8 Data Ensembles in Orca C

C. Lin and L. Snyder

University of Washington at Seattle

Abstract
This paper describes Orca C, the first language built on the Phase Abstractions programming model. The focus is on the syntax and semantics of the *data ensembles*, a parallel data structuring facility to support machine-independent parallel programs. These features are compared with the data decomposition capabilities of other parallel languages, including Fortran D, Vienna Fortran, and Kali.

1 Introduction

Orca C is the first programming language based on the Phase Abstractions parallel programming model. The importance this model has already been demonstrated: Experiments have shown that it leads to programs that are both efficient and portable. (Scalability is another goal, but this has yet to be demonstrated.) For example, for P processors, the Phase Abstractions version of Livermore's SIMPLE fluid dynamics code has achieved at least P/2 speedup on all types of MIMD parallel machines, including the Sequent Symmetry, the BBN Butterfly GP1000, the Intel iPSC/2 and nCUBE/7 hypercubes, and a mesh machine built from the Transputer T800's [10]; similar results have been reported for the modified Gram-Schmidt method of QR factorization [11]. Though the results do not *prove* the claims of efficiency and portability, they strongly suggest that a language based on the Phase Abstractions might have these properties. Orca C is such a language.

"Phase Abstractions" is a collective term referring to two types of programming concepts for MIMD parallel computation: the *XYZ programming levels* and *ensembles* [14]. The XYZ programming levels structure a parallel program into its constituent parts, including processes, parallel algorithms (phases), data parallel operations and high level control. Ensembles, used to define parallel algorithms (phases), enable the programmer to think of a

computation globally while having local control over the details of general MIMD processes. Three kinds of ensembles are required to define a parallel application: data ensembles define global data structures, code ensembles define the process structure of the computation, and port ensembles specify the communication induced by the data dependencies. Though all three ensemble types are essential for defining parallel algorithms, data ensembles are first among equals, both when designing programs and when understanding Phase Abstractions.

In their most general formulation data ensembles are partitioned data structures [8]. In Orca C, a simple Phase Abstraction language designed for scientific computations, data ensembles are limited to being partitioned arrays. Even so, Orca C's data ensembles provide greater flexibility than the array partitioning facilities of recent parallel language proposals, e.g., Fortran D. The goal of this paper is to describe the Orca C formulation of data ensembles, present solutions to certain implementation issues, and compare it with related work.

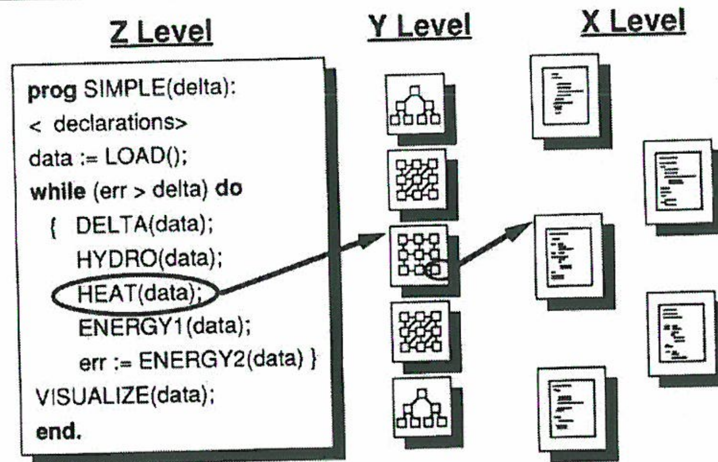

Figure 1: The XYZ Illustration of SIMPLE.

2 Introduction to the Ensemble Abstractions

To describe the role of ensembles in specifying a parallel computation, it is necessary to summarize the details of the XYZ programming levels [13]: A parallel program is viewed as having three kinds of code:

process level (X): composition of instructions to define a lightweight thread,
phase level (Y): composition of processes into a scalable parallel algorithm,
problem level (Z): composition of phases to solve a user's problem.

