of the expected range. Each scalar value is used to define a dynamic singleton region in line 17. This region controls a shattered conditional which is entered only if `NumVals` has not exceeded the bucket size. Since `NumVals` is a 1D array, the enclosing region scope for this conditional is the singleton region `[value]`. This implies that the conditional will only be evaluated on the processor that owns index `value`, and that its unique `NumVals` element will be read. If the processor's bucket is not yet full, `NumVals` is incremented and the scalar value is stored in its bucket at the corresponding location. Then a new value is read and the loop continues.

Note that such a program would be difficult to write without grid dimensions. For example, in traditional ZPL, it would be difficult to store a unique count of values per processor due to the fact that scalars and flood dimensions must contain consistent values for all processors. Possible approaches would be to use a traditional region and have multiple buckets per processor or to set its size to be equal to the number of processors. However, neither of these solutions is quite as elegant as the one supported by grid dimensions.

3.11 Related Work

This section describes other approaches to parallel programming—in particular, libraries and languages that support local and global views. Due to the sheer volume of parallel programming systems that have been developed in the past few decades, this section concentrates on those approaches that are most notable, that are in active use and development today, and that are most closely related to ZPL. The array access mechanisms and support for a performance model are considered for each approach. In addition, most descriptions include a code sample excerpting the main loop from the Jacobi iteration as written in the language to serve as an example. These codes were written by experts in the languages and have been modified only for readability. References for the original code sources are given in their captions. Many of the implementations vary slightly in terms of the algorithm's details, but the expression of the general Jacobi technique will typically be recognizable.

3.11.1 Local-View Libraries

As described in the introduction, local-view libraries are those that give the programmer an interface which allows them to specify the per-processor behavior of a parallel machine. These libraries typically provide several methods of transferring data between processors. This section considers some of the primary examples.

Note that since local-view libraries are used in conjunction with traditional programming languages like C or Fortran 90, the array access methods available to the programmer are those supported by the base language. In addition, as local-view languages, since programmers are responsible for explicitly specifying both the distribution of data and work as well as the program's communication, the performance model is inherent in the code's implementation.

MPI

MPI [Mes94] stands for the *Message Passing Interface*. It is a library specification for message-passing-based routines between cooperating processes. The initial specification of MPI (version 1.1) supports point-to-point communications (sends and receives) in a multitude of varieties (including blocking, non-blocking, buffered, ready-send, and send-receive). In addition, MPI supports higher-level operations such as broadcasts, all-to-all scatters and gathers, reductions, scans, and barrier synchronizations. MPI is a highly structured interface, allowing users to specify new datatypes and to organize processors into different groups. MPI version 2 adds support for additional features such as one-sided communications, process creation and management, extended collective operations, and I/O [Mes97].

MPI has succeeded in becoming the *de facto* standard for parallel programming due to its portability and widespread availability. MPI is supported on almost every parallel platform, and free implementations are available for most commodity platforms [GLDS96, Ohi96].

Listing 3.6: MPI Jacobi Implementation Excerpt [Gro01]

```
/* Main loop of Jacobi Iteration */
do {
  /* Send up unless I'm at the top, then receive from below */
  /* Note the use of xlocal[i] for &xlocal[i][0] */
  if (rank < size - 1)
    MPI_Send(xlocal[maxn/size], maxn, MPI_DOUBLE, rank + 1, 0,
             MPI_COMM_WORLD);
  if (rank > 0)
    MPI_Recv(xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0,
             MPI_COMM_WORLD, &status);

  /* Send down unless I'm at the bottom */
  if (rank > 0)
    MPI_Send(xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1,
             MPI_COMM_WORLD);
  if (rank < size - 1)
    MPI_Recv(xlocal[maxn/size+1], maxn, MPI_DOUBLE, rank + 1, 1,
             MPI_COMM_WORLD, &status);

  /* Compute new values (but not on boundary) */
  diffnorm = 0.0;
  for (i=i_first; i<=i_last; i++)
    for (j=1; j<maxn-1; j++) {
      xnew[i][j] = (xlocal[i][j+1] + xlocal[i][j-1] +
                    xlocal[i+1][j] + xlocal[i-1][j]) / 4.0;
      diffnorm += (xnew[i][j] - xlocal[i][j]) *
                  (xnew[i][j] - xlocal[i][j]);
    }
  /* Only transfer the interior points */
  for (i=i_first; i<=i_last; i++)
    for (j=1; j<maxn-1; j++)
      xlocal[i][j] = xnew[i][j];

  MPI_Allreduce(&diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM,
                MPI_COMM_WORLD);
  gdiffnorm = sqrt(gdiffnorm);
} while (gdiffnorm > 1.0e-2);
```

One of MPI's primary disadvantages is that message-passing is not always the cheapest means of moving data around a parallel machine. For example, message-passing semantics often require buffering and synchronization that are not inherently required by shared memory or shared address space machines. This can result in overheads beyond those required for communication on a given architecture.

