Incremental parallelization of non-data-parallel programs using the Charon message-passing library

*Rob F. Van der Wijngaart**

1 Introduction

Message passing is among the most popular techniques for parallelizing scientific programs on distributed-memory architectures. The reasons for its success are wide availability (MPI [8]), efficiency, and full tuning control provided to the programmer. A major drawback, however, is that incremental parallelization, as offered by compiler directives, is not generally possible, because all data structures have to be changed throughout the program simultaneously. Charon remedies this situation through mappings between distributed and nondistributed data. It allows breaking up the parallelization into small steps, guaranteeing correctness at every stage.

Several tools are available to help convert legacy codes into high-performance message-passing programs. They usually target data-parallel applications, whose loops carrying most of the work can be distributed among all processors without much dependency analysis (e.g. [5]). Others do a full dependency analysis and then convert the code virtually automatically (e.g. [6]). Even more toolkits are available that aid construction from scratch of message passing programs [2, 3, 4, 7]. None, however, allows piecemeal translation of codes with complex data dependencies (i.e.

*Computer Sciences Corporation; M/S T27A-1, NASA Ames Research Center, Moffett Field, CA 94035-1000; e-mail: wijngaar@nas.nasa.gov, Numerical Aerospace Simulation Systems Division, NASA Ames Research Center

non-data-parallel programs) into message passing codes.

The Charon library (available in both C and Fortran) provides incremental parallelization capabilities by linking legacy code arrays with distributed arrays. During the conversion process, nondistributed and distributed arrays exist side by side, and simple mapping functions allow the programmer to switch between the two in any location in the program. Charon also provides wrapper functions that leave the structure of the legacy code intact, but that allow execution on truly distributed data. Finally, the library provides a rich set of communication functions that support virtually all patterns of remote data demands in realistic structured-grid scientific programs, including transposition, nearest-neighbor communication, pipelining, gather/scatter, and redistribution. At the end of the conversion process most intermediate Charon function calls will have been removed, the nondistributed arrays will have been deleted, and virtually the only remaining Charon functions calls are the high-level, highly optimized communications.

Distribution of the data is under complete control of the programmer, although a wide range of useful distributions is easily available through predefined functions. A crucial aspect of the library is that it does not allocate space for distributed arrays, but accepts programmer-specified memory. This has two major consequences. First, codes parallelized using Charon do not suffer from encapsulation; user data is always directly accessible. This provides high efficiency, and also retains the possibility of using message passing directly for highly irregular communications. Second, nondistributed arrays can be *interpreted* as (trivial) distributions in the Charon sense, which allows them to be mapped to truly distributed arrays, and vice versa. This is the mechanism that enables incremental parallelization.

In this paper we provide a brief introduction of the library (for more details see [9]) and then focus on the actual steps in the parallelization process, using some representative examples from, among others, the NAS Parallel Benchmarks [1]. We show how a complicated two-dimensional pipeline—the prototypical non-data-parallel algorithm— can be constructed with ease. To demonstrate the flexibility of the library, we give examples of the stepwise, efficient parallel implementation of nonlocal boundary conditions common in aircraft simulations, as well as the construction of the sequence of grids required for multigrid.

2 Data distribution

Charon supports the parallelization of scientific computations through domain decomposition. One or more multi-dimensional grids are defined, and arrays are associated with these grids. The grids are divided into nonoverlapping pieces, which are assigned to the processors in the computation. The associated arrays are thus distributed as well. This process takes place in several steps.

