# Synthesis of Distributed Arrays in Titanium

Amir Kamil
*Computer Science Division, University of California, Berkeley*
*kamil@cs.berkeley.edu*

May 17, 2006

## 1 Introduction

In parallel programs, data is distributed between all the threads in the program. Many times it is necessary for one thread to access data that is located on a different thread, requiring communication between threads. The cost of communication can be a significant portion of the total running time of a parallel program (e.g. >40% for Titanium adaptive mesh refinement on 3 nodes [13]), and load imbalance can have a significant detrimental effect on performance. As a result, much of the effort involved in writing parallel code is focused on optimizing the data structures and access patterns, and the resulting implementation tends to be far removed from an abstract description of the structure. The end result is a large piece of code that is difficult to understand from the points of view of both correctness and performance. The goal of this project is to allow a programmer to specify data structure details at a high level, reducing the required programming effort while still resulting in good performance.

Ignoring data distribution, most data structures used in parallel programming have a relatively simple abstract layout. For example, the main data structure may be just a multidimensional array. Distributing the structures across multiple threads introduces significant complexity. The following are a few issues that come up:

- Which thread owns what pieces of the data structure?
- Which parts of the data structure should be replicated?
- When and how should consistency of the replicated parts be maintained?

Ideally, these issues would be resolved independently of the algorithm that operates on the distributed data, decoupling the data structure implementation from the algorithm.

In this paper, we present a distributed array synthesizer for the Titanium language. The synthesizer allows specification of arrays with varying answers to the above questions in a simple language, and generates the Titanium code to implement the arrays. The implementation is abstracted from the user, so that the same algorithm can be performed on different types of distributed arrays.

## 2 Background

### 2.1 Titanium Background

The Titanium programming language [14] is a high performance dialect of Java designed for distributed machines. It is a *single program, multiple data* (SPMD) language, so all threads execute the same code image. In addition, Titanium has a global address space abstraction, so that any thread can directly access memory on another thread. However, accessing memory on another thread can be much slower than on the same thread, since network communication may be required.

Titanium provides support for multidimensional arrays. Arrays are specified over $N$-dimensional *rectangular domains*, which consist of a lower bound, an upper bound, and a stride. For example, the domain `[[1,1] : [5,5] : [2,2]]` has a lower bound of `[1,1]`, an upper bound of `[5,5]`, and a stride of `[2,2]`, so it consists of the set of points {`[1,1]`, `[3,3]`, `[5,5]`}.

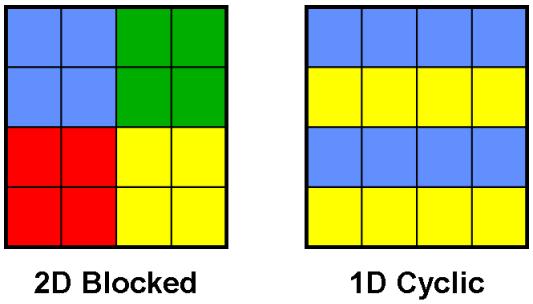


Figure 1: Two possible distributions of a two dimensional grid.

Support for distributed arrays, however, is weak in Titanium. They can be built from the ground up by allocating each thread's portion locally on that thread, and then exchanging pointers to these portions among all threads. Each thread's portion must be manually computed, and a programmer must manually translate an access to the distributed array into an access to the right thread's portion of the array. This process is very tedious and error-prone.

## 2.2 Distributed Arrays

Distributed arrays in general are characterized by multiple factors, including dimension, distribution, replicated portions, and access patterns. Dimension is obvious, but the rest require some explanation.

### 2.2.1 Distribution

Arrays can be distributed among the available threads in many ways, and each dimension can be distributed in different ways. In this paper, we focus on regular distributions, of which the following are most common:

- **blocked:** cells in a dimension are divided into blocks of a given size, and each block is given to a different thread. The left side of Figure 1 shows a distribution that is blocked in two dimensions, with a block size of 2 in each dimension.
- **cyclic:** each thread gets every $n$th cell in a dimension, where $n$ is the total number of threads. The right side of Figure 1 shows a cyclic distribution in one dimension, where there are two threads total.
- **blocked-cyclic:** similar to **blocked**, except that each thread gets every $n$th block in a dimension. In this paper, we do not support this distribution type.

