Charon message-passing toolkit for scientific computations

Rob F. Van der Wijngaart, Computer Sciences Corporation
NASA Ames Research Center, Moffett Field, CA 94035

Abstract: Charon is a library, callable from C and Fortran, that aids the conversion of structured-grid legacy codes—such as those used in the numerical computation of fluid flows—into parallel, high-performance codes. Key are functions that define distributed arrays, that map between distributed and non-distributed arrays, and that allow easy specification of common communications on structured grids. The library is based on the widely accepted MPI message passing standard. We present an overview of the functionality of Charon, and some representative results.

1 Introduction

A sign of the maturing of the field of parallel computing is the emergence of facilities that shield the programmer from low-level constructs such as message passing (MPI, PVM) and shared memory parallelization directives (P-Threads, OpenMP, etc.), and from parallel programming languages (High Performance Fortran, Split C, Linda, etc.). Such facilities include: 1) (semi-)automatic tools for parallelization of legacy codes (e.g. CAPTools [8], ADAPT [5], CAPO [9], SUIF compiler [6], SMS preprocessor [7], etc.), and 2) application libraries for the construction of parallel programs from scratch (KeLP [4], OVERTURE [3], PETSc [2], Global Arrays [10], etc.).

The Charon library described here offers an alternative to the above two approaches, namely a mechanism for *incremental* conversion of legacy codes into high-performance, scalable message-passing programs. It does so without the need to resort to explicit parallel programming constructs. Charon is aimed at applications that involve structured discretization grids used for the solution of scientific computing problems. Specifically, it is designed to help parallelize algorithms that are not naturally data parallel—i.e., that contain complex data dependencies—which include almost all advanced flow solver methods in use at NASA Ames Research Center. While Charon provides strictly a set of user-callable functions (C and Fortran), it can nonetheless be used to

convert serial legacy codes into highly-tuned parallel applications. The crux of the library is that it enables the programmer to codify information about existing multi-dimensional arrays in legacy codes and map between these non-distributed arrays and newly defined truly distributed arrays *at runtime*. This allows the programmer to keep most of the serial code unchanged and only use distributed arrays in that part of the code of prime interest. This code section, which is parallelized using more functions from the Charon library, is gradually expanded, until the entire code is converted. The major benefit of incremental parallalization is that it is easy to ascertain consistency with the serial code. In addition, the user keeps careful control over the amount of data transferred between processes, which is important on high-latency distributed-memory machines[1].

The usual steps that a programmer follows when parallelizing a code using Charon are as follows. First, define a distribution of the arrays in the program, based on a division of the grid(s) among all processors. Second, select a section of the code to be parallelized, and construct a so-called parallel bypass: map from the non-distributed (legacy code) array to the distributed array upon entry of the section, and back to the non-distributed array upon leaving it. Third, do the action parallelization work for the section, using more Charon functions.

The remainder of this paper is structured as follows. In Section 2 we explain the library functions used to define and manipulate distributed arrays (*distributions*), including those that allow the mapping between non-distributed and distributed arrays. In Section 3 we describe the functions that can be used actually to parallelize an existing piece of code. Some examples of the use and performance of Charon are presented in Section 4.

2 Distributed arrays

Parallelization of scientific computations using Charon is based on the concept of domain decomposition. Typically, one or more multi-dimensional grids are defined, and arrays—representing computational work—are associated with these grids. The grids are divided into nonoverlapping

[1]We will henceforth speak of *processors*, even though *processes* is the more accurate term.

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, as illustrated in Figure 1, and as described below.

First (Figure 1a), the logically rectangular discretization *grid* of a certain dimensionality and extent is defined, using `CHN_Create_grid`. The purpose of this step is to establish a geometric framework for all arrays associated with the grid. It also attaches to the grid an MPI [11] communicator, which serves as the context and processor subspace within which all subsequent Charon-orchestrated communications will 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 (Figure 1b), tessellations of the domain (*sections*) are defined, based on the grid variable. The associated library call is `CHN_Create_section`. Sections contain a number of cutting planes (*cuts*) along each coordinate direction. The grid is thus carved into a number of *cells*, each of which contains a logically rectangular block of grid points. Whereas the programmer can specify any number of cuts and cut locations, it is often sufficient to make a single call to a high-level routine to define all the cuts belonging to a particular kind of domain decomposition. For example, defining a section with just a single cell (i.e. a non-divided grid with zero cuts) is accomplished with `CHN_Set_solopartition_cuts`. Using `CHN_Set_unipartition_cuts` divides the grid evenly into as many cells as there are processors in the communicator.

Third (Figure 1c), the cells created in a section have to be assigned to processors, resulting in a *decomposition*. The associated library call is `CHN_Create_decomposition`. The reason why the creation of section and decomposition are separated is to provide flexibility. For example, the programmer may want to divide a grid into ten slices for execution on a parallel computer, but may want to assign all these slices to the same processor for the purpose of debugging on a serial machine. As in the case of the creation of sections, the programmer can choose to assign each cell to a processor individually, or make a single call to a high-level routine. For example, the function `CHN_Set_unipartition_owners` assigns each of the cells in the unipartition section

to a different processor. But regardless of the number of processors in the grid communicator, the function `CHN_Set_solopartition_owners` assigns all the cells in the section to the same processor.

Finally (Figure 1d), arrays with one or more spatial dimensions (same as the grid) are associated with a decomposition, resulting in *distributions*. The associated library call is `CHN_Create_distribution`. The arrays may represent scalar quantities at each grid point, or higher-order tensors. In the example in Figure 1d the tensor rank is 1, and the thusly defined vector has 5 components at each grid point. The data type of a distribution is one of a subset of the regular MPI data types (`MPI_REAL` in this case). Since Charon expressly supports the application of stencil operations on multi-dimensional grids, we also specify a number of so-called ghost points. These form a border of points (indicated by the shaded area) around each cell in the decomposition, which can be used as a cache to store data copied from adjacent cells. In this case the undivided distribution has zero ghost points, whereas the unipartition distribution has two ghost points, which can support higher-order difference stencils (e.g. a 13-point star in three dimensions, as shown in Figure 2).

The most salient aspect of distributions that sets Charon apart from other parallelization libraries is the fact that the programmer also supplies the memory occupied by the distributed array; the Charon distribution provides a *structuring interpretation* of user space. In the example in Figure 1d it is assumed that `arr` is the starting address of the actual, non-distributed array used in the legacy code, whereas `arrd` is the starting address of a newly declared array that will hold data related to the unipartition distribution. By mapping between `arr` and `arrd`—accomplished using `CHN_Redistribute` (see also Figure 1d)—we can dynamically switch from the serial legacy code to truly distributed code, and back. All that is required is that the programmer define distribution `arr_` such that the memory layout of the (legacy code) array `arr` coincide exactly with the Charon specified layout. This act of reverse engineering is supported by functions that allow the programmer to specify array padding and offsets, by a complete set of query functions, and by Charon's unambiguously defined memory layout model (see Section 3.3).