Phase=Data Ensembles+Code Ensemble+Port Ensemble

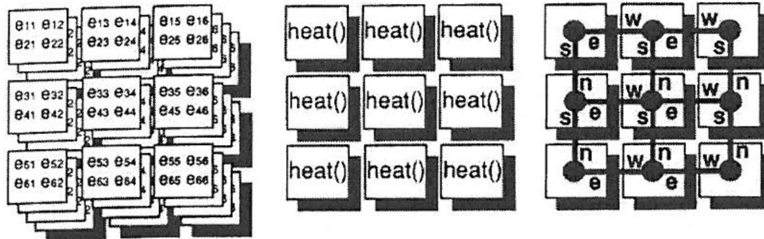

Figure 2: The Combination of Ensembles to Form a Phase.

Viewed in the Z-Y-X order, this formulation gives a top down view of a parallel computation. The Z code gives the high level logic in terms of the invocation of parallel algorithms; the Y code specifies the parallel algorithms as the concurrent execution of a collection of process instances; the X level defines the processes needed to implement the algorithms.

Figure 1 gives a schematic diagram of the XYZ levels of the SIMPLE code [10] illustrating features typical of these levels. The Z level code is typically a short sequence of code; in this case it is an iterative algorithm composed of five phases. The phases, depicted as graphs, are defined by ensembles as explained below; each graph describes the characteristic communication structure of a phase. The processes, illustrated as segments of sequential code, form the nodes of the graph. Besides providing a convenient way to structure a large computation, the XYZ levels also describe the computation so that it can be easily compiled into efficient code. The key components are the ensembles that specify the Y level phases.

A phase is defined by the composition of three types of ensembles: data ensembles, code ensembles and port ensembles [2, 8]. An ensemble is a set with a partitioning. In particular, a data ensemble partitions the items of a data structure, a code ensemble partitions a set of (X level) process instances, and a port ensemble partitions a graph. In order to compose ensembles of the three types into a phase, it is necessary that the ensembles be *conformable*, i.e., the partitionings have the same cardinality (and in Orca C, the same dimensions). Since the partitionings conform, a particular partition, called a *section*, in one ensemble can be associated with a corresponding section in another ensemble. This allows us to define informally the high level semantics of a phase: The process in the ith section of a code ensemble executes on the data of the ith section of the data ensembles and communicates with the neighbors of the vertices of the ith section of the port ensembles; the initial state of a phase is the content of the data ensembles prior to invocation of the phase, and the result of the phase is the content of the data ensembles upon completion of the processes.

Figure 2 shows a schematic diagram of the `heat()` phase of the SIMPLE computation. For this iterative approximation there are several data ensembles that correspond to discretized quantities, e.g., energy at each point, temperature at each point, etc. The data

ensembles are partitioned into blocks. That is, each block contains a set of contiguous data points. The code ensemble specifies that each section will execute the same process, `xHeat()`, so in this case we have an SPMD computation. Finally, the port ensemble specifies that the communication of the computation will be to nearest neighbors named N,E,W,S.

A key concept of ensembles is that their size is parameterized and their partitioning is parameterized. The parameterization specifies how the number of sections is to grow or shrink in response to changes in problem size, number of processors, preference for many or few multiplexed threads per processor, etc. Thus, *the number of sections in a phase's ensembles specifies the logical concurrency of the computation.* This control is essential in making the computation malleable enough to run on many different machine types, i.e., to be portable.

3 Orca C Overview

Orca C is a simple language implementation of Phase Abstractions. Facilities are provided for programming at the X, Y and Z levels. Additionally, facilities are provided for defining data, code and port ensembles. An Orca C program has the following structure:

```
program <name> (<parameter list>);
    configuration and constraint computations;
(<configuration parameter list>):
    data ensemble definitions;           |
    code ensemble definitions;           |  Y level
    port ensemble definitions;           |
    phase definitions;                   |
    process definitions;                    X level
begin
    program body                            Z level
end.
```

The items of the five declaration sections prior to the **begin** can be presented in any order.

The parameter list specifies arguments to the computation including computation-specific information such as the convergence limit, and (possibly implicit) environment characteristics such as the number of processors on the host machine. This data, plus other information that might be read in from external media, such as the size of data sets, are input to configuration computations that the programmer defines. These computations determine how the structures of the program are to be configured to respond to the execution environment. Typical of a configuration computation is the determination of the number of sections the computation should have, i.e., how much logical concurrency is appropriate for the prevailing conditions. Once computed, the parameters are explicitly listed in the configuration parameter list, followed by the definition of the parallel program.