First, Fig. 1a, a logically rectangular *grid* of a certain dimensionality and extent is defined, using `CHN_Create_grid` and `CHN_Set_grid_size`, respectively. This step establishes a geometric framework for all arrays associated with the grid. It also attaches to the grid an MPI [8] communicator, which serves as the context and processor subspace within which all subsequent Charon-orchestrated commu-

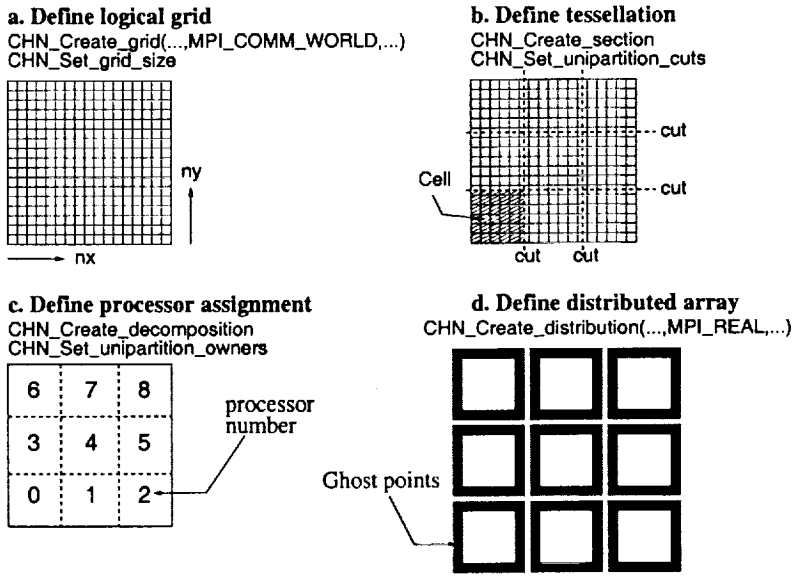

Figure 1. *Defining distributed arrays using Charon*

nications take place. Multiple coincident or non-coincident communicators may be used within one program, allowing the programmer to assign the same or different (sets of) processors to different grids in a multiple-grid computation.

Second, Fig. 1b, the grid is divided into a number of logically rectangular *cells*, using `CHN_Create_section`, and `CHN_Set_cuts`. The resulting *section* contains a number of cutting planes (*cuts*) along each coordinate direction. Whereas the programmer can specify any number of cuts and cut locations, a single high-level instruction often suffices to define all the cuts belonging to a particular domain decomposition. For example, `CHN_Set_unipartition_cuts` divides the grid evenly into as many cells as there are processors in the communicator—nine in this case.

Third, Fig. 1c, cells are assigned to processors, resulting in a *decomposition*, which is initialized using `CHN_Create_decomposition`. The programmer can assign ownership of each cell in the decomposition to any processor individually, or all at once through a single high-level instruction. For example, `CHN_Set_unipartition_owners` assigns each cell in the unipartition section to a different processor. Creating the cells and assigning them to processors are separated to provide flexibility. We may divide a grid into a target number of cells for execution on a parallel machine, but assign all cells to the same processor for debugging on a workstation.

Finally, Fig. 1d, arrays with one or more spatial dimensions (same as the grid) are associated with a decomposition, resulting in *distributions*. The associated function is `CHN_Create_distribution`. The arrays may represent scalar quantities at each grid point, or higher-order tensors. A distribution has a regular MPI data type (`MPI_REAL` in this case). To accommodate stencil operations we specify a number of *ghost points*. These form a border (shaded area) around each cell, which acts