Although `CHN_Redistribute` can be used to construct parallel 'bypasses' of serial code of the kind demonstrated above, it can actually be used to map between *any* two compatible distributions (same grid, data type, and tensor rank). This is shown in Figure 3, where two stripwise distributions, one aligned with the first coordinate axis, and the other with the second, are mapped into each other, thereby establishing a dynamic transposition. This is useful when there are very strong but mutually incompatible data dependencies in different parts of the program (e.g. 2D Fast Fourier Transforms). By default, the unipartition decomposition divides all coordinate directions evenly, but by excluding certain directions from partitioning (`CHN_Exclude_partition_direction`) we can force a stripwise distribution.

3 Distributed and parallel execution support

While it is an advantage to be able to keep most of a legacy code unchanged and focus on a small part at a time for parallelization, it is often still nontrivial to arrive at good parallel code for complicated numerical algorithms. Charon offers support for this process at two levels.

The first concerns a collection of wrapping and utility functions that allows the programmer to keep the entire serial logic and structure of the legacy code unchanged, while making all assignments to elements of genuinely distributed arrays. These functions incur a significant overhead, and are meant to be removed in the final version of the code. They provide merely a stepping stone in the parallelization process, and may be skipped altogether by the more intrepid programmer.

The second is a collection of versatile and highly optimized bulk communication functions that support the implementation of sophisticated data-parallel and—more importantly—non-data-parallel numerical methods, such as various pipelined algorithms.

3.1 Wrapping functions

In a serial program it is immediately clear what the statement

```
a(i,j,k) = b(i+2,j-1,k)
```

means, provided a and b have been properly typed and dimensioned, but when these arrays are distributed across several processors the result is probably not what is expected, and most likely wrong. This is due to one of the fundamental complexities of message-passing, namely that the programmer is responsible for defining explicitly and managing the data distribution. Charon can relieve this burden by allowing the programmer to write the above assignment as

```
call CHN_Assign(CHN_Address(a_,i,j,k),CHN_Value(b_,i+2,j-1,k),ier)
```

with no regard for how the data is distributed (assuming that a_ and b_ are distributions related to arrays a and b). The benefit of this wrapping is that the user need not worry (yet) about communications, which are implicitly invoked by Charon, as needed.

The three functions introduced here have the following properties. `CHN_Value` inspects the distribution b_, determines which processor owns the grid point that holds the value—there can be only one owner—and broadcasts that value to all processors in the communicator ('*owner serves*' rule). `CHN_Address` inspects the distribution a_ and determines which processor owns the grid point that holds the value. If the calling processor is the point owner, the actual address—also called *lvalue*—is returned, and `NULL` otherwise[2]. `CHN_Assign` stores the value of its second argument at the address in the first argument **if** the address is not `NULL`. Consequently, only the point owner of left hand side of the assignment stores the value ('*owner assigns*' rule). No distinction is made between values obtained through `CHN_Value` and local values, or expressions containing any combination of each; all are *rvalues*. Similarly, no distinction is made between an address obtained through `CHN_Address` and a local address. Hence, it is perfectly legitimate to make the following assignments:

```
call CHN_Assign(CHN_Address(a_,i,j,k),5.0,ier)                          !1
call CHN_Assign(aux,CHN_Value(b_,i,j-1,k)+1.0,ier)                      !2
aux = CHN_Value(b_,i,j-1,k)+1.0                                         !3
call CHN_Assign(CHN_Address(a_,i,j,k),CHN_Value(a_,i,j,k)**2+1.0,ier)   !4
```

[2]In Fortran return values cannot be used as lvalues, but this problem can be easily circumvented, since the address is immediately passed to a C function.

It should be observed that assignments 2 and 3 are equivalent. Notice also that assignment 4 will yield the expected result: the expression `CHN_Value(a_,i,j,k)**2+1.0` is evaluated first, and then stored at the (unique) address `CHN_Address(a_,i,j,k)`. An important feature of wrapped code is that it is completely serialized. All processors execute the same statements, and whenever an element of a distributed array occurs on the right hand side of an assignment, it is broadcast[3]. As a result, it is guaranteed to have the correct serial logic of the legacy code.

3.2 Bulk communications

The performance of wrapped code can be improved by removing the need for the very fine-grained, implicitly invoked communications, and replacing them with explicitly invoked bulk communications. Within structured-grid applications the need for non-local data is often limited to (spatially) nearest-neighbor communication; stencil operations can usually be carried out without any communication, provided a border of ghost points (see Figure 1d) is filled with array values from neighboring cells. This fill operation is provided by `CHN_Copyfaces`, which lets the programmer specify exactly which ghost points to update. The function takes the following arguments:

- the thickness of layer of ghost points to be copied; this can be at most the number of ghost points specified in the definition of the *distribution*,
- the components of the tensor to be copied. For example, the user may wish only to transfer the diagonal elements of a matrix,
- the coordinate direction in which the copying takes place,
- the sequence number of the cut (defined in the *section*) across which copying takes place,

[3]The semantics of C and Fortran demand the use of broadcasts instead of point-to-point communications; if we could insist that the address parameter of the `CHN_Assign` call be evaluated before the rvalue expression, it would be possible to have the contributor(s) to the rvalue send their data directly to the processor(s) responsible for assigning it to a non-NULL address. However, the C and Fortran standards do not specify in which order actual parameters of subroutines are evaluated, so we must assume that they will be evaluated independently. This means that *all* processors must be able to evaluate the rvalue, necessitating the broadcasts.

- the rectangular subset of points within the cut to be copied.

In general, all processors within the grid's MPI communicator call `CHN_Copyfaces`, but those that have no work to do (because they do not own points involved in the copy operation) may safely skip the call. There are also wild cards for copying all components of a tensor, for copying across all cuts simultaneously, and for copying all points along a cut (see Figure 1b). A useful variation of `CHN_Copyfaces` is `CHN_Copyfaces_all`, which fills all ghost points of the distribution in all coordinate directions. It is the variation that is most commonly encountered in other parallelization packages for structured-grid applications, since it conveniently supports explicit, data parallel computations. But it is not sufficient to allow, for example, implementation of the pipeline algorithm described in Section 4.2.

The remaining two bulk communications provided by Charon are the previously described `CHN_Redistribute`, and `CHN_Gettile`. The latter copies a subset of a distributed array—which may be owned by several processors—into the local memory of a specified processor (cf. Global Arrays' `ga_get` [10]). This is useful for applications that have non-nearest-neighbor remote data dependencies, such as non-local boundary conditions for flow problems.