This problem tends to be combatted by expanding the MPI interface, particularly by the addition of one-sided communications in MPI-2. The problem is that such generalizations are at odds with portability. In particular, while an MPI program may be written in a style that is suitable for a particular machine (*e.g.*, a one-sided message passing style on the Cray T3E), its suitability for other machines is not guaranteed, forcing the programmer to consider changing the MPI calls for each platform.

Note that this problem is not particular to MPI, but is rather a seemingly inevitable problem with any interface that specifies particular communication semantics and yet wants to run on a diverse set of parallel architectures. Chapter 4 introduces ZPL's paradigm-neutral interface that manages to avoid this paradigm-mismatch problem.

In spite of its drawbacks, MPI should be considered a success, due not only to its popularity, but also the fact that it is enabling more people to write parallel programs today than any other approach described in this dissertation (including ZPL, regrettably).

Listing 3.6 shows the inner loop of the Jacobi iteration written using C and MPI for a 1D processor grid. Lines 5–18 implement the communication required to exchange boundary values between processors to the north and south. Note that a 2D decomposition would not only require similar lines for east and west communications, but also code to marshall the data in and out of temporary buffers, since those elements would not be adjacent in memory. Lines 21–32 perform the main computation and update. Lines 34–35 use an MPI reduction routine to calculate the global normalized difference.

PVM

PVM stands for *Parallel Virtual Machine* [BDG+91] and could be considered the awkward cousin of MPI. A rough characterization of PVM might describe it as being a quickly-built, practical solution to parallel programming which gradually became more sophisticated as time passed. In contrast, MPI was developed as a standard for which complete implementations have gradually been developed (for example, a complete implementation of MPI-2 is not yet available as of this writing). One effect of this is that while each MPI routine takes a dozen parameters allowing for type information, error conditions, processor sets, identifiers, and the actual data to be described, PVM's interface tends to be somewhat simpler and less general. In spite of this, PVM supports many of the same communication types as MPI, including blocking and non-blocking sends and receives, process creation and management, broadcasts, reductions, and barrier synchronizations. PVM has similar portability problems as MPI, but has not taken the same approach of diversifying the available communication styles. This may prevent it from performing as well on diverse architectures, but prevents the interface from being as confusing.

PVM is freely available for most standard platforms [PVM01], and should be considered a reasonable and somewhat simpler alternative to using MPI. A PVM implementation of Jacobi would look extremely similar to the MPI version in Listing 3.6, simply replacing the MPI calls with their equivalent PVM routines.

SHMEM

The SHMEM interface [BK94] is another inter-process communication library that was developed for the Cray T3D in order to expose the low-overhead one-sided communications supported by its architecture. In addition to one-sided *puts* and *gets*, SHMEM supports higher-level operations such as broadcasts, reductions, and barrier synchronizations.

One disadvantage of the SHMEM interface is that some characteristics of the Cray T3D architecture are embedded in the routines themselves. For example, collective operations such as reductions and broadcasts can only be performed on subsets of processors that are strided by powers of two. Thus, to perform a reduction or broadcast over a general set of processors, users must write their own routines using puts and gets.

Another disadvantage of SHMEM is again one of paradigm/architecture mismatches. SHMEM's routines are ideal for shared address space machines like the Cray T3D and T3E which support one-sided communications. It is also appropriate for shared memory machines, on which such routines can easily be implemented. However, on distributed memory architectures that have no hardware support for writing directly to a processor's local memory, some amount of overhead is required to watch for incoming messages and store them to memory appropriately. Though there have been some early attempts to support such implementations [BB00], it has yet to be proven that one-sided communication interfaces can support low-overhead data transfers on such architectures.

A SHMEM version of Jacobi would also look very similar to the MPI version, except that each pair of send/receive calls would be replaced by a single `shmem_put()` or `shmem_get()` call that specifies both the local and remote addresses. SHMEM's symmetric memory allocation routines would be used to avoid explicitly transferring addresses between processors.

Other Local-View Libraries

Two other local-view libraries that deserve brief mention are Intel's NX library [Pie93] and the Active Messages work by Culler, von Eicken, *et al.* [vECGS96]. NX is yet another message-passing standard, developed for parallel Intel machines like the Paragon. It has largely faded from use with the passing of those platforms. NX differs somewhat from MPI and PVM in that it provides routines not only for blocking and non-blocking communications, but also for sends and receives with *callback routines* that are executed when a message is successfully sent or received. These routines are ideally run on a dedicated co-processor to perform actions such as unpacking the message into memory while the original program thread continues unhindered.