Not all dimensions of an array need be distributed. The right side of Figure 1 shows a distribution in which only one dimension of a two-dimensional array is distributed.

### 2.2.2 Replicated Portions

Some parts of a distributed array may be replicated, for decreased communication, for algorithmic purposes such as in multi-grid [6, 10, 7], or for lowering memory and computational requirements such as in adaptive mess refinement [13].

In many parallel algorithms, the value of an array element depends on the values of its neighbors. At thread boundaries in a distributed array, these neighbors may be on a different thread. In order to decrease communication, *ghost cells* are used to cache these values. Ghost cells must updated whenever the cached value changes, so as to be coherent with the real cells that they correspond to.

In multigrid algorithms, the distributed array consists of multiple *refinement levels*. Each level corresponds to the entire physical domain represented by the array, but at a different resolution. For example, the bottom level of an array may contain a cell for each point in the physical domain, the next level a cell for every four points, and so on. The *refinement ratio* is the ratio of the resolutions between levels, so if the previous example was a single-dimensional array, the ratio would be 4. In these algorithms, a *coarsening* operation is used to translate the values of multiple cells in a higher resolution level to a single cell in a lower resolution level, and an *interpolation* operation is used to do the reverse.

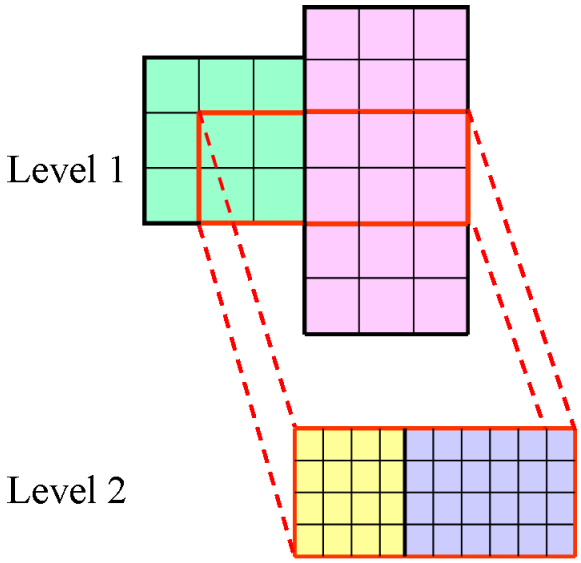


Figure 2: A multilevel irregular grid, a subset of which is replicated at a higher resolution.

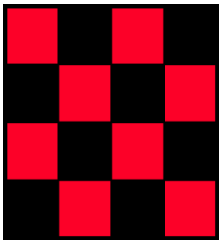

Figure 3: Some algorithms require a division of the grid into alternating red and black cells.

In adaptive mesh refinement, multiple levels are also used. Higher resolution grids require more computation and memory, since they consist of more cells, so higher resolution is only used where needed. Figure 2 shows an example of a two-level grid.

For this paper, we concentrate on providing ghost cells and simple multiple levels such as those used in multigrid. We do not yet provide a mechanism for distributing individual levels differently.

#### 2.2.3 Access Patterns

The order in which array elements are accessed depends on the algorithm being performed. Many algorithms access elements sequentially. Linear algebra algorithms often access elements in a blocked fashion in order to maximize cache use. Other algorithms such as multigrid perform more complicated access patterns, such as red/black order. In this case, every other cell is red and the rest are black, as in Figure 3, and the red cells are first accessed followed by the black cells.

Knowledge of access patterns can be leveraged by an array synthesizer. For example, it may be difficult to manually specify which cells are red, so the synthesizer could automatically generate functions to compute the red cells. The synthesizer could also optimize the layout of an array for a particular access pattern.

