a Cartesian subset of a distributed variable (similar to the fetch and store into a rectangular patch of a two-dimensional array in the Global Arrays [41] toolkit). If the data is local, only a copy operation will be executed. If the data is remote, a communication will be initiated. In general, both remote and local data will be involved in the grid-based private communications. The user may interrogate the domain decomposition to find out which processors contribute what part of the data, but this is not necessary to execute `copy_tile` efficiently. The user need only specify the requested data segment of the appropriate distributed variable. As with all data-parallel communications, all processors included in the communicator used to define the distributed variable must issue the copy call, but only those that are contained in the destination communicator receive a copy of the tile data.

Besides `redistribute` and `copy_faces/tile`, no standard data-parallel communications are envisaged. That means that any other communication must either be expressed explicitly as a private operation, or implicitly as a remote data access invoked by an assignment or other data-usage statement (e.g. `print`). Private communications take the form of standard message passing calls, which have been fully described in the MPI reference [48]. They offer complete flexibility, but no explicit support for distributed data types. Implicitly invoked communications will generally be inefficient. Rather than attempting to improve efficiency through an optimizing, parallelizing compiler—no generally successful attempts to produce

such a compiler have been reported yet in the literature (see, for example, reference [31])—we accept the fine granularity of such operations. Implicitly invoked communications allow a coding style that is very close to serial legacy code, which is a convenience when converting large programs. The performance degradation is usually severe, because we also make the conservative assumption that code fragments resulting in implicit communications need to be executed serially to guarantee deterministic program execution. Hence, frequent synchronization will be necessary. Two vehicles to improve performance, both involving user interaction, are available.

- Before the operation responsible for invoking implicit communications is executed, the user makes sure that all remote data requirements are satisfied implicitly by placing private or data parallel communication calls. Moreover, the target loop or assignment statement is marked as `independent`, which means it can be executed independently from all other processors (similar to HPF's `INDEPENDENT` directive). It does *not* mean that the statements in the loop body (if any) can be executed in arbitrary order. Since the remote data requirement must be satisfied implicitly, this strategy can only be applied when the nonlocal data corresponds to the ghost points of a partition (the assignment operator 'finds' the data where it expects it).

- Before the operation responsible for invoking implicit communications is executed, the user makes sure that all remote data requirements are satisfied *explicitly* using private communications. This implies a (partial) recoding of the original operation to assimilate the foreign data, which should only be done as a last tuning step, after correctness of the parallel program has been verified.

Note that implicitly generated communications can often be avoided. Only segments of the code that involve recurrences that cannot be satisfied with local data (i.e. the non-data-parallel segments) require real work to increase the granularity of the communications.

4 Summary

Previous efforts at providing efficient and general tools for creating parallel programs have not been successful, except for the cumbersome message-passing mechanism. All general-purpose tools surveyed in this report lack expressiveness in both data distribution and control structures, depriving the programmer of the means to obtain efficiency. Most parallelization tools only allow data parallelism, which is not adequate for the efficient implementation of implicit numerical algorithms in computational fluid dynamics. Those systems that do accommodate task parallelism generally do so at a very high level, mainly to support multi-discipline applications and for very-coarse-grain pipelining of image processing applications. Parallel libraries that target a certain problem domain generally provide more efficiency, but suffer from reduced functionality due to encapsulation of data types and operations.

Charon recognizes the need for sophisticated data distributions and control structures. *Cartesian sections* of structured grids and user-controlled *partition* assignments lead to arbitrarily complex *decompositions* of multi-dimensional arrays associated with variables defined on the grid. Control structures requiring non-local data are available at three levels of abstraction, listed in order of increasing efficiency and decreasing user friendliness.

1. Global indexing of variables and implicitly invoked communications, resulting in serialized execution and frequent communication and synchronization.

2. Global indexing of variables and implicitly satisfied remote data requests, either through data parallel communications provided by Charon, or through private communications using regular message passing.

3. Local indexing of variables and implicitly satisfied remote data requests, either through data parallel communications provided by Charon, or through private communications using regular message passing.

This hierarchy of mechanisms allows a gradual transition from serial legacy code to high-quality parallel code.

References

[1] S. Ahmed, N. Carriero, D. Gelernter, *A program building tool for parallel applications*, DIMACS Workshop on Specifications of Parallel Algorithms, Princeton University, Princeton, NJ, May 1994

[2] P. Alpatov, G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, R. van de Geijn, Y.J. Wu, *PLAPACK: Parallel linear algebra package*, Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997

[3] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, S.K. Weeratunga, *The NAS Parallel Benchmarks*, Int. J. Supercomputing Applications, Vol. 5, No. 3, pp. 63-73, 1991

[4] D.H. Bailey, T. Harris, W.C. Saphir, R.F. Van der Wijngaart, A.C. Woo, M. Yarrow, *The NAS parallel benchmarks 2.0*, Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA 94035, December 1995