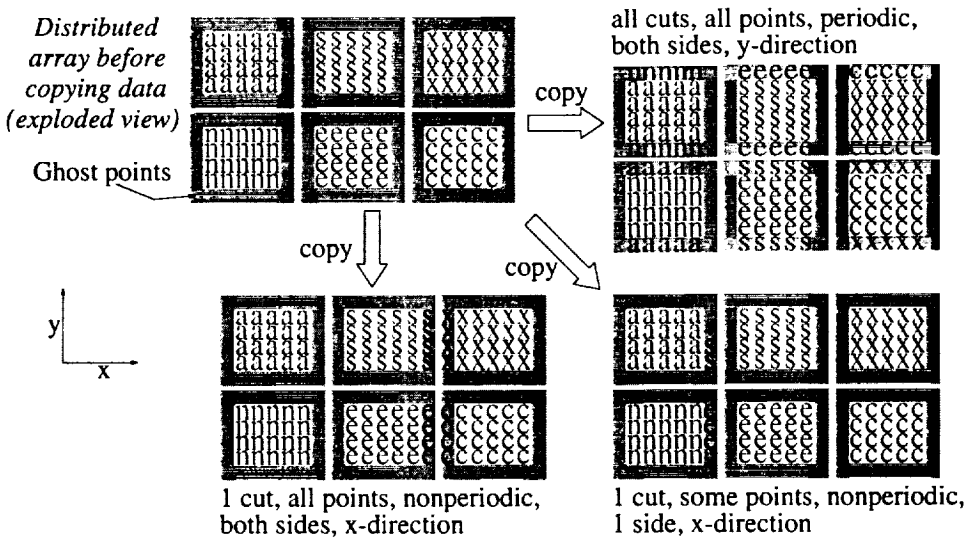

Figure 2. *Examples of copying data from adjacent cells (`CHN_Copy_face`) for a two-dimensional grid*

as a buffer for data copied from adjacent cells. Most importantly, the programmer specifies the starting address of the data storing the distributed array. While at first this may appear a disadvantage—the user has to manage all memory—it enables Charon to support incremental parallelization, as explained in Section 3.2.

3 Communications

A *data-parallel computation* can be loosely defined as a collection of independent operations on a shared data set. Task assignment is determined by the subset of the data owned by each processor. We can similarly define a *data-parallel communication* as a collection of independent data transfers within a shared data set. Assignment of transfer tasks is determined by the data source and destination locations. Most of Charon's communications are data parallel, specified completely in terms of grid points, not processors.

3.1 Copying interface data

In stencil computations on structured grids the need for nonlocal data is often limited to (spatially) nearest neighbors. `CHN_Copy_faces` lets the programmer specify exactly which ghost points to update. Some examples of the effect of the copy operation are given in Fig. 2. Since Charon does not require that all potential communication candidate points in a grid be involved, it is possible to support non-data-parallel computations through data-parallel communications. In general, all processors within the grid's MPI communicator execute `CHN_Copy_faces`, but those that do not own points involved in the operation may safely skip the call. A useful

variation is `CHN_Copy_faces_all`, which fills all ghost points of the distribution in all coordinate directions. It is the variation most commonly encountered in other parallelization packages for structured-grid applications, since it conveniently supports data parallel computations. But it is not suitable to implement, for example, the pipeline algorithm of Section 4.1.

3.2 Redistribution

`CHN_Redistribute` maps between *any* two compatible distributions (same grid, data type, and tensor rank). This enables, for example, transposition of data [9], but also mapping between distributed and nondistributed arrays. This is a result of the fact that Charon takes addresses of user data to store distributed arrays. A nondistributed array in a legacy code can be viewed as a trivially distributed array (section consists of a single cell) in the Charon sense, and can therefore be mapped to a truly distributed array. Let `ad` and `a` be the integer handles of the distributed and nondistributed arrays, respectively, that Charon assigns to distributions. The single statement `CHN_Redistribute(ad,a)` suffices to scatter the nondistributed array. `CHN_Redistribute(a,ad)` does the gather.

A practical way of constructing *parallel bypasses* of serial code is as follows. First, define trivial distributions for nondistributed legacy code arrays. This requires figuring out the dimensions of the grids used and defining the sections, decompositions and distributions accordingly. Each section has only one cell. No new space is needed; the programmer points to the starting addresses of the legacy code arrays when defining the distributions. Next, define the corresponding truly distributed arrays (again using Charon), and allocate the associated memory. Let `a1d`, `a2d`, `a3d` and `a1`, `a2`, `a3` be the handles of distributed and nondistributed arrays, respectively. Now we can define distributed/serial adaptor functions `spread/collect`.

```
void spread(ad1,ad2,ad3,a1,a2,a3){
  int ad1,ad2,ad3,a1,a2,a3;
  CHN_Redistribute(a1d,a1);
  CHN_Redistribute(a2d,a2);
  CHN_Redistribute(a3d,a3); return;
}
```

```
void collect(a1,a2,a3,a1d,a2d,a3d){
  int ad1,ad2,ad3,a1,a2,a3;
  CHN_Redistribute(a1,a1d);
  CHN_Redistribute(a2,a2d);
  CHN_Redistribute(a3,a3d); return;
}
```

Code bracketed by these calls has full access to the distributed arrays, but the rest of the serial program remains unchanged. Thus, the programmer can focus on a single loop or even statement and parallelize it, using Charon's communication and/or wrapping [9] functions, and be assured that the rest of the program is still correct.