3.3 Parallelizing distributed code segments

Once the remote-data demand has been satisfied through the bulk copying of ghost point values, the programmer can instruct Charon to suppress broadcasts by declaring a section of the code *local* (see below). Within a local section not all code should be executed anymore by all processors, since assignments to points not owned by the calling processor will usually require remote data not present on that calling processor. Thus, the code must be restructured to restrict the index sets of loops over (parts of) the grid. This is the actual process of parallelization, and it is left to the programmer. It is usually conceptually simple for structured-grid codes, but the bookkeeping matters of changing all data structures at once throughout a code have traditionally hampered such parallelization. The advantage of using Charon is that the restructuring focuses on small segments of the code at any one time, and that the starting point is code that already executes

correctly on distributed data structures. The parallelization of a complicated loop nest typically involves the following steps.

1. Determine the order in which the cells in the grid should be visited to resolve all data dependencies in the target parallel code. For example, in the case of the x-sweep in the SP code described in Section 4.1 (see Figure 5), grid cells are visited layer by layer in a marching procedure in the positive x-direction. In this step all processors still visit all cells, and no explicit communications are required, thanks to the wrapper functions. This step is supported by Charon query functions that return the number of cells in a particular coordinate direction, and that return the starting and ending grid indices of these cells (useful for computing loop bounds).

2. Fill ghost point data in advance. If the loop is completely data parallel, a single call to `CHN_Copyfaces` or `CHN_Copyfaces_all` before entering the loop is usually sufficient to fill ghost point values. If a non-trivial data dependence exists, then multiple calls to `CHN_Copyfaces` are usually required. For example, in the x-sweep in the SP code `CHN_Copyfaces` is called between each layer of cells in the x-direction. At this stage all processors still execute all statements in the loop nest, so that they can participate in broadcasts of data not resident on the calling processor. However, whenever it is a ghost point value that is required, it is served by the processor that owns it, rather than the processor that owns the cell that has that point as an interior point. This seeming ambiguity is resolved by placing calls to `CHN_Begin_ghost_access` and `CHN_End_ghost_access` around the code that accesses ghost point data, which specify the index of the cell whose ghost points should be used.

3. Suppress broadcasts. This is accomplished by using the bracketing construct `CHN_Begin_local/CHN_End_local` to enclose the code that accesses elements of distributed arrays. For example:

```
call CHN_Begin_local(MPI_COMM_WORLD,ier)
call CHN_Assign(CHN_Address(a_,i),CHN_Value(b_,i+1)-CHN_Value(b_,i-1),ier)
```

```
call CHN_End_local(MPI_COMM_WORLD,ier)
```

At the same time, the programmer restricts accessing lvalues to points actually owned by the calling processor. This is supported by the query functions `CHN_Point_owner` and `CHN_Cell_owner`, which return the MPI rank of the processor that owns the point and the grid cell, respectively.

Once the code segment is fully parallelized, the programmer can strip off the wrappers to obtain the final, high-performance code. This stripping process effectively consists of translating global grid coordinates into local array indices, a chore that is again easily accomplished, due to Charon's transparent memory layout model. By default, all subarrays of the distribution associated with individual cells of the grid are dimensioned identically, and these dimensions can be computed in advance, or can be obtained through Charon query functions. For example, assume that the maximum number of cells owned by each processor in the computation is `nmax`, the dimensions of the largest cell in the grid are `nx`×`ny`, and the number of ghost points is `gp`. Then the array `flux` related to the scalar distribution `flux_` can be dimensioned as follows.

```
dimension flux(1-gp:nx+gp,1-gp:ny+gp,nmax)
```

Assume further that the programmer has filled the arrays `start(2,nmax)` and `end(2,nmax)` with the starting and ending point indices, respectively (using Charon query functions), of the cells in the grid owned by the calling processor. Then the following two loop nests are equivalent, provided the calling processor owns at least `n` cells.

```
do j = start(2,n), end(2,n)
 do i = start(1,n), end(1,n)
   call CHN_Assign(CHN_Address(flux_,i,j),5.0*(i+j),ier)
 end do
end do


do j = 1, end(2,n)-start(2,n)+1
 do i = 1, end(1,n)-start(1,n)+1
```

```
    flux(i,j,n) = 5.0*(i+j)
  end do
 end do
```

The above example illustrates the fact that Charon minimizes encapsulation; it is always possible to access data related to distributed arrays directly, without having to copy data or call specialized access functions. This is a programming convencience, as well as a performance gain. Most significantly, programs parallelized using Charon usually ultimately only contain library calls that create and query distributions, and that perform high-level communications.

Finally, it should be noted that it is not necessary first to wrap legacy code to take advantage of Charon's bulk communication facilities for the construction of parallel bypasses. Wrappers and bulk communications are completely independent.

4 Examples

We present two examples of numerical problems, SP and LU, with complicated data dependencies. Both are taken from the NAS Parallel Benchmarks (NPB) [1], of which hand-coded MPI versions (NPB-MPI) and serial versions are freely available. They have the form: $Ax^{n+1} = b(x^n)$, where x is the time-dependent solution, n is the number of the time step, and b is a nonlinear 13-point stencil operator (see Figure 2). The difference is in the shape of A, the discretization matrix that defines the 'implicitness' of the numerical scheme. For SP it is effectively: $A_{SP} = L_z L_y L_x$, and for LU: $A_{LU} = L_+ L_-$.

4.1 SP code

L_z, L_y and L_x are fourth-order difference operators that determine data dependencies in the z, y, and x directions, respectively. A_{SP} is numerically inverted in three corresponding phases, each of which involves the solution of a large number of independent banded (penta-diagonal) matrix equations, three for each grid line. Each equation is solved using Gaussian elimination without pivoting, which is implemented as two sweeps along the grid line, one in the positive

direction to carry out the forward elimination, and one in the negative direction to complete the backsubstitution. The method chosen in NPB-MPI is the so-called multipartition (MP) decomposition strategy. It assigns to each processor multiple blocks of grid points in such a manner that, regardless of sweep direction, each processor has work to do during each stage of the solution process; the load is fully balanced. An example of a 9-processor 3D MP is shown in Figure 4. Details can be found in [1]. Whereas most of the packages in the literature that we have studied do not allow the definition of MP, it is easily specified in Charon, i.e.,

```
call CHN_Create_section(multi_sec,grid,ier)
call CHN_Set_multipartition_cuts(multi_sec,ier)
call CHN_Create_decomposition(multi_cmp,multi_sec,ier)
call CHN_Set_multipartition_owners(multi_cmp,ier)
```

The solution process for a particular coordinate direction, say x, is as follows (see Figure 5). All processors start the forward elimination on the left side of the grid. When the boundary of the first layer of grid blocks is reached, the elements of the penta-diagonal matrix that need to be passed to the next layer of blocks are copied in bulk using `CHN_Copyfaces`. Then the next layer of blocks is traversed, followed by another copy operation, etc. The number of floating point operations and words communicated in this fashion in the Charon version of the code is exactly the same as in NPB-MPI. The only difference is that Charon copies the communicated values into ghost points, whereas in NPB-MPI they are immediately used to update the matrix system without going to main memory. The latter is more efficient, but comes at the cost of a much greater program complexity, since the communication must be fully integrated with the computation.