In this paper, we allow a user to specify the access pattern in each dimension. We do not use this to modify data layout, but our synthesizer does attempt to generate functions to help the user perform accesses in the specified pattern.

```
array data {
  dimension[2];
  distribution[BLOCKED(length[1] / Ti.numProcs()), NONE];
  access[NONE, BLOCKED(2)];
  boundary[GHOST(1), NONE];
  refine[2, 2, AVERAGE, COPY];
}
```

Figure 4: A sample distributed array specification.

# 3 Language Modifications

The Titanium language contains no support for specifying distributed arrays, so we defined a new language subset for this purpose. In addition, we added a new construct to Titanium for operating over distributed arrays.

## 3.1 Specification Language

A distributed array is specified using syntax similar to class and interface specification in Java. Figure 4 shows a sample array specification. The keyword `array` is used in place of `class` or `interface` to denote a distributed array definition.

Within an array specification, five different commands are allowed:

1. The mandatory `dimension` command specifies the dimension of the distributed array.
2. The mandatory `distribution` command specifies the distribution type in each dimension of the array. The type can be `NONE` for no distribution, `BLOCKED` for a blocked distribution with the block size given as an argument, or `CYCLIC` for a cyclic distribution.
3. The optional `access` command specifies the local access pattern in each dimension. This can be `NONE` for no specified pattern, `SEQUENTIAL` for sequential access, `BLOCKED` for blocked access, and `RED_BLACK` for a red/black pattern.
4. The mandatory `boundary` command specifies the boundary type in each dimension. This can be `GHOST` for ghost cells, with the width provided as an argument, or `NONE` for no ghost cells.
5. The optional `refine` command specifies a multilevel array. The first argument is the number of levels, the second is the refinement ratio between levels, the third is the coarsening operation, and the last is the interpolation operation.

## 3.2 Using Distributed Arrays

### 3.2.1 Declaration and Construction

At the moment, we leverage Titanium's support for templates to specify the type of a distributed array. The Titanium type for a distributed array of `int`s using the specification in Figure 4 would be `data<int>`.

Constructing a distributed array is similar to allocating a normal object, with the argument the domain of the array. For example, the code to create an `int` array using Figure 4 over the domain $D$ would be `new data<int>(D)`.

### 3.2.2 Array Access and Functions

Elements of distributed arrays can be accessed using the normal array operators (`[]` and `[]=`). Any element can be accessed in this way, including those on other threads. The generated code automatically determines which thread owns the element and retrieves or updates it accordingly.

Various additional functions are provided for distributed arrays. The `synchronize()` function updates the values in each thread's ghost cells. Other functions are provided for restricting access to local elements and for accessing the array in the pattern specified in the `access` command.

```
forall (p in grid.domain()) {
    newgrid[p] = (grid[p+[-1,0]] + grid[p+[1,0]] +
                  grid[p+[0,-1]] + grid[p+[0,1]]) / 4;
}
```

Figure 5: An example of a `forall` loop. Each thread performs the computation for the points it owns.

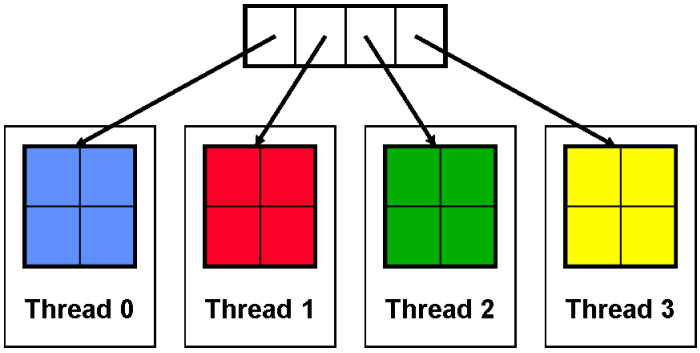


Figure 6: The actual implementation of the blocked distribution in Figure 1.

#### 3.2.3 New Constructs

In order to simplify code that uses a distributed array, we added a new `forall` loop to the Titanium language. Figure 5 shows an example of such a loop. The semantic meaning of the loop is that each thread executes the body for each of the points it owns in the specified domain, in some arbitrary order.