[5] S. Balay, W.D. Gropp, L. Curfman McInnes, B.F. Smith, *Efficient management of parallelism in object-oriented numerical software libraries*, Modern Software Tools in Scientific Computing, E. Arge, A.M. Bruaset, H.P. Langtangen, Ed., Birkhauser Press, 1997

[6] S. Balay, W.D. Gropp, L. Curfman McInnes, B.F. Smith, *PETSc 2.0 Users manual*, Report ANL–95/11 - Revision 2.0.17, Argonne National Laboratory, Argonne, IL 60439, 1997

[7] L.S. Blackford et al., *ScaLAPACK: a portable linear algebra library for distributed memory computers - design issues and performance*, Proc. Supercomputing '96, Pittsburgh, PA, November 1996

[8] P. Boulet, T. Brandes, *Evaluation of automatic parallelization strategies for HPF compilers*, Research Report 95-44, Ecole Normale Supérieure de Lyon, France, November 1995

[9] D.L. Brown, G.S. Chesshire, W.D. Henshaw, D.J. Quinlan, *Overture: An object-oriented software system for solving partial differential equations in serial and parallel environments*, Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997

[10] N. Carriero, D. Gelernter, *How to write parallel programs: a guide to the perplexed*, ACM Computing Surveys, Vol. 21, No. 3, pp. 323–357, 1989

[11] D.C. Cann, *The Optimizing SISAL Compiler: Version 12.0* Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, Livermore, CA 94550, April 1992

[12] D.C. Cann, *Retire Fortran? A debate Rekindled*, Communications of the ACM, Vol. 35, No. 9, pp. 81-89, August 1992

[13] B. Chapman, P. Mehrotra, J. Van Rosendale, H. Zima, *A software architecture for multi-disciplinary applications: Integrating task and data parallelism*, Technical Report 94-18, ICASE, NASA Langley Research Center, Hampton, VA, March 1994

[14] B. Chapman, P. Mehrotra, H. Zima, *Programming in Vienna Fortran*, Scientific Programming, Vol. 1, 1, 1992

[15] S.W. Chappelow, P.J. Hatcher, J.R. Mason, *Optimizing data-parallel stencil computations in a portable framework*,Proc. Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, Troy, New York, May 1995

[16] C. Clémençon, K.M. Decker, V.R. Deshpande, A. Endo, J. Fritscher, P.A.R. Lorenzo, N. Masuda, A. Müller, R. Rühl, W. Sawyer, B.J.N. Wylie. F. Zimmerman, *Tools-supported HPF and MPI parallelization of the NAS Parallel Benchmarks*, Proc. 6^{th} Symp. on the Frontiers of Massively Parallel Computing (Frontiers'96, Annapolis), pp. 309-318, IEEE Comp. Soc. Press, 1996

[17] M. Colajanni, M. Cermele, *DAME: An environment for preserving efficiency of data parallel computations on distributed systems*, IEEE Concurrency, pp. 41-55, January-March 1997

[18] Culler et al., *Parallel programming in Split-C*, Proc. Supercomputing '93, Portland, OR, pp. 262–273, November 1993

[19] R. Das, J. Saltz, *Parallelizing molecular dynamics codes using PARTI primitives software*, Proc. Sixth SIAM Conference on Parallel Processing for Scientific Computing, Norfolk, VA, March 1993

[20] T. Drahansky, A. Joshi, J. Rice, E. Houstis, S. Weerawarana, *A multiagent environment for MPSEs*, Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997

[21] R.C. Ferrel, D.B. Kothe, J.A. Turner, *PGSLib: A library for portable, parallel, unstructured mesh simulations*, Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997

[22] S.J. Fink, S.B. Baden, S.R. Kohn, *Flexible communication mechanisms for dynamic structured applications*, Proc. Third International Workshop on Parallel Algorithms for Irregularly Structured Problems, Santa Barbara, CA, August 1996

[23] I.T. Foster, K.M. Chandy, *Fortran M: A language for modular programming*, Technical Report MCS-P327-0992, Argonne National Laboratory, June 1992

[24] I.T. Foster, D.R. Kohr, R. Krishnaiyer, A. Choudhary, *Double standards: Bringing task parallelism to HPF via the Message Passing Interface*, Proc. Supercomputing '96, Pittsburgh, PA, November 1996

[25] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, M. Wu, *Fortran D language specification*, Technical Report TR90079, Dept. of Computer Science, Rice University, December 1990

[26] Gannon et al., *Implementing a parallel C++ runtime system for scalable parallel systems*, Proc. Supercomputing '93, Portland, OR, pp. 588–597, November 1993

[27] A.S. Grimshaw, *Easy to use object-oriented parallel programming with Mentat*, IEEE Computer, pp. 39-51, May 1993

[28] T. Gross, D. O'Hallaron, J. Subhlok, *Task parallelism in a High Performance Fortran framework*, IEEE Parallel & Distributed Technology, Vol. 2, No. 3, pp. 16-26, 1994

[29] M.W. Hall, S.P. Amarasinghe, B.R. Murphy, S-W. Liao, M.S. Lam, *Detecting coarse-grain parallelism using an interprocedural parallelizing compiler*, Proceedings Supercomputing '95, San Diego, CA, December 1995

[30] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.-W. Liao, E. Bugnion, M.S. Lam, *Maximizing Multiprocessor Performance with the SUIF Compiler*, IEEE Computer, December 1996.