3.3 Gather/scatter

The last communication provided is `CHN_Get_tile` (and its inverse `CHN_Put_tile`). It copies a subset of a distributed array—possibly owned by several processors—into the local memory of a specified processor (cf. Global Arrays' `ga_get` [7]). This is useful for applications that have non-nearest-neighbor data dependencies, such as nonlocal boundary conditions for flow problems (Section 4.3).

4 Examples

We show by example the steps taken to parallelize legacy code using Charon, and demonstrate its expressiveness and functionality.

4.1 NAS LU Parallel Benchmark

In [9] we described the results of parallelizing two Fortran codes (SP, LU) from the well-known NAS Parallel Benchmarks [1], where it was found that the performance of the Charon versions was comparable to that of the original, painstakingly derived plain MPI (NPB-MPI) versions. Here we describe in more detail the steps in creating the two-dimensional pipeline in the parallel LU. The numerical problem is $Au^{n+1} = b(u^n)$, where u is the time-dependent solution, n the time step index, and b an explicit 13-point-star stencil operator. The discretization matrix A is the sum of L_+ and L_-, first-order direction-biased difference operators that define two sweeps over the entire grid. L_- demands that no point (i,j,k) be updated before points (i_p,j_p,k_p) with smaller indices: $(i_p \le i, j_p \le j, k_p \le k, (i_p,j_p,k_p) \ne (i,j,k))$. This data dependency is the same as for the Gauss-Seidel method with lexicographical point ordering. L_+ sweeps in the other direction.

NPB-MPI divides the grid into pencils, one per processor, and pipelines the solution process, Fig. 3; we do the same with Charon. Here is the code that defines a 3D grid, creates a section, excludes the third dimension from partitioning, creates and assigns the pencils, and defines a distribution (two ghost points) representing a vector with five components at each grid point on the resulting decomposition.

```
call CHN_Create_grid(grid, MPI_COMM_WORLD, 3)
call CHN_Set_grid_size(grid,0,nx)
call CHN_Set_grid_size(grid,1,ny)
call CHN_Set_grid_size(grid,2,nz)
call CHN_Create_section(section,grid)
call CHN_Exclude_partition_direction(section,2)
call CHN_Set_unipartition_cuts(section)
call CHN_Create_decomposition(decomp,section)
call CHN_Set_unipartition_owners(decomp)
call CHN_Create_distribution(dist,section,MPI_REAL8,arr_dat,2,1,5)
```

The variable `arr_dat` contains the starting address of the memory associated with the distribution. In Fortran this is usually an array name.

The initial step in the parallelization of the serial version of LU is to select a target piece of code, bracket it with a parallel bypass (see Section 3.2) and parallelize the intervening statements. We focus on `blts`, which applies the L_--operator. The serial code contains a triple loop (over `k`, `j`, and `i`) that traverses the grid in the natural order and applies L_- in a pointwise fashion. That is, the third array index `k` is in the outer loop, and the computation proceeds plane by plane. We rewrite the loop nest so that the index space is tiled, where each tile corresponds to a unit-thickness `k`-slice of a pencil of the decomposition. This is a simple process, because Charon provides query functions that return start and end indices of cells in a decomposition. Each tile fetches ghost point data from its two neighbors in

the -i and -j direction (CHN_Copy_faces), regardless of the location of the pencil. Communications beyond the boundary of the grid are simply ignored if periodicity is set to false. The start of the two-dimensional pipeline is illustrated in Fig. 3.