Active Messages represents a related idea in which a function pointer is bundled into a message along with its data. Upon receiving the message, the receiving processor calls the specified function, allowing the message to be handled in a way that best suits the details of the architecture. As such, Active Messages have been touted as an extremely portable, low-overhead communication strategy, suitable for use in implementing higher-level languages such as Split-C (described in the next section).

3.11.2 Local-View Languages

Split-C

Split-C [CDG+93, CDG+95] is an extension to C that was designed to support parallel programming. In doing so, its designers were motivated to provide efficient access to the underlying machine with no surprises, much as in traditional C (and ZPL). Split-C supports a global address space across all processors as well as global pointers that can refer to objects located anywhere in the address space (*i.e.*, on any processor). Global pointers are represented using a 2-tuple containing a processor number and a local address on that processor. Arrays, whether pointer-based or C-style arrays, can be declared to be *spread*, which causes indices in the spread dimensions to be distributed cyclically across the processor set.

Split-C also supports a number of novel assignment operators that reduce the synchronization requirements of a traditional assignment. For example, the split-phase assignment (`:=`) allows a non-local value to be assigned to a local value or vice-versa. This operator specifies that such an assignment should take place and initiates the communication required to carry it out. However, rather than waiting for the assignment to complete, it simply proceeds to the next statement. To ensure that a split-phase assignment has completed, programmers use synchronization calls that block until all preceding split-phase assignments are done. In this sense, split-phase assignment offers a language-level equivalent to one-sided communication.

In spite of being a relatively high-profile and well-published language, Split-C has never become widely used in the community, possibly due to the fact that its one-sided communication style was difficult to implement efficiently on distributed memory platforms.

Co-Array Fortran

Developed at Cray Research, Co-Array Fortran (CAF) [NR98] is another parallel language developed by extending a traditional language, in this case Fortran 90. The primary addition to the language is the concept of the *co-array*. Co-array dimensions are simply extra array dimensions that refer to processor space rather than data space. For example, a scalar variable declared with a co-array dimension causes each processor to allocate a copy of that variable, much like ZPL's grid arrays. Appending co-array dimensions to traditional arrays results in a blocked parallel array. Remote data is referred to by indexing into a co-array dimension with the remote processor's index. This serves as a concise representation of interprocessor communication that is simple, yet extremely elegant and powerful. CAF also provides a number of synchronization operations which are used to maintain a consistent global view of the problem.

Listing 3.7 shows a Jacobi implementation in CAF. The co-array references are the expressions in square brackets on lines 8–11, 30, and 35. All of these references are on the right-hand side of the assignment, indicating that a get-style communication is used to access the remote processor's memory. The initial lines exchange boundary values between processors, while the final pair are used to implement an $\Theta(p)$ reduction and broadcast. Synchronization is used in lines 3–7, 27, and 34 to ensure that all computations have completed before the data transfers take place. The rest of the code is a straightforward local-view implementation.

Listing 3.7: CAF Jacobi Implementation Excerpt [Wal01]

```
      DO
!       update halo.
        IF (MIN(P, Q) >= 3) THEN
          CALL SYNC_ALL( WAIT=NEIGHBORS )
        ELSE
          CALL SYNC_ALL()
        ENDIF ! neighbor images have ANS(1:NN, 1:MM) up to date
        ANS(1:NN, MM+1) = ANS(1:NN, 1   )[ME_P,  ME_QP]  ! north
        ANS(1:NN,    0) = ANS(1:NN,   MM)[ME_P,  ME_QM]  ! south
        ANS(NN+1, 1:MM) = ANS(1,    1:MM)[ME_PP, ME_Q ]  ! east
        ANS(   0, 1:MM) = ANS(  NN, 1:MM)[ME_PM, ME_Q ]  ! west

!       5-point stencil is correct everywhere,
!       since halo is up to date.
        DO J= 1, MM
          DO I= 1, NN
            WRK(I, J) = (1.0/4.0) * (ANS(I-1, J  ) + &
                                     ANS(I+1, J  ) + &
                                     ANS(I  , J-1) + &
                                     ANS(I  , J+1)   )
          ENDDO
        ENDDO

!       calculate global maximum residual error.
        PMAX = MAXVAL( ABS( WRK(1:NN, 1:MM) - ANS(1:NN, 1:MM)))
        CALL SYNC_ALL()  ! protects both PMAX and ANS
        IF (ME == 1) THEN
          DO I= 2, NUM_IMAGES()
            PMAXI = PMAX[I]
            PMAX  = MAX( PMAX, PMAXI )
          ENDDO
        ENDIF
        CALL SYNC_ALL( WAIT=(/1/) )  ! protects PMAX[1]
        RESID_MAX = PMAX[1]

!       update the result, note that above SYNC_ALL() guarantees
!       that the old ANS(1:NN,1:MM) is no longer needed for halo
!       update.
        ANS(1:NN, 1:MM) = WRK(1:NN, 1:MM)
      UNTIL (RESID_MAX <= TOL)
```

Local Language Summary

The local views promoted by Split-C and CAF represent both a blessing and a curse. To their credit, they give programmers the means to control the details of an algorithm's parallel implementation, relying on the compiler only to insert and implement the actual communication. Use of certain language primitives like split-phase assignment and co-array dimensions make it reasonably clear where communication is required by the program (though it should be noted that traditional assignments using global pointers in Split-C may hide communication). Combining these cues with the explicit user-specified distributions of data and computation, programmers have a reasonably good performance model for evaluating their codes.