The focus of this paper is the data ensemble specification portion of the program and its semantics.

4 The Syntax of Data Ensembles

In its simplest form, Orca C uses a syntax for specifying data ensembles that leads to the following kinds of specifications.

```
ensemble  square[p][q]      float a[m][n];
```

This Y level declaration begins with the keyword `ensemble`, declares an ensemble named `square`, and defines a partitioning where data is decomposed into p rows and q columns of data blocks. Following the partition definition is a standard C language variable declaration, in this case a two dimensional array of `float`'s named `a`. This declaration specifies the data from a *global* view. That is, the overall size of the `a` array is $m \times n$, which is to be partitioned into an array of $p \times q$ sections. Together, these two sets of parameters implicitly define the size of each local section. If the number of sections do not evenly divide the array dimensions, i.e., $m = (p-1) \cdot s + r$, then $(p-1)$ sections will have s elements and the last section will have r elements. The values of p, q, m, and n can be runtime parameters computed in the "configuration and constraint computations" section of the program.

The general rule for decomposing Orca C arrays is given below, where $d(Part)$ is the number of dimensions of the partitioning and $d(Array)$ is the number of dimensions of the array. These rules for Orca C were chosen under the assumption that computations will iterate faster over the lower dimensions of an array.

1. If $d(Part) = d(Array)$, the ith dimension of the array is decomposed according to the ith dimension of the partitioning.
2. If $d(Part) < d(Array)$, the array is decomposed in its first $d(Part)$ dimensions. The remaining dimensions of the array are not decomposed.
3. If $d(Part) > d(Array)$, the first $(d(Array) - 1)$ dimensions of the array are decomposed as in case 1, but the last dimension of the array is decomposed into a number of sections that is the product of the remaining dimensions of the partitioning.

The following examples illustrate these partitioning rules. The `c` array adheres to rule 2 and is not partitioned across its third dimension, while the `d` array obeys rule 3 and is partitioned into $p \times q$ sections. The last example shows that multiple ensembles can be defined with a single statement, and that once a partitioning has been defined, it can be used again in subsequent declarations.

```
ensemble  square[p][q]      float b[m][n];              /* rule 1 */
ensemble  square[p][q]      float c[m][n][2];           /* rule 2 */
ensemble  square[p][q]      float d[m];                 /* rule 3 */
ensemble  square            float e[m][n], f[m][n];
```

The examples below show how horizontal and vertical strip partitionings can be defined.

```
ensemble  hstrips[P][1]     float g[m][n];              /* horizontal strips */
ensemble  vstrips[1][P]     float h[m][n];              /* vertical strips */
```

In practice, these partitionings could be defined as special cases of the block decomposition by setting the values of p and q appropriately. For example, a horizontal strip decomposition is defined by using the `block` decomposition specified above and by setting p and q as follows (in the configuration and constraint computation section):

$p = P;$
$q = 1;$

Implicit in the above declarations is the choice of a contiguous block partitioning. This is the default in Orca C, but other decompositions are possible. In a full language arbitrary user-specified decompositions can be given, including those computed at run-time or read in from a file. This will allow Orca languages to accommodate computations with irregular structures such as unstructured meshes.

To indicate the type of decomposition, a modifier is placed before the variable declarations as illustrated below. The `contiguous` modifier is the default and specifies a contiguous block partitioning. The `interleaved` decomposition assigns consecutive elements of an array to different processes; this partitioning accepts an argument to specify the granularity of the interleaving, i.e., the number of consecutive elements to assign to each process. For example, the `block3` partitioning specifies an interleaving by blocks of 4, while the `block4` partitioning indicates interleaving by some variable amount that is bound at runtime.

```
ensemble  block[p][q]   contiguous        float a[m][n];
ensemble  block2[p][q]  interleaved       float b[m][n];
ensemble  block3[p][q]  interleaved(4)    float c[m][n];
ensemble  block4[p][q]  interleaved(x)    float d[m][n]; /* x is a config parameter */
```

4.1 Mapping to the Sections

Although Orca C programs consist of three levels of code, the ensembles properly belong to all levels with each level seeing a different view. The Z and Y levels see a global view of the data ensembles, while each X level process sees a local portion of the data. Furthermore, data ensembles in Orca C exist across phases. So logically each phase manipulates some global state (the data ensembles) that persists after the phase completes.