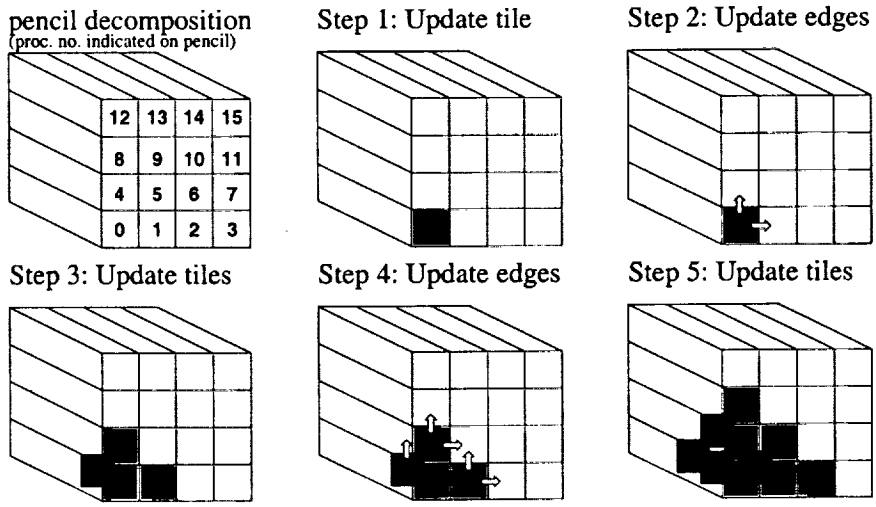

Figure 3. *Pencil decomposition (16 processors); Start of pipelined solution process for LU code (L_-)*

All processors participate in the copying of neighbor data, even though they have no work to do for pencils not (adjacent to) their own. However, the calling processor only applies L_- if it owns the current tile (this test, line 7, is also supported using a query function). In outline this process looks as follows:

```
001    do 10  k = 2, k_max-1
002      do 10  jcell = 1, jcell_max
003        do 10  icell = 1, icell_max
004          copy ghost point data from cell(icell,jcell-1) in slice k
005          copy ghost point data from cell(icell-1,jcell) in slice k
006          if (calling processor owns cell(icell,jcell))
007            loop over points in slice k of cell(icell,jcell) and apply L-
008 10 continue
```

Lines 4 and 5 are single calls to function CHN_Copy_faces. We describe the parameters for copying in the i-direction (cf. Figures 1, 2): periodicity = false, copy direction = right (positive i-direction), copy dimension = 0 (first coordinate direction), cut number = icell-1, number of ghost points copied = 1 (L_- is a first-order difference operator), tensor components = all. The remaining two parameters are vectors of length two that indicate the starting and ending (j,k)-coordinates of the subset of the face of the pencil across which the copying should take place. In this case these subsets are line segments (edges of the tile).

The above construction of the parallel code segment can itself be achieved in a piecemeal fashion. First, the serial code is rewritten to tile the index space

(this modification can be checked for correctness), after which the parallel/serial adaptors are inserted and lines 4, 5, and 6 are added to make the code segment parallel (line 7 must then reference distributed instead of nondistributed arrays). Even smaller steps are possible using Charon's wrapper functions [9].

An interesting feature of the parallel code is that communications appear functionally to occur only at the left and bottom edges of each tile. This runs counter to common experience in message passing codes, with receive calls for left and bottom edges, and send calls for right and top edges. The difference comes about because all processors participate in all communications, and each invocation of `CHN_Copy_faces` precipitates both send and receive calls on those processors whose data is involved. This keeps the structure of the code simple. In a subsequent optimization we can restrict tile visits to only those owned by the calling processor (eliminating the loops over all pencils in lines 2 and 3, and the test in line 6). In that case we must add calls to `CHN_Copy_faces` for the right and top edges as well. The resulting code is as efficient as the originally published NPB-MPI LU code [9].

4.2 NAS MG Parallel Benchmark

The NAS Multigrid (MG) Parallel Benchmark [1] is simpler than LU, SP, or BT, because the core solution procedure is explicit, and thus data parallel. However, a complication is that several levels of grid refinement must be created and managed, and the coarsest grids cannot be distributed among more than eight processors. In NPB-MPI we skip certain processors in the assignment of cells. For example, a 2×2×2-grid is distributed among only the even numbered processors in a 16-processor computation. This produces a good load balance, but leads to rather complex logic and communications. Although Charon easily supports this kind of custom data distribution, we choose a much simpler and equally efficient decomposition, illustrated in two dimensions in Fig. 4. We only show interior grid points. Grid cells, indicated by shading, are assigned to different processors.