On the other hand, local-view languages have the disadvantage that the programmer must manage all of these details by hand, which can become tedious even for trivial codes like Jacobi. This is especially true for cases in which a distributed array's size does not divide evenly amongst the processors. As another example, consider the amount of work that would be required to rewrite the CAF reduction in Listing 3.7 to use a general $\Theta(\log p)$ reduction scheme for arbitrary values of p. These are the sorts of details that global-view languages like ZPL strive to manage on the user's behalf. Moreover, ZPL's performance model does so without letting programmers lose sight of the lower-level details of their program's parallel implementation.

3.11.3 Global-View Languages

High Performance Fortran

High Performance Fortran (HPF) [Hig94] is another extension to Fortran 90 that was developed by the High Performance Fortran Forum, a coalition of academic and industrial experts. HPF is based on the earlier parallel Fortran dialects Fortran-D and Vienna Fortran [FHK+90, ZBC+92]. These languages support parallel computation through the use of programmer-inserted compiler directives. HPF's directives allow users to give hints

Listing 3.8: HPF Jacobi Implementation Excerpt [Joi01]

```
      PROGRAM jacobi1
!
! Solve the Poisson equation with the Jacobi method
!
      PARAMETER (nx = 100, ny=100)
      PARAMETER (tol = 1.0e-8)
      REAL u(0:nx,0:ny), unew(0:nx,0:ny)
      REAL dx, dy, error, tol
      INTEGER i, j
!HPF$ DISTRIBUTE u(BLOCK,*)

      ! initialize all data
      ...

      ! Jacobi iteration
      error = tol + 1.0
      DO WHILE (error > tol)
        FORALL ( i=1:nx-1, j=1:ny-1 )
          unew(i,j) = (u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)) / 4
        END FORALL
        error = MAXVAL( ABS(unew-u) )
        u = unew
      END DO
      STOP
      END
```

for array distribution and alignment, loop scheduling, and other details relevant to parallel computation. The hope is that with a minimal amount of effort, programmers can modify existing Fortran codes by inserting directives that will enable HPF compilers to generate an efficient parallel implementation of the program. HPF supports a global view of computation, managing parallel implementation details such as array distribution and interprocessor communication in a manner that is invisible to the user without the use of compiler feedback or analysis tools. HPF also adds a forall loop construct as in F95 and FIDIL.

The primary disadvantage of HPF is that its specification makes no guarantees as to how compilers will interpret and implement its directives. In particular, no guidelines are provided for programs with conflicting or underspecified directives, and the compiler may choose to ignore a program's directives altogether. The result is that the programmer has little basis for evaluating a code's parallel implementation. Moreover, since each compiler may implement its own interpretation of the directives, programmers may have to re-tune their programs from one compiler or architecture to the next [NSC97, Ngo97].

Listing 3.8 shows a Jacobi iteration kernel written in HPF. Line 10 contains the only HPF directive, specifying that the first dimension of `u` should be distributed in a blocked manner, and that the second dimension should not be distributed. This implementation does not specify `unew`'s distribution, relying on the compiler to align it with `u` for optimal performance. A more cautious programmer would add a second directive specifying its alignment explicitly: `ALIGN unew(:,:) WITH u(:,:)`. Note that as a global-view language, HPF's references to `nx` and `ny` describe the global size of the problem rather than a processor's local block. Finally, note that the reduction is implemented using HPF's intrinsic `MAXALL()` function.

OpenMP

OpenMP is not technically a language, but rather a set of compiler directives and library routines that can be used in C or Fortran programs to specify their parallel execution on shared memory platforms [Ope01]. Nevertheless, OpenMP's directives resemble those of