The results of running both Charon and NPB-MPI versions of SP for three different grid sizes on an SGI Origin2000 (250 MHz MIPS R10000 processor) are shown in Figure 6. Save for a deterioration of performance of the Charon code at 81 processors for class B (102^3 grid) due to a bad stride, the results indicate that the Charon version achieves approximately 70% of the performance of NPB-MPI, with roughly the same scalability characteristics. While this difference

is significant, it should be noted that the Charon version was derived from the serial code in approximately three days, whereas NPB-MPI took more than one month (both by the author of this paper). Moreover, since Charon and MPI calls can be freely mixed, it is always possible for the programmer who is not satisfied with the performance of Charon communications to do (some of) the message passing by hand.

Interestingly, if SP is run on a 10-CPU Sun Ultra Enterprise 4000 Server (250 MHz Ultrasparc II processor), whose cache structure differs significantly from the Origin's, the results for the Charon and NPB-MPI versions are almost identical, meaning that there is no gain in performance through hand coding. The graphical Sun results for Class A (grid size 64^3 points) are shown in Figure 7, along with the results for the same grid on the SGI Origin machine, for comparison.

4.2 LU code

L_- and L_+ are first-order direction-biased difference operators. They define two sweeps over the entire grid. The structure of L_- dictates that no point (i, j, k) can be updated before updating all points (i_p, j_p, k_p) with smaller indices: $\{(i_p, j_p, k_p) | i_p \leq i, j_p \leq j, k_p \leq k, (i_p, j_p, k_p) \neq (i, j, k)\}$. This data dependency is the same as for the Gauss-Seidel method with lexicographical point ordering. L^+ sweeps in the other direction. Unlike in the SP case, there is no concept of independent grid lines for LU. The solution method chosen for NPB-MPI is to divide the grid into pencils, one for each processor (see Figure 8), and pipeline the solution process (see Figure 9). Each unit of computation is a single plane of points (a *tile*) of the pencil. Once a tile is updated, the updated points on its boundary are communicated to the pencil's Eastern and Northern (for L_-) neighbors. Subsequently, the next tile in the pencil is updated. Of course, not all boundary points of the whole pencil should be updated after completion of each tile update, but only the boundary points of the 'active' tile. This is easily specified in `CHN_Copyfaces`.

The results of both Charon and NPB-MPI versions of LU for three different grid sizes on an SGI Origin2000 250 MHz system are shown in Figure 10. Now the performance of the Charon code is almost the same as that of NPB-MPI. This is because both use ghost points for transferring

information between neighboring pencils.

Again, if the problem is run on the Sun Enterprise 4000, the performances of the hand coded and Charon parallelized programs are nearly indistinguishable.

5 Discussion and conclusions

We have given a brief presentation of some of the major capabilities of Charon, a parallelization library for scientific computing problems. It is useful for those applications that need high scalability, or that have complicated data dependencies that are hard to resolve by analysis engines. Since some hand coding is required, it is more labor intensive than using a parallelizing compiler or code transformation tool. Moreover, the programmer must have some knowledge about the application program structure in order to make effective use of Charon. When the programmer does decide to make the investment to use the library, the results are close to the performance of hand-coded, highly tuned message passing implementations, at a fraction of the development cost. Future work on the library will include a graphical user interface that assists in the wrapping of legacy code statements. More information and a user guide are being made available by the author [12].

References

[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] M. Frumkin, J. Yan, "Automatic Data Distribution for CFD Applications on Structured Grids," NAS Technical Report NAS-99-012, NASA Ames Research Center, CA, 1999.

[6] 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, Vol. 29, pp. 84–89, December 1996.

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

[8] 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.

[9] H. Jin, M. Frumkin, J. Yan, "Use Computer-aided tools to parallelize large CFD applications," NASA High Performance Computing and Communications Computational Aerosciences (CAS) Workshop 2000, NASA Ames Research Center, Moffett Field, CA, February 2000.

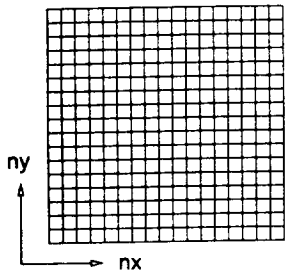
[10] J. Nieplocha, R.J. Harrison, R.J. Littlefield, "The global array programming model for high performance scientific computing," SIAM News, Vol. 28, August-September 1995.

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

[12] R.F. Van der Wijngaart, Charon home page, http://www.nas.nasa.gov/~wijngaar/charon.

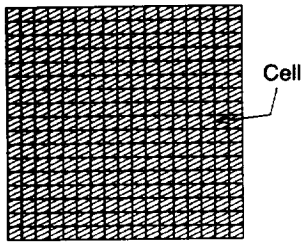
a. Define logical grid (Step 1)

```
CHN_Create_grid(grd,MPI_COMM_WORLD,2);
CHN_Set_grid_size(grd,0,nx);
CHN_Set_grid_size(grd,1,ny);
```

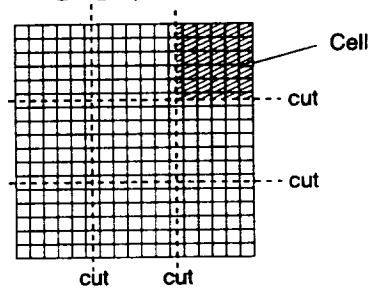


b. Define sections (tessellations) based on grid (Step 2)

```
CHN_Create_section(solo_sec,grd);
CHN_Set_solopartition_cuts(solo_sec);
```



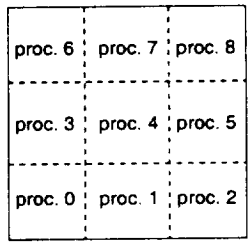

```
CHN_Create_section(uni_sec,grd);
CHN_Set_unipartition_cuts(uni_sec);
```



c. Define decompositions (processor assignments) based on sections (Step3)

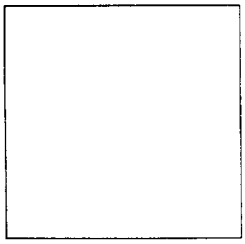
```
CHN_Create_decomposition(solo_cmp,solo_sec);
CHN_Set_solopartition_owners(solo_cmp,0);
```


```
CHN_Create_decomposition(uni_cmp,uni_sec);
CHN_Set_unipartition_owners(uni_cmp);
```



d. Define distributed arrays (exploded view) based on decompositions (Step 4)

```
CHN_Create_distribution(arr_,solo_cmp,MPI_REAL,arr,0,1,5);
```