We create fully distributed arrays for all grid levels that can be distributed uniformly among all processors. For all other grid levels, and also for the coarsest level that can still be distributed uniformly, we create nondistributed arrays (inherited from the serial code). When an operation involves two grid levels (interpolation or restriction), one of which cannot be fully distributed, we use `CHN_Redistribute` to map input or output arrays to the appropriate distributed or nondistributed version. Before interpolation from grid level two to three, we gather (redistribute) the distributed array at level three, so that the entire operation (involving both levels) takes place on nondistributed arrays. Afterwards, we scatter the result for use in other operations. Restriction is implemented similarly. The redistributions have trivial cost, since they are used when there is only one (interior) grid point per processor.

Performance results for the code, running on an SGI Origin2000 with 256 250MHz R10000 processors are compared with those for NPB-MPI on the same machine in Fig. 5. Classes A, B, and C refer to standard NPB problem sizes, ranging from 130^3 to 514^3 grid points. Within the range of scalability there is only a very small difference, pointing to the efficiency of Charon, and its validity as an

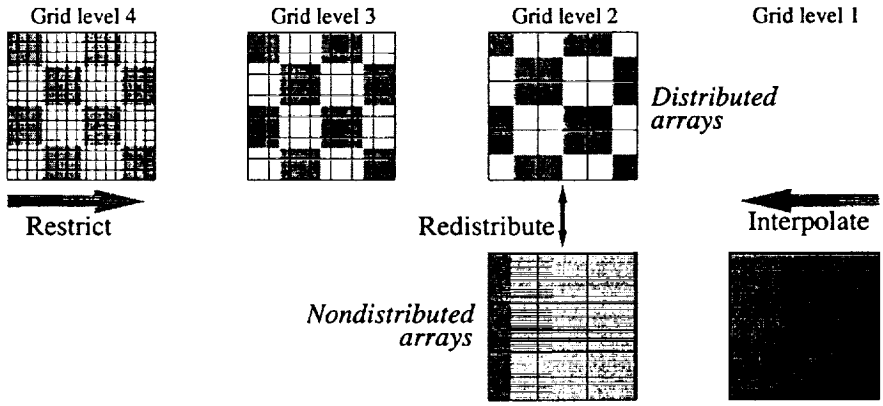

Figure 4. *Distributed and nondistributed arrays for 2D multigrid code at different levels of refinement. Each patch (indicated by shading) is assigned to a different processor.*

incremental parallelization tool.

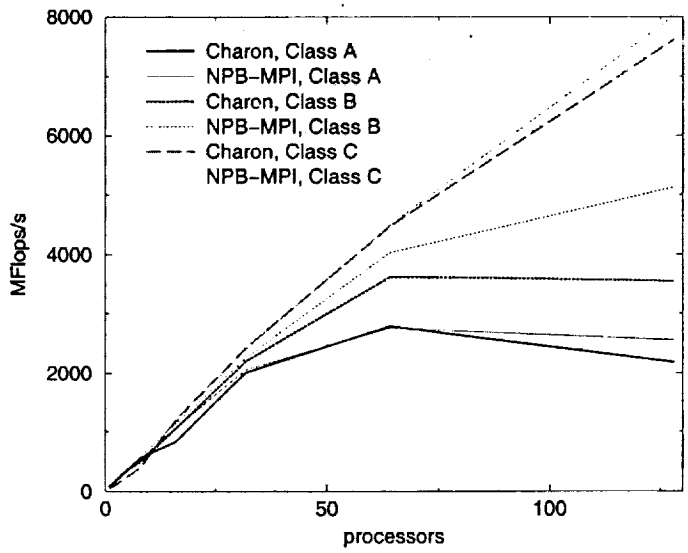

Figure 5. *Performance results for Charon and original NPB-MPI versions of MG code on SGI Origin*

4.3 Nonlocal boundary condition

Scientific programs for realistic problems need to accommodate realistic boundary conditions. Some of these are notoriously difficult to implement using message

passing. We take as an example the C-grid flow-through conditions for airfoil computations. The physical problem is shown schematically in Fig. 6a.