When a phase is invoked at the Z level, the necessary ensembles are passed to the X level through an interface similar to the way parameters are passed to procedures in sequential imperative languages. The semantics are pass by reference. This parameter passing is helpful in defining the interface to each phase and in restricting the scope of the various ensembles. Moreover, from the X process viewpoint, this parameter passing is useful in defining the local dimensions of each data ensemble. The dimensions of each section of a data ensemble are implicitly defined at the Y level, as described in the previous subsection, and these dimensions are then passed to the X level through the X process' declaration of ensembles. This is illustrated in the next subsection.

In Orca C ensembles have a single partitioning for the entire execution of the program. This will be relaxed in future versions of Orca. For example, it may be useful for one

phase to view an ensemble as a 2D array of values, while another views it as a 1D array of values.

4.2 The X Level View

In contrast to the Y level where arrays are viewed globally, the X level sees the ensembles from a local point of view. The interface between the X and Z levels declares the data ensembles required by each phase.

For example, for a phase that is invoked at the Z level as follows

```
    . . .            /* other Z level code */
xHeat(x, rho, J);
    . . .            /* other Z level code */
```

the X level may declare the parameters as below.

```
xHeat(x[0:s-1][0:t-1], rho[0:s-1][0:t-1], J[0:s-1][0:t-1])
{
    . . .            /* body of X level code goes here */
}
```

Several details are worth noting.

- From the X view, only a local portion of each ensemble is visible. So for example, each **xHeat** process sees only an $s \times t$ section of data points for the variable **x**.
- The formal parameters for arrays have both lower and upper array bounds included in their declaration. This is the mechanism through which the ensemble's local dimensions are passed to the X level.
- The size of the different sections of code may differ. For example, one process may have s bound to 100 while another has s bound to 99 because the numbers may not divide evenly. Such differences are transparent to the X level code. The values of s and t are passed in from the Z level and are read-only parameters at the X level. Because the size of the sections are parameters, the X level can accommodate decompositions of any shape, e.g., either strips or squares.
- The X level sees a contiguous block of data regardless of the actual decomposition used. In particular, the indexing of the local arrays does not change with the choice of data decomposition.

5 Fluff

Scientific computations often involve local neighbor communication in which one process requires data from the edge of a neighboring process' array. This common operation is simplified in Orca C through the notion of *fluff*, which is similar to Overlaps [7] and GuardStrips [12]. Fluff is a software cache that can be used to maintain a current copy of data between adjacent processes. In Orca C, fluff is allowed for any dimension of an ensemble array and is simply an extension of the array boundaries. This allows X level code to access fluff in the same manner that actual array values are accessed.

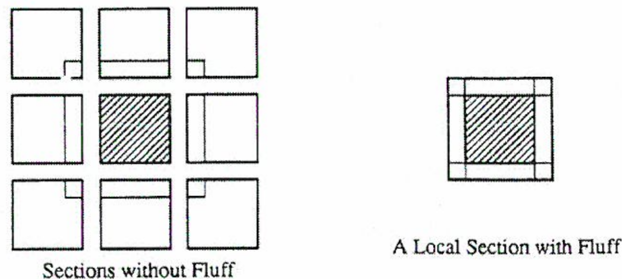

Figure 3: Fluff.

As an example of the use of fluff, consider the Jacobi Iteration, where each process requires the edge values from its four nearest neighbors. The usual technique is to have each process send edge values to neighbors, receive edge values from neighbors, and then compute the average of each data point using the newly received edge values. (As an optimization the computation of interior points can be computed without communication and can be inserted after the sends.) Without fluff, the programmer must declare a larger array and adjust the indexing accordingly. In Orca C, the local X level view of the data is an array large enough to include the fluff. From the global view, the distributed array does not logically include fluff.