No ghost points

```
CHN_Redistribute(arrd_,arr_);
```

scatter

gather

```
CHN_Redistribute(arr_,arrd_);
```

```
CHN_Create_distribution(arrd_,uni_cmp,MPI_REAL,arrd,2,1,5);
```

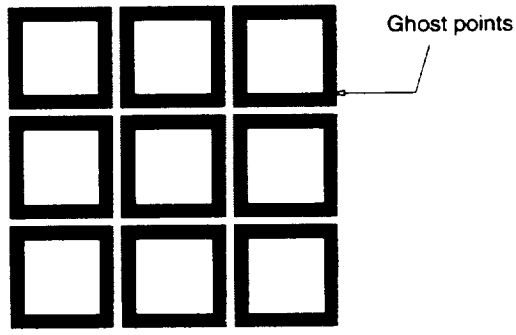


Figure 1: Defining and mapping distributed arrays

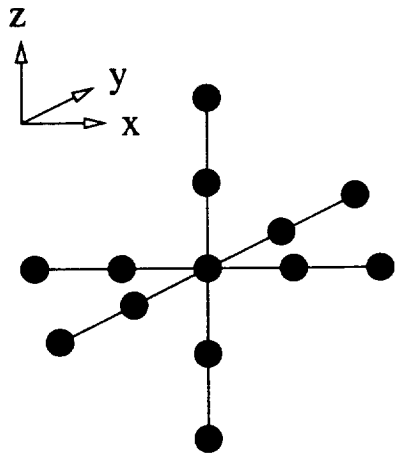


Figure 2: 13-point star-shaped difference stencil

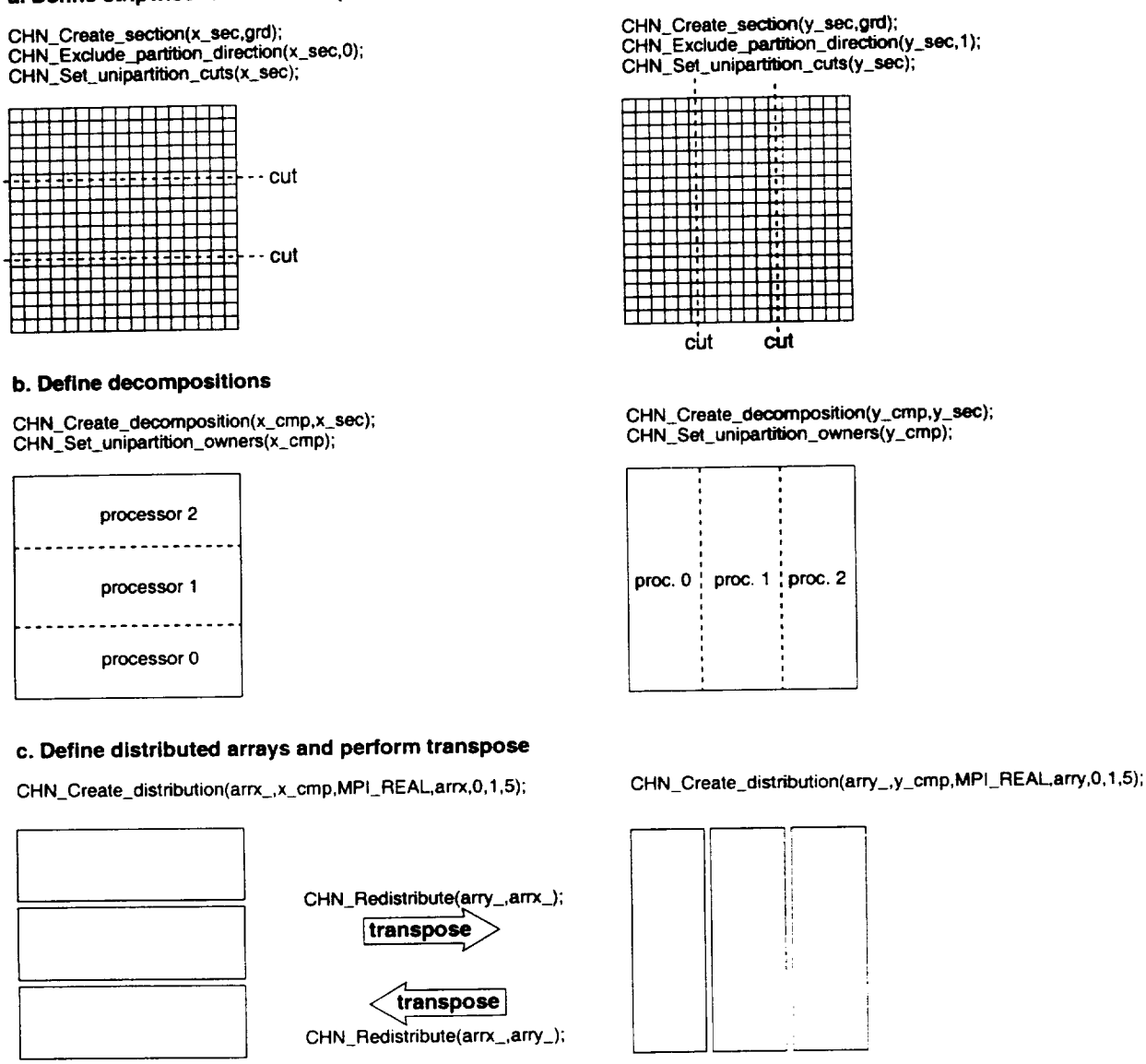


Figure 3: Transposing distributed arrays

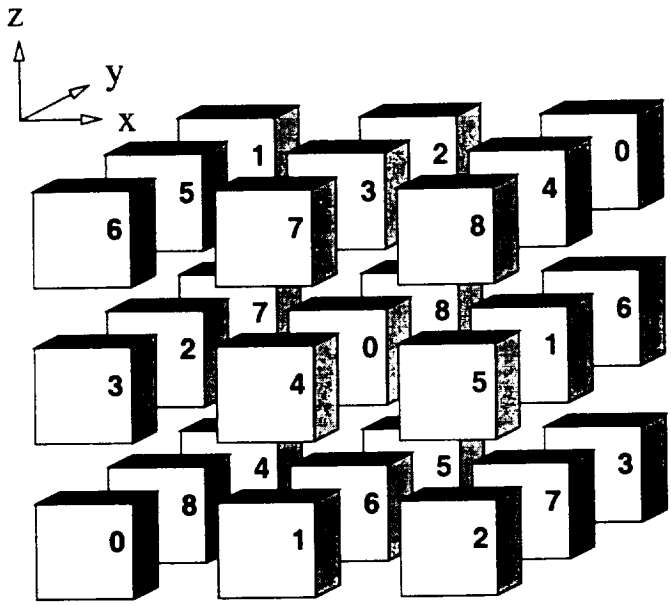


Figure 4: 3D Multipartition decomposition (9 processors); indices indicate processor ownership