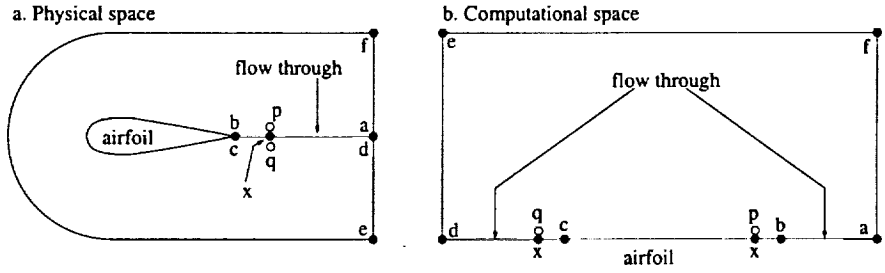

Figure 6. *C-grid with flow-through condition along branch cut*

A structured grid is 'wrapped' around the airfoil, such that the computationally distinct grid line segments a-b and d-c coincide in physical space, as indicated in Fig. 6b. To ensure single-valuedness of the solution, we compute the average of the flow solutions immediately above and below the cut and store this value at both grid points that physically coincide on the cut. The problem is that the two physically close contributors (p and q) to the value at a point x on the cut are not adjacent in computational space. They may reside on different processors. It is possible that contributions for one processor come from more than one other processors, or that some contributions are local while others are remote.

The simplest solution to the problem, requiring hardly any change to the serial code, lets one processor gather all the data, implement the boundary condition, and scatter the results. Gathers and scatters are accomplished by `CHN_Get_tile` and `CHN_Put_tile`. They specify a root processor that receives in or sends from a local buffer data corresponding to *any* Cartesian subset of a distributed array, regardless of data location. While this is not a scalable solution, it is often efficient enough if the number of points on the flow-through boundary is small. Further optimization is achieved by engaging more processors in the evaluation of the averages. Using query functions, each processor determines which of the contributing values are local, and then requests remote 'counterparts' to be fetched. This halves the communication volume, and also balances the computational load more evenly.

5 Conclusions

We have demonstrated, mostly through examples, the power and efficiency of Charon for incremental parallelization of scientific codes using message passing. While the library targets structured-grid applications, its concept, namely providing formalisms for the interpretation of user space in terms of high-level data abstractions, can be extended to other areas of scientific computing, such as unstructured-grid applications, which would then also become amenable to incremental parallelization.

Bibliography

[1] 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, December 1995.

[2] S. Balay, W.D. Gropp, L. Curfman McInnes, B.F. Smith, "PETSc 2.0 Users manual," Report ANL-95/11 - Revision 2.0.24, Argonne National Laboratory, Argonne, IL, 1999.

[3] 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," 8^{th} SIAM Conf. Parallel Proc. for Scientific Computing, Minneapolis, MN, March 1997.

[4] S.B. Baden, D. Shalit, R.B. Frost, "KeLP User Guid Version 1.3," Dept. Comp. Sci. and Engin., UC San Diego, La Jolla, CA, January 2000.

[5] T. Henderson, D. Schaffer, M. Govett, L. Hart, "SMS Users Guide," NOAA/Forecast Systems Laboratory, Boulder, CO, January 2000.

[6] 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.

[7] J. Nieplocha, R.J. Harrison, R.J. Littlefield, "The global array programming model for high performance scientific computing," SIAM News, Vol. 28, August-September 1995.

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

[9] R.F. Van der Wijngaart, "Charon message-passing toolkit for Scientific computations," 7^{th} Int'l Conf. High Performance Computing, Bangalore, India, December 17-20, 2000. see also:
http://www.nas.nasa.gov/-wijngaar/charon.