Listing 3.9: OpenMP Jacobi Implementation Excerpt [KAI01]

```
      do while (error.gt. tol)

         error = 0.0

!* Copy new solution into old
!$omp paralleldo schedule(static)
!$omp& shared(n,m,uold,u)
!$omp&private(i,j)
         do j=1,m
            do i=1,n
               uold(i,j) = u(i,j)
            enddo
         enddo
!$omp end paralleldo

!* Compute stencil, residual, & update

!$omp paralleldo schedule(static)
!$omp& shared(omega,error,tol,n,m,ax,ay,b,alpha,uold,u,f)
!$omp&private(i,j,resid)
!$omp&reduction(+:error)
         do j = 2,m-1
            do i = 2,n-1
!*     Evaluate residual
               resid = (ax*(uold(i-1,j) + uold(i+1,j))
     &                 + ay*(uold(i,j-1) + uold(i,j+1))
     &                  + b * uold(i,j) - f(i,j))/b
!* Update solution
               u(i,j) = uold(i,j) - omega * resid
!* Accumulate residual error
               error = error + resid*resid
            enddo
         enddo
!$omp end paralleldo

!* Error check

         error = sqrt(error)/dble(n*m)

      enddo                          ! End iteration loop
```

a global-view language like HPF closely enough that this section is most appropriate for its discussion.

OpenMP's model is to spawn threads for structured blocks of a program to implement them in parallel. The hope is that this can be done incrementally to convert a sequential program into a parallel one. As in most languages described here, parallelism is achieved by assigning a loop's iterations to different threads. OpenMP's directives give cues to the compiler, such as how loops should be parallelized and whether the code contains operations such as reductions. As in most shared memory programming, explicit locks and synchronization are required to prevent race conditions and data conflicts, and OpenMP's library routines support such mechanisms.

OpenMP has rapidly gained support by the industry, primarily due to the large number of symmetric multiprocessors being produced by traditional sequential hardware vendors. At this point, the main question regarding its success is whether its directives are rich enough and well-defined enough to make OpenMP programs readable and portable. In addition, the fact that OpenMP is not intended for distributed memory platforms will restrict its portability, especially with the community's recent enthusiasm for commodity distributed memory clusters. Even in its target domain of shared memory machines, OpenMP makes data locality seductively invisible. This supports the myth that shared memory systems make all memory equally available to all processors. In reality, such a model disguises the fact that data locality is as crucial to performance on shared memory architectures as it is on distributed memory machines.

Listing 3.9 shows the main loop of an OpenMP Jacobi implementation. OpenMP directives are applied in lines 6–8 and 18–20 to specify the parallel execution of the do loops. In addition, these directives indicate that the arrays and problem sizes should be shared amongst the threads while the loop iterators and local reduction value `resid` should be private for each thread. Line 21 specifies that the sum and assignment into `error` on line 29 should be implemented as a reduction across threads. The rest of the code describes the Jacobi iteration using a global view.

Unified Parallel C

Unified Parallel C (UPC) [CDC+99] is the latest in the evolution of C-based parallel programming languages. In many ways it resembles Split-C or CAF, except that it supports a global view of computation rather than a local view. This distinction is subtle, but can best be characterized by the fact that UPC codes tend to declare variables using their global size rather than a local size crossed with a number of processors. Furthermore, UPC codes are typically written without referring to a processor's unique index.

UPC provides a *distributed shared memory programming model* that gives each thread a private local memory, but also supports a logically shared portion of the address space that any thread can access. Thus, pointer variables may be private or shared, and may point to memory that is either private to the thread, or shared between all of them. Parallelism is typically expressed in UPC using a forall-style loop that not only specifies the loop's bounds, but also the assignment of loop iterations to processors (*affinity*). This mechanism allows the expression of parallel computations over shared arrays using blocked, cyclic, or other traditional distributions. Due to its distributed shared memory model, UPC programmers must specify synchronization explicitly using barrier, split-phase barrier, and locking primitives provided by the language. The current UPC specification has limited support for operations like reductions.

UPC is the youngest of the languages considered in this section, and as such, it has not yet had much of an opportunity to prove itself. The biggest hurdle it seems to face is the fact that moving data from the shared portion of the address space to a processor's physical memory occurs rather transparently in the language, disguising the overhead of communication on both shared and distributed memory platforms. This will obfuscate the performance model for the programmer, though careful programmers may be able to use the affinity field of UPC's forall loops to carefully manage their programs' locality.

Single-Assignment C

Single Assignment C (SAC) is a functional variation of ANSI C developed at the University of Kiel [Sch98b, Sch94]. Its extensions to C support multidimensional arrays, APL-like operators for dynamically querying array properties, and functional semantics. SAC also supports a *with-loop* construct that superficially resembles ZPL's regions. Like regions, with loops are used to specify the indices involved in a computation for a statement or group of statements. However, unlike regions, the with loop provides a mechanism for iterating over an index set by generating indices that can be used to index into an array rather than replacing array indexing altogether. In this sense, the with loop resembles the forall construct of languages like F95 and FIDIL rather than ZPL's regions. SAC currently runs only on shared memory machines, and issues like array distribution and interprocessor communication are invisible to the programmer.