Fluff at the Y Level. Fluff is declared at both the Y and X levels to allow for compile-time error checking. At the Y level, Orca C restricts fluff to be symmetric in all dimensions. Future versions of Orca will relax this restriction to allow arbitrary amounts of fluff. The Y level declaration below states that the `square` partitioning extends each dimension of the data ensemble by 1 in each direction. This declaration asserts that maximum allowable fluff any X level process can access. Note that fluff is a characteristic of a partitioning, so all ensembles using the `square` partitioning, such as the `b` array, have fluff similar to the `a` ensemble.

```
ensemble square[p][q] fluff(1)    float a[m][n];
ensemble square                   float b[m][n];
```

Fluff at the X Level. Fluff is also specified at the X level through the declaration of array bounds of ensemble arrays. Here, fluff need not be symmetric. For example, the following process header shows that the x array has fluff of 1 around all four borders because the local array bounds go from -1 to s and -1 to t rather than 0 to $(s-1)$ and 0 to $(t-1)$. By contrast, the rho array only has fluff along two sides. Note that in addition to specifying the size of fluff, these declarations indicate the alignment of the fluff within the local array.

```
xHeat(x[0:s-1][0:t-1], rho[0:s-1][0:t-1], J[0:s-1][0:t-1])
     float      x[-1:s][-1:t];
     float      rho[0:s][0:t];
{
     . . .            /* body of X level code goes here */
}
```

Exchange. The Exchange statement uses fluff and the data ensembles to provide high level communication that is less error prone than programmer-specified message sends and receives. The Exchange statement causes each process to update its neighboring process' fluff.

The syntax for Exchange is as follows, where <*ensemble variable*> is any data ensemble. Note that Exchange may be specified at both the X and Z levels but is conceptually a Z level operation.

exchange <*ensemble variable*>;

The semantics are that messages implementing the exchange are initiated no later than the Exchange statement. Each process may continue executing asynchronously until the process' first access to exchanged data, at which point the process must block until arrival. This allows processes to hide communication latency as much as possible. Of course, when it is safe to do so, the compiler may optimize the Exchange statement by sending the messages earlier than specified.

6 Comparison with Related Work

We now compare Orca C with several related languages: Kali [9], Fortran D [6], Vienna Fortran [4], and MetaMP [12]. We consider these state-of-the-art languages because they focus on the data decomposition as the expression of parallelism, so it is instructive to make a comparison with Orca C's data ensembles. These languages are also of interest because they represent an approach that is fundamentally different from Orca's. In Orca, parallel programs are explicitly created from a parallel algorithm. In these other languages, parallel programs are obtained through the transformation of a sequential program, though they differ in the amount of information supplied by the programmer. While the latter

approach has clean semantics because the languages can guarantee deterministic – that is, sequential – semantics, the former approach has much greater potential for parallelism.

Data Decomposition. Arrays are commonly used in scientific computations. All of these languages, including Orca C, only allow arrays to be distributed. But the ensembles are more general, and future versions of Orca will relax this constraint to general data structures.

Arrays in Fortran D are aligned with respect to some abstract partitioning using the `ALIGN` statement. A separate `DISTRIBUTE` statement is used to map partitioned data to physical processors. Because the `ALIGN` and `DISTRIBUTE` statements can be thought of as executable statements as well as declarations, Fortran D supports remapping of arrays and allows distributed arrays to be passed as parameters to any procedure. Each procedure may define its own data distribution, which may cause an implicit redistribution of data upon procedure call. Together these features can be potentially very expensive.

Vienna Fortran supports the same data distributions that Orca C does. These are specified by passing the keywords `BLOCK` or `CYCLIC` as parameters to the `DIST` statement. (`CYCLIC` distributions are analogous to Orca C's `interleaved` distributions.) In Vienna Fortran a distinction is made between dynamic and static arrays. Only dynamic arrays can be dynamically redistributed. Arrays may be aligned with respect to each other.

Kali provides `BLOCK`, `CYCLIC`, and `BLOCK_CYCLIC` distributions, as well as irregular distributions. The programmer controls the assignment of loop iterations to processors through the use of the `On clause`. There are plans for Kali to support dynamic data distribution in the future.

Orca C does not currently support dynamic redistribution of arrays.

Parallel Loops. Except for Orca C, the languages discussed here depart from sequential languages primarily in their support for data decomposition, although some of these languages do provide mechanisms for specifying parallel loops.