[31] S. Hiranandani, K. Kennedy, C. Tseng, *Preliminary experiences with the Fortran D compiler*, Proc. Supercomputing '93, Portland, OR, pp. 338–350, November 1993

[32] E.N. Houstis, S.B. Kim, S. Markus, P. Wu, A.C. Catlin, S. Weerawarana, T.S. Papatheodorou, *Parallel ELLPACK Elliptic PDE Solvers*, Proc. INTEL SuperComputer User's Group Conference, Albuquerque, New Mexico, June 1995.

[33] S.A. Hutchinson, J.N. Shadid, R.S. Tuminaro, *Aztec User's Guide: Version 1.1,* Sandia National Laboratories Technical Report, SAND95-1559, October 1995

[34] C.S. Ierotheou, S.P. Johnson, M. Cross, P.F. Leggett, *Computer aided parallelisation tools (CAPTools)—conceptual overview and performance on the parallelisation of structured mesh codes*, Parallel Computing Vol. 22, pp. 163–195, 1996

[35] E. Johnson, D. Gannon, *Programming with the HPC++ parallel Standard Template Library*, Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997

[36] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, M. Zosel, *The High Performance Fortran Handbook*, MIT Press, Cambridge, MA, 1994

[37] D.B. Kothe, R.C. Ferrel, J.A. Turner, S.J. Mosso, *A high resolution finite volume method for efficient parallel simulation of casting processes on unstructured meshes*, Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997

[38] A. Krommer, M. Derakhshan, S. Hammarling, *Solving PDE problems on parallel and distributed computer systems using the NAG parallel library*, Proc. High Performance Computing and Networking '97, Vienna, Austria, April 1997

[39] V. Kumar, A. Grama, A. Gupta, G. Karypis, *Introduction to parallel computing*, Benjamin/Cummings, 1994

[40] MPI Forum, *MPI-2: Extensions to the Message-Passing Interface*, URL: "http://www.mpi-forum.org/docs/docs.html"

[41] J. Nieplocha, R.J. Harrison, R.J. Littlefield, *The global array programming model for high performance scientific computing*, SIAM News, Vol. 28, No. 7, August-September 1995

[42] Parallel Computing Forum, *Parallel Extensions for Fortran*, X3H5/93-SD1-Revision L, URL: "http://www.cs.orst.edu/standards/ANSI-X3H5/94/"

[43] R. Ponnusamy, Y-S. Hwang, R. Das, J. Saltz, A. Choudhary, G. Fox, *Supporting irregular distributions in Fortran D/HPF compilers*, Technical report CS-TR-3268, University of Maryland, Department of Computer Science, 1994

[44] J.V.W. Reynders, *The POOMA framework—a templated class library for parallel scientific computing*, Proc. Eighth SIAM Conference on Parallel Processing for Parallel Processing, Minneapolis, MN, March 1997

[45] Y. Saad, S. Kuznetsov, G-C. Lo, A. Malevsky, A. Chapman, *PSPARSLIB: A portable library of parallel sparse iterative solvers*, Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997

[46] S.J. Scherr, *Implementation of an explicit Navier-Stokes algorithm on a distributed memory parallel computer* AIAA Paper 93-0063, 31^{st} Aerospace Sciences Meeting and Exhibit, Reno, NV, 1993

[47] W. Schönauer, H. Häfner, R. Weiss, *LINSOL, a parallel iterative linear solver package of generalized CG-type for sparse matrices*, Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997

[48] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, *MPI: The Complete Reference*, MIT Press, 1995.

[49] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, *PARTI primitives for unstructured and block structured problems*, Computing Systems in Engineering (Proc. Noor's Flight Systems Conference), pp. 73-86, Vol. 3, No. 1, 1992

[50] J.A. Turner, R.C. Ferrel, D.B. Kothe, *JTPack90: A parallel, object-based, Fortran90 linear algebra package*, Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997

[51] R.F. Van der Wijngaart, *Efficient implementation of a 3-dimensional ADI method on the iPSC/860*, Proc. Supercomputing '93, Portland, OR, November 1993

[52] URL: "http://www.epm.ornl.gov/~walker/scalable-libraries.html"

[53] URL: "http://www.sgi.com/TEXT/Products/hardware/servers/products/MipsProCompilers.html"

[54] URL: "http://www.kai.com:80"

[55] ——, *CM Fortran Reference Manual, version 5.2*, Thinking Machines Corporation, Cambridge, MA, 1989

[56] ——, *Forge Explorer User's Guide*, Advanced Parallel Research
URL: "http://www.qpsf.edu.au/workshop/forge/forge.html"

[57] ——, *CF90 commands and directives reference manual, SR-3901.10*, Cray Research, Inc., 1993