Listing 3.10 shows a SAC implementation of Jacobi designed to run for `LOOP` iterations. Line 10 shows the use of a with-loop to iterate over `B`'s bounds, applying the five-point stencil using the `modarray` operator to modify `B`. Note that `B` can serve as both the source and destination of the stencil due to SAC's functional semantics. Lines 35–36 show a sum reduction that is used by this program as a checksum.

NESL

NESL is a data-parallel programming language developed at Carnegie Mellon that implements nested parallelism using functional semantics [Ble95, Ble96]. Functional semantics dictate that many functions can be executed in parallel due to the complete lack of aliasing between sibling function calls. NESL supports nested parallelism by allowing these functions to spawn parallel function calls of their own.

NESL also supports the concept of data parallelism using its *sequence* concept—a one-dimensional distributed array that can consist of data items or other sequences. NESL provides a parallel *apply-to-each* construct to operate on a sequence's elements in parallel.

Listing 3.10: SAC Jacobi Implementation Excerpt [SAC01]

```
/*************************************************************
 * description:
 *
 *   This SAC demo program implements 2-dimensional relaxation
 *   on double precision floating point numbers applying a
 *   5-point stencil and fixed boundary conditions.
 *************************************************************/

inline double[] onestep(double[] B) {
  A = with ( . < x < . )
      modarray(B, x, 0.25*(B[x+[1,0]]
                          + B[x-[1,0]]
                          + B[x+[0,1]]
                          + B[x-[0,1]]) );

  return(A);
}

inline double[] relax(double[] A, int steps) {
  for (k=0; k<steps; k++) {
    A = onestep(A);
  }

  return(A);
}

int main () {
  A = with( . <= x <= .)
      genarray([SIZE1, SIZE2], 0.0d);

  A = modarray(A, [0,1], 500.0d);

  A = relax( A, LOOP);

  z = with( 0*shape(A) <= x < shape(A))
      fold(+, A[x]);

  printf("%.10g\n", z);

  return(0);
}
```

Listing 3.11: NESL Jacobi Implementation Excerpt [NES01]

```
function sparse_MxV(mat,vect) =
let l = {#row: row in mat};
    i,v = unzip(flatten(mat))
in {sum(row): row in partition({x*v: x in vect->i; v},l)} $

% Each Jabobi iteration is just a matrix vector product,
  this will just repeat it n times. %
function Jacobi_Iterate(Mat,vect,n) =
  if (n == 0) then vect
  else Jacobi_Iterate(Mat,sparse_MxV(Mat,vect),n-1) $

% THE FOLLOWING TWO FUNCTIONS ARE MESH SPECIFIC AND ONLY GET
  CALLED ONCE TO INITIALIZE THE MESH AND VECTOR %

function make_2d_n_by_n_mesh(n) =
  let
    % A sequence of indices of the internal cells. %
    intrnl_ids = flatten({{i+n*j: j in [1:n-1]}: i in [1:n-1]});

    % Creates a matrix row for each internal cell.
      Each row points left, right, up and down with weight .25 %
    internal = {(i,[((i+1), .25), ((i-1), .25),
                    ((i+n), .25), ((i-n), .25)]):
                 i in intrnl_ids};

    % Creates a default matrix row (used for boundaries).
      Each points to itself with weight 1 %
    default = {[(i,1.0)]: i in [0:n^2]}

  % Insert internal cells into defaults %
  in default <- internal $

% Assumes mesh is layed out in row major order %
function make_initial_vector(n) =
  dist(0.0,n^2) <- { i, 50.0: i in [n^2-n:n^2]} $

% ENTRY POINT %
function runit(n, steps) =
let matrix = make_2d_n_by_n_mesh(n);
    vector = make_initial_vector(n);
in Jacobi_Iterate(matrix,vector,steps) $
```

The language's combination of nested parallelism and data parallelism allows the expression of parallel divide-and-conquer algorithms such as Quicksort, as well as traditional data parallel algorithms such as matrix additions and multiplications.

NESL has a well-defined performance model that uses a work/depth scheme to calculate asymptotic bounds for the execution time of NESL programs on parallel computers. Although this model is well-suited to the language's functional paradigm and allows users to make coarse-grained algorithmic decisions, it reveals very little about the lower-level impact of one's implementation choices and how they will be mapped to the target machine. In particular, issues of data locality and interprocessor communication are completely invisible in NESL's syntax and performance model, and are therefore inscrutable to the user.

Listing 3.11 gives a NESL implementation of the Jacobi iteration that runs for `steps` iterations. This version implements Jacobi by multiplying the data array (stored in the n^2-element sequence `vect`) by an extremely sparse matrix `Mat` that describes the 5-point stencil. This matrix is set up in the `make_2d_n_by_n_mesh()` routine in lines 15–31. The actual computation is expressed recursively in lines 1–10.