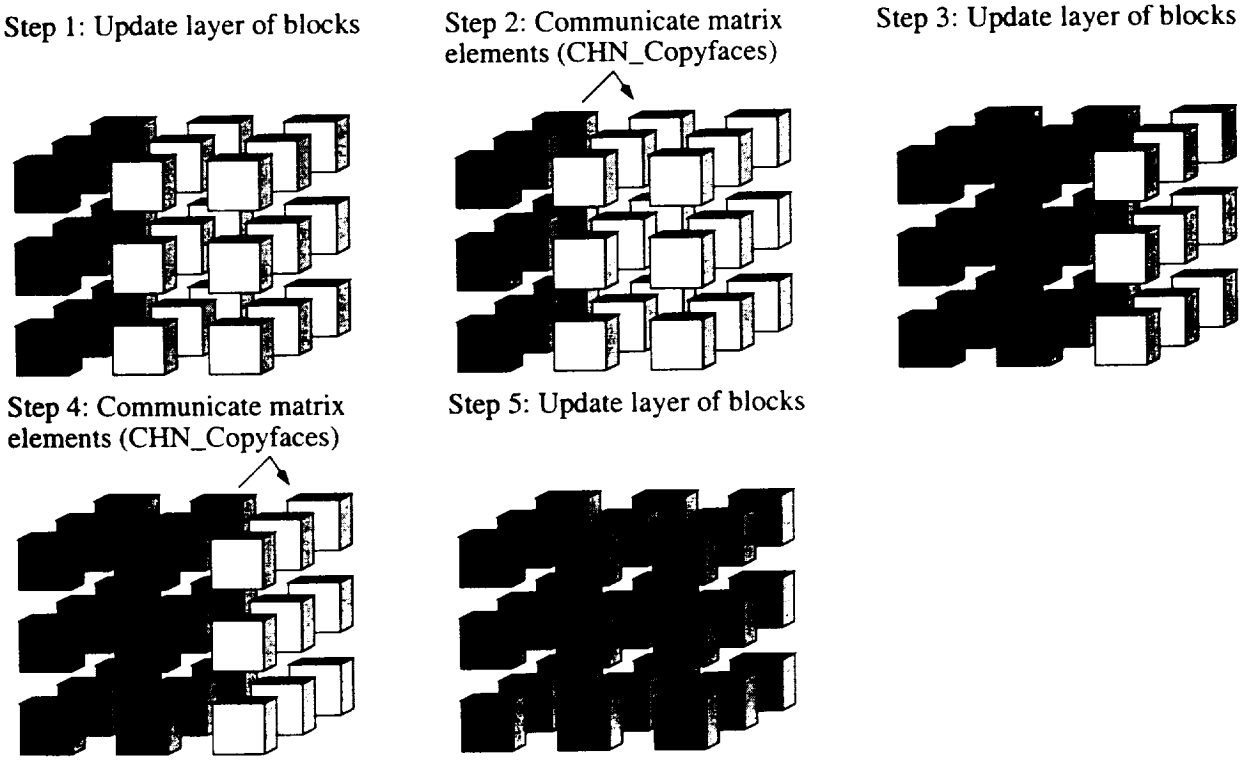


Figure 5: Multipartition solution process for SP code (L_x: forward x-sweep)

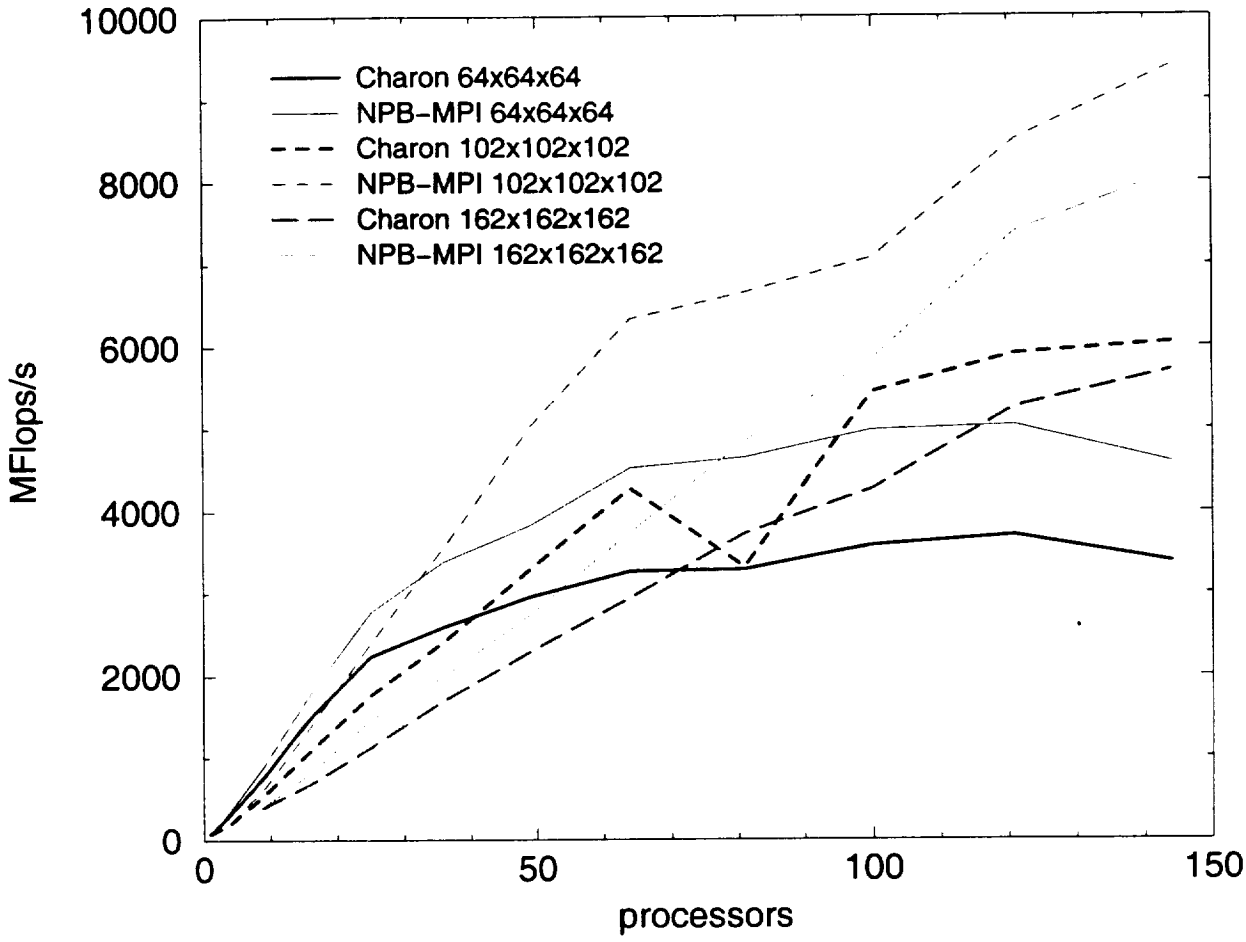


Figure 6: Performance of NAS Scalar Penta-diagonal benchmark on 250 MHz SGI Origin2000