Vienna Fortran provides no form of parallel loops. Fortran D has the `FORALL` statement that can be used to specify loops with no loop carried dependencies. To ensure deterministic semantics of updates to common variables by different loop iterations, values are deterministically merged at the end of the loop. This construct is optional in that the compiler will attempt to extract parallelism even where `FORALL` is not used. In contrast to Fortran D's optional `FORALL` loops, Kali requires `FORALL` loops with the same restriction that each loop can execute independently. Similarly, MetaMP's `splitFor` directive is required to identify parallel loops.

One limitation of all of these mechanisms for parallel loops is they either assume loop iterations that can execute independently, or they rely on static data dependency analysis to identify such loops. It is not always possible or natural to write such loops, and it is often difficult to perform data dependency analysis across procedure boundaries. The parallelism of an Orca C program does not rely on the automatic extraction of loop-level parallelism, so Orca C programs can exhibit more parallelism than those that require loops with no loop-carried dependencies.

Explicit Communication. MetaMP provides a set of commonly used communication operations. Directives are provided to rotate data (the `roll` directive) and perform global reduction. Orca C supports fully general communication patterns. Any commonly used communication structures can be captured in ensemble libraries for reuse. None of the other languages provide mechanisms for explicit communication among processes.

Virtual Processors. Fortran D, Vienna Fortran, Kali and MetaMP provide no language support for the notion of virtual processors. In Orca C the number of processes, and hence the granularity of parallelism, is controlled through the ensembles. This ability to have more threads than physical processors can be useful in hiding communication latency, particularly in modern multi-threaded architectures such as the Tera Computer [3] and the MIT Alewife [1]. Of course, sophisticated hardware is not a requirement for exploiting multiple threads [5].

Fluff. MetaMP's *guardStrips* are similar to Orca's fluff. It appears that guardStrips must be symmetric, and guardStrips are explicitly controlled by the `update guard` directive. Since the other languages do not support explicit communication, they have no need for fluff or guardStrips at the language level.

7 Conclusion

Orca C ensembles have clean data decomposition facilities that are comparable to those found in other languages, and more expressive in terms of control of granularity. A key point is that while Orca C provides a global view of data (as do other languages), it also supports the full flexibility of MIMD parallelism. The portability potential of Orca C has been shown. The implementation is well underway.

Acknowledgments. We acknowledge the valuable contributions of the other members of the Orca group - Langdon Beeck, George Forman, Scott Hauck, Ton Ngo and Dave Swan - for their ideas and insights in the design Orca C.

References

[1] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife machine: A large scale distributed-memory multiprocessor. In *The Workshop on Multithreaded Computers at Supercomputing 91*, Nov 1991.

[2] G. Alverson, W. Griswold, D. Notkin, and L. Snyder. A flexible communication abstraction for nonshared memory parallel computing. In *Proceedings of Supercomputing '90*, November 1990.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[4] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran - a Fortran language extension for distributed memory multiprocessors. Technical Report No. 91-72, ICASE, September 1990.

[5] E. W. Felten and D. McNamee. Improving the performance of message-passing applications by multithreading. In *Proceedings of Scalable High Performance Computing Conference, SHPCC-92*, pages 84–89. IEEE Computer Society Press, April 1992.

[6] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. FORTRAN D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[7] M. Gerndt. Updating distributed variables in local computations. *Concurrency- Practice and Experience*, 2(3):171–193, September 1990.

[8] W. Griswold, G. Harrison, D. Notkin, and L. Snyder. Scalable abstractions for parallel programming. In *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990. Charleston, South Carolina.

[9] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[10] C. Lin and L. Snyder. Portable parallel programming: Cross machine comparisons for SIMPLE. In *Fifth SIAM Conference on Parallel Processing*, 1991.

[11] C. Lin and L. Snyder. An algorithm of choice for solving QR factorization. Technical report, Dept. of Computer Science and Engr., University of Washington, February 1992.

[12] S. Otto. MetaMP: A higher level abstraction for message-passing programming. Technical Report CS/E 91-003, Oregon Graduate Institute of Science and Technology, 1991.

[13] L. Snyder. The XYZ abstraction levels of Poker-like languages. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 470–489. MIT Press, 1990.

[14] L. Snyder. Applications of the "Phase Abstractions" for portable and scalable parallel programming. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*, pages 79–102. North Holland, 1992.