3.11.4 Global-View Libraries

Mathematical Libraries

As described in the introduction, the vast majority of global-view libraries export interfaces for high-level mathematical operations such as matrix multiplications or matrix solvers. While such routines tend to be highly optimized, they strive to serve a more restricted purpose than the other work described in this section. As a result, these libraries should not be viewed as competitors with languages so much as resources that can be used within a parallel language. The primary challenge to doing so is that each language and library tends to have its own array format and distribution scheme. As a result, trying to get them to interoperate often requires remapping an array from one allocation to another, which can be quite expensive.

KeLP

KeLP [BCSvS01, BFS01, FKB98] is a C++ library that evolved from LPARX [Koh95] and more distantly from FIDIL. It supports rectangular index set classes called *regions* that are used for iteration and to declare arrays. Rather than distributing a single region between processors as in ZPL, KeLP breaks a problem's global index set down into a collection of regions, each of which is assigned to a specific processor to achieve parallelism. Groups of cooperating regions are managed and distributed across physical processors using the *FloorPlan* class. FloorPlans are also used to declare parallel array instance variables, known as *XArrays*. Data transfers between XArrays are captured using *MotionPlan* objects that describes the indices that need to be communicated between processors for a particular FloorPlan, while *Mover* objects implement the specific schedule that a MotionPlan describes for an XArray. KeLP's regions support high-level operators such as shift, intersect, and grow. KeLP expresses iteration over a region's indices using an *indexIterator* class that provides forall-style semantics.

KeLP is unique among the languages described in this section due to the fact that it takes a very practical stance for solving extremely hard problems utilizing multiple arrays at multiple scales. While other languages are trying hard to get a simple Jacobi iteration or matrix multiplication to run well, KeLP is on the front lines working on a crucial class of problems that most languages (including ZPL) hope to be able to handle "someday." This practicality is not without a certain amount of unwieldiness, however, as KeLP's Jacobi implementation demonstrates. Furthermore, KeLP is targeted heavily at grid-based computing, and therefore may not be as general as other languages described here.

The KeLP implementation of Jacobi is shown in Listings 3.12 and 3.13. Listing 3.13 contains the code which declares the main Region (line 19) and then partitions it into subregions, capturing them in a FloorPlan (line 20). It then declares two arrays, `grid1` and `grid2` using that FloorPlan (line 22). Lines 2–15 set up the MotionPlan required to transfer boundary values between processors for the FloorPlan. Line 26 establishes pointers to

Listing 3.12: KeLP Jacobi Implementation Excerpt [KeL01]

```
/* perform one 5-point jacobi stencil operation on oldgrid,      *
 * storing the result in newgrid                                  */
void ComputeLocal(XArray2<Grid2<double> >& oldgrid,
                  XArray2<Grid2<double> >& newgrid) {
   int i;

   for (nodeIterator ni(oldgrid); ni; ++ni) {
      i = ni();

      Grid2<double>& OG = oldgrid(i);
      Grid2<double>& NG = newgrid(i);

      Region2 interior = grow(OG.region(),-1);

      FortranRegion2 Foldgrid(OG.region());

      f_j5relax(OG.data(), FORTRAN_REGION2(Foldgrid),
             NG.data());
   }
}

/* compute the error norm max(abs(newgrid - oldgrid))            *
 * Note: for better performance this should be done in Fortran  */
double computeNorm(XArray2<Grid2<double> >& oldgrid,
                   XArray2<Grid2<double> >& newgrid) {
  double result = 0.0;
  int i;
  Point2 p;

  for (nodeIterator ni(oldgrid); ni; ++ni) {
    int i = ni();
    const Region2 interior = grow(oldgrid(i).region(), -1);
    for (indexIterator2 ii(interior); ii; ++ii) {
      p = ii();
      result = MAX(ABS(newgrid(i)(p)-oldgrid(i)(p)),result);
    }
  }

  mpReduceMax(&result,1);
  return(result);
}
```

Listing 3.13: KeLP Jacobi Implementation Excerpt (continued) [KeL01]

```
/* fill in a one-cell ghost region for each Grid in X          */
void initMotionPlan(FloorPlan2& X, MotionPlan2 &M) {
  int i;
  int j;

  for (indexIterator1 ii(X); ii; ++ii) {
    i = ii(0);
    Region2 inside = grow(X(i), -1);

    for (indexIterator1 jj(X); jj; ++jj) {
      j = jj(0);
      if (i != j) M.CopyOnIntersection(X,i,X,j,inside);
    }
  }
}

void main() {
  MotionPlan2 M;
  Region2 domain(1,1,N,N);
  FloorPlan2 T = UniformPartition(domain);

  XArray2<Grid2<double> > grid1(T), grid2(T);
...
  initMotionPlan(T,M);

  XArray2<Grid2<double> > *oldgrid = &grid1, *newgrid = &grid2;

  Mover2<Grid2<double>, double>* pDM1 =
            new  Mover2<Grid2<double>, double>(grid1,grid1,M);
  Mover2<Grid2<double>, double>* pDM2 =
            new  Mover2<Grid2<double>, double>(grid2,grid2,M);

  do {
    /* Exchange boundary data with neighboring processors */
    pDM1->execute();
    SwapPointer(&pDM1, &pDM2);

    /* Perform the local jacobi computation */
    ComputeLocal(*oldgrid,*newgrid);

    /* Compute the stopping criterion */
    stop = computeNorm(*oldgrid, *newgrid);

    /* Swap the pointers to the grids */
    SwapPointer(&oldgrid,&newgrid);
  } while (stop > epsilon);
}
```