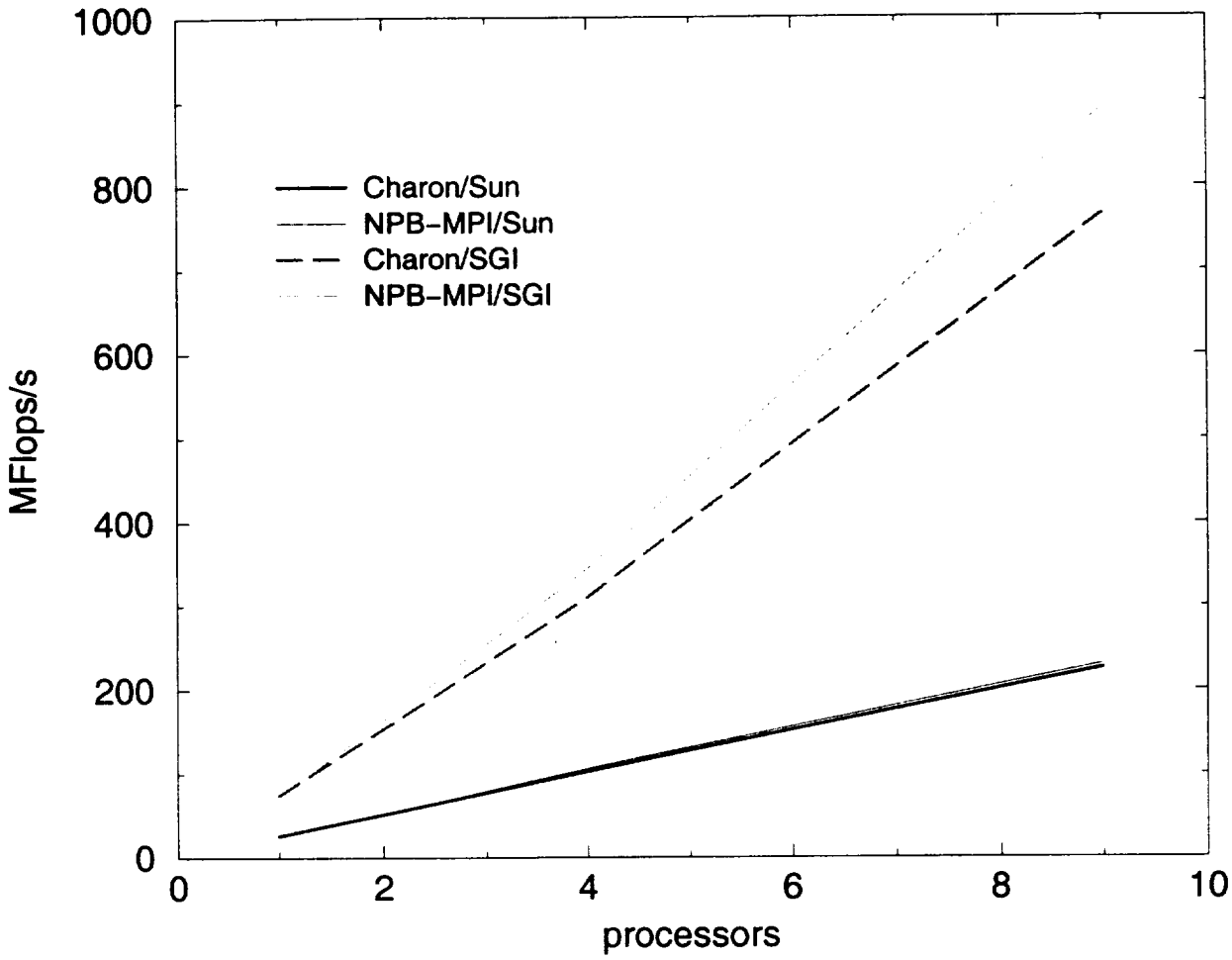


Figure 7: Performance of NAS Scalar Penta-diagonal benchmark on Sun Enterprise 4000 server, compared with SGI Origin2000 (64^3 grid)

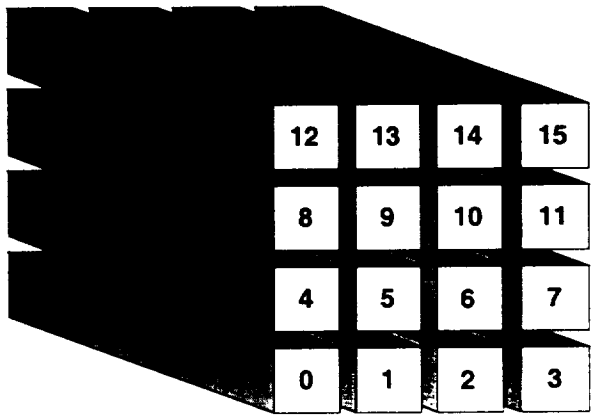


Figure 8: Unipartition pencil decomposition (16 processors); indices indicate processor ownership