each grid to support quick swapping on each iteration, while lines 28–31 establish a Mover for each grid. The main loop takes place in lines 33–46. The Mover is executed on line 35, causing boundary values to be updated for the current grid. The five point stencil is expressed using a simple Fortran routine for efficiency (not shown here), and is applied to the arrays using the `ComputeLocal()` function of Listing 3.12. Then the normal value is computed, the pointers are swapped, and the next iteration begins.

3.11.5 SIMD Programming Languages

Parallaxis-III and C∗

Though the SIMD (Single Instruction, Multiple Data) programming model seems to be in a state of indefinite retirement, two SIMD programming languages deserve mention here for their index set concepts: Parallaxis-III and C∗ [Brä95, Thi91]. Both languages support dense multidimensional index spaces that are used to declare parallel arrays. Parallaxis-III array statements are performed over the entire array, and therefore do not use index sets to describe computation. In contrast, C∗ does use its index sets (*shapes*) to designate parallel computation over entire arrays. However, it enforces a tight correspondence between the shapes of the computation and the arrays being used. Due to this restriction, its shapes are more of a type modifier than a general index set for expressing array computation. Both languages allow for individual elements to be masked. Neither provides support for strided index sets.

3.11.6 Summary

Table 3.2 provides an overview of the main approaches considered in this section. Its columns indicate: whether each approach is a library- or language-based approach; whether it supports a local-view or global-view of computation; whether it assumes a distributed memory (DM), shared address space (SAS), or shared memory (SM) memory model; its supported notation for array accesses; how its syntax indicates concurrency and communi-

Table 3.2: Summary of Main Programming Approaches

Name	Type of Approach	Programmer View	Memory Model	Array Access Style	Concurrency Indicator	Communication Indicator	Synchro-nization
CAF	language	local	SAS/SM	F90	explicit	co-array reference	explicit
HPF	language	global	DM/SM	F90+forall	directives	none	implicit
KeLP	library	both	DM/SM	forall+indexing	forall	motion plans	implicit
MPI	library	local	DM/SM	base language	explicit	explicit	implicit
NESL	language	global	DM/SM	forall	forall; work/depth	none	implicit
OpenMP	directives	local	SM	base language	directives	none	explicit
PVM	library	local	DM/SM	base language	explicit	explicit	implicit
SAC	language	global	SM	forall+indexing	forall	none	implicit
Split-C	language	local	DM/SM	C	explicit	novel assignments/none	explicit
SHMEM	library	local	SAS/SM	base language	explicit	explicit	explicit
UPC	language	global	DM/SM	forall+indexing	forall	none	explicit
ZPL	language	global	DM/SM	regions+array ops	region scope	array operators	implicit

cation; and whether synchronization is implicit in the approach or explicitly specified by the programmer.

Generally characterizing the space of related work, the trend seems to be that local-view libraries and languages support good flexibility and a performance model for programmers, but require a greater programming effort to explicitly manage the details of parallel programming. By contrast, global-view languages relieve programmers of much of this burden, but often hide it so well that users cannot make informed decisions about their programs. One such example is the lack of cues to indicate communication and data distribution information in languages like HPF and NESL. In ZPL, the goal is that the distribution of regions and their use with array operators will provide the programmer with sufficient information to understand the program's parallel implementation without explicit compiler feedback or performance analysis tools.

3.12 Discussion

3.12.1 Current Support for Region Distribution

The current implementation of ZPL's regions matches this chapter's content very closely with two important limitations. First, a region's dimensions can currently only be distributed in a blocked manner. Second, all of a program's regions are considered interacting. This second limitation implies that there will only be a single processor grid per program.

These limitations exist primarily due to insufficient motivation rather than technical challenges. In particular, most of the applications that have been written in ZPL to date only use a single index space to describe their algorithms. Furthermore, the algorithms' characteristics have always been amenable to block distribution. As a result, the mechanisms for specifying alternative distribution schemes or additional processor grids have never been developed. This is not to say that support for other distributions or multiple problem sizes is not important in a language like ZPL, but simply that the problems which require such mechanisms have not yet been studied in ZPL.