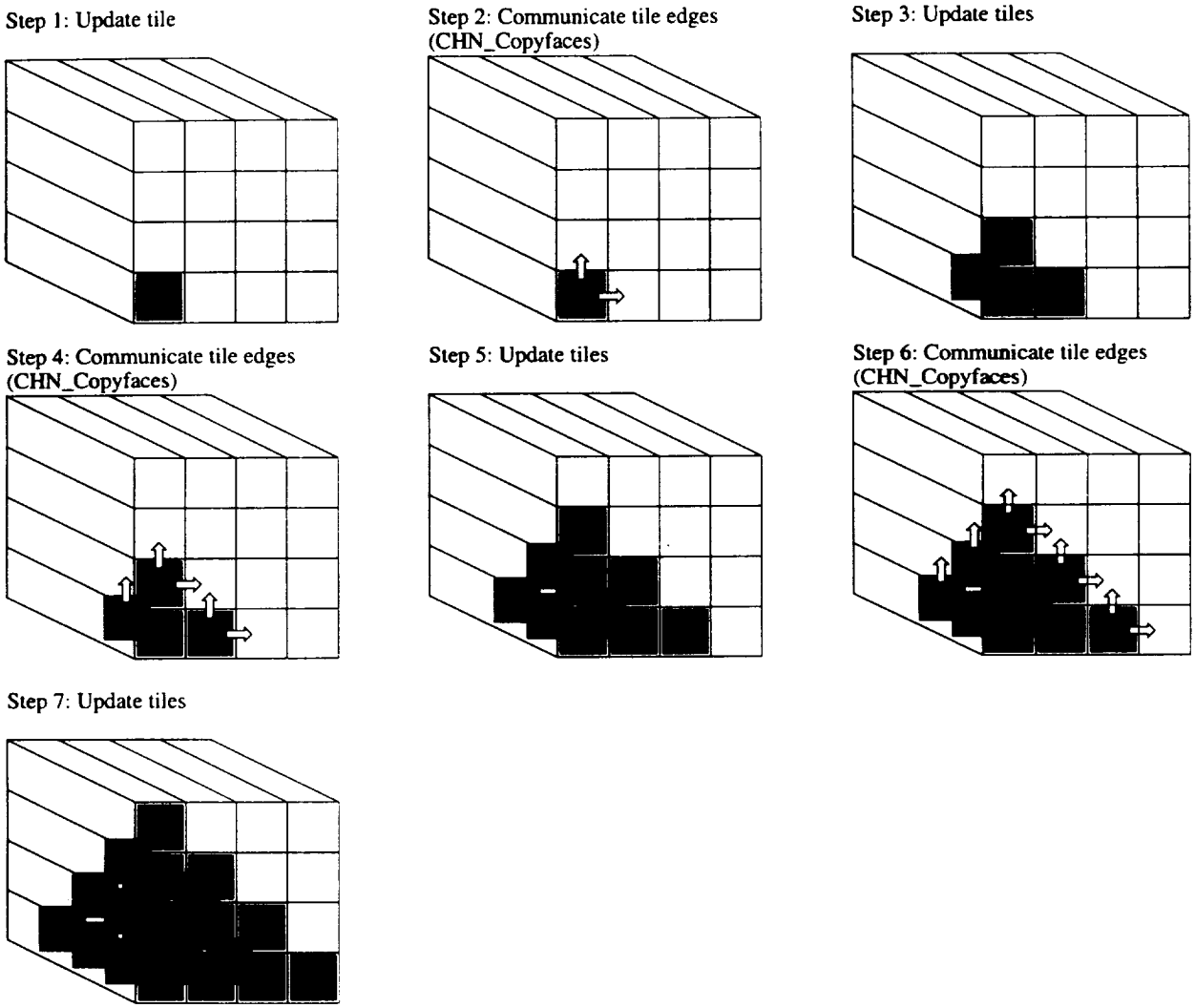


Figure 9: Start of pipelined solution process for LU code ($L\_$)

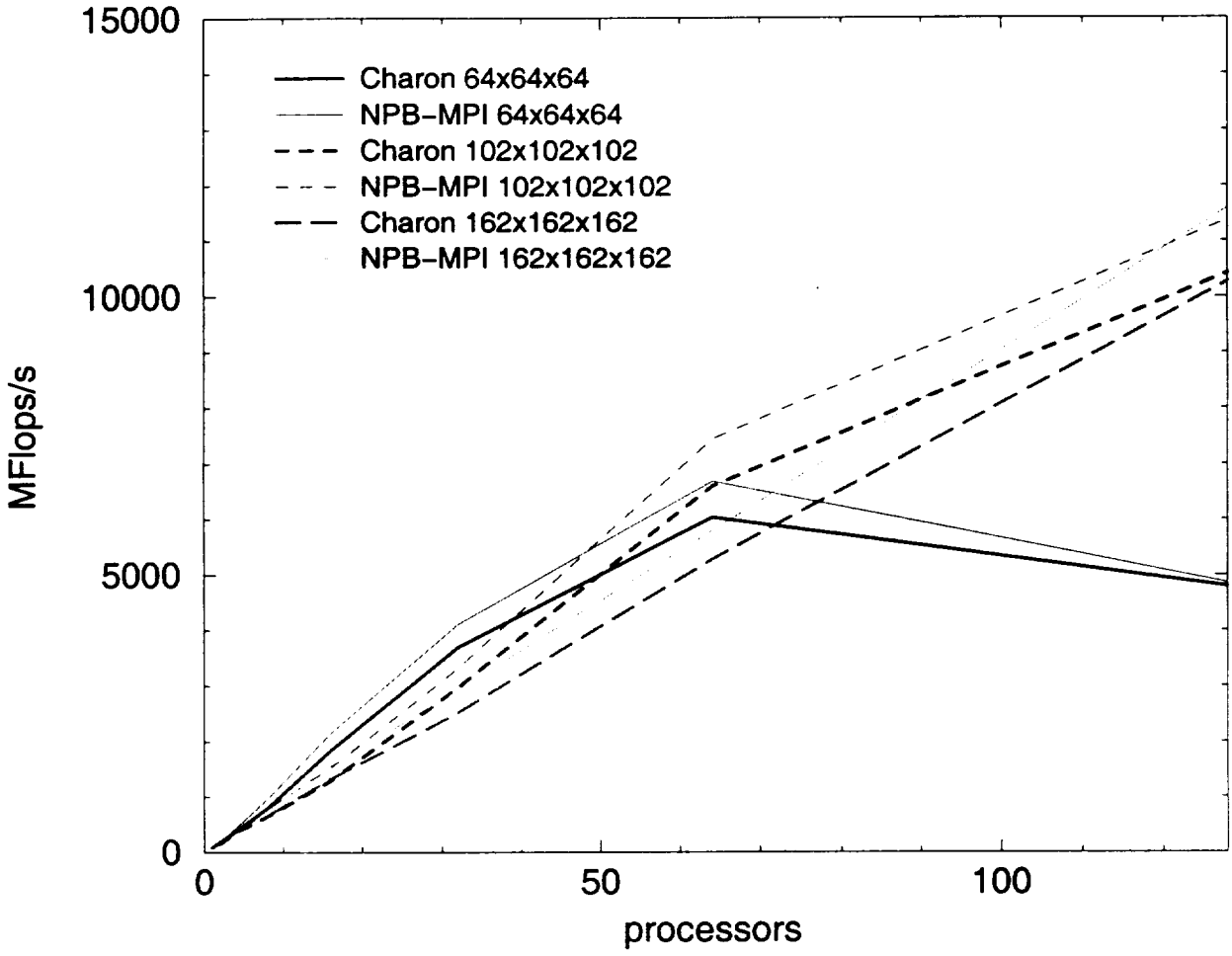


Figure 10: Performance of NAS Lower Upper benchmark on 250 MHz SGI Origin2000