## 4 Implementation

A prototype distributed array synthesizer has been built to implement the language specified in §3. The synthesizer is currently written in Java, and requires that the array specification be given in its own file. It produces a Titanium class corresponding to the specification, which can be compiled with the rest of the application.

Since the synthesizer generates legal Titanium code, no modifications were required to the Titanium compiler to implement distributed arrays. The new `forall` construct, however, requires compiler support.

### 4.1 Implementation Strategy

Distributed arrays are represented by Titanium classes, and the details of the implementation are abstracted from the user's view. The only evidence of the class basis is in declaring and constructing arrays (§3.2.1). Titanium provides operator overloading, so actual access to distributed arrays is identical to normal arrays.

Since Titanium already has support for multidimensional arrays, we elected to build distributed arrays on top of them. Each thread's portion of the distributed array is represented by a normal multidimensional array in that thread's memory space. Each thread then maintains a set of pointers to the other threads' portions of the global array and accesses their elements through these pointers, as shown in Figure 6.

The actual work required in computing each thread's portion of the global array is divided between the synthesizer and the generated code. The synthesizer can determine the proper algorithm based on the distribution types in each dimension, but the number of processors and size of the global domain are not know statically. The actual computation must therefore be performed in the generated code. Other functions, such as global array indexing, require a similar division of work between the synthesizer and the generated code.

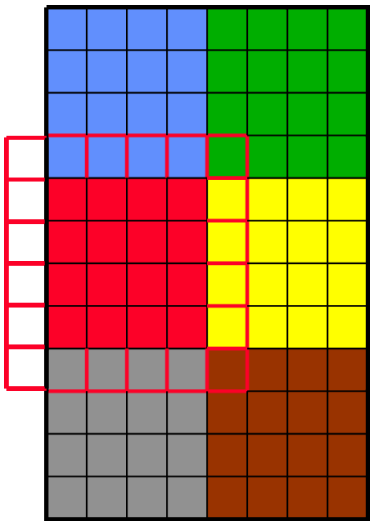

Figure 7: The game of life uses a grid with a blocked distribution in each dimension, and single width ghost cells in each dimension.

### 4.2 Work in Progress

Due to time constraints, we concentrated on implementing the features required by the applications in §5. Some of the remaining features can be easily implemented, while other require more work to overcome the limitations in Titanium.

#### 4.2.1 Multilevel Arrays

Due to time limitations, the implementation of multilevel arrays is only partially complete at the moment. It also only supports refining the entire array instead of subsets of it, which is sufficient for multigrid but not for adaptive mesh refinement.

#### 4.2.2 Titanium Limitations

Normal arrays in Titanium are specified over rectangular domains, which consist of bounds and strides in each dimension. Such domains are insufficient to describe certain patterns, such as the red cells in a red/black ordering or the set of local points to a thread under a distribution that is cyclic in multiple dimensions. Under the current strategy of implementing distributed arrays on top of normal arrays that represent local data, red/black ordering and cyclic distributions cannot be provided, so they are currently unimplemented.

## 5 Applications

In order to test and evaluate distributed arrays, we used two applications. The first is a simulation of the game of life, and the second is a pseudo-polynomial solver for the knapsack problem.

### 5.1 Game of Life

The game of life is a simulation of fish living in the ocean. The ocean is represented by a two dimensional grid, and each grid cell holds at most one fish. The simulation proceeds in time steps, and at each step, the value of a cell is a function of the cell and its eight neighbors at the previous step.

The rules for deciding whether a fish occupies a cell during the next time step are:

- A new fish is born at a grid cell if it is currently empty and exactly three of its eight neighboring cells are nonempty

```
array ocean {
  dimension[2];
  distribution[BLOCKED(length[1] / 3),
               BLOCKED(3 * length[2] / Ti.numProcs()];
  boundary[GHOST(1), GHOST(1)];
}
```

Figure 8: The distributed array used in the game of life.

```
for (int single i = 0; i < steps; i++) {
  forall (p in cells.domain()) {
    int numfish = cells[p+[-1,-1]] + cells[p+[-1,0]] +
                  cells[p+[-1,1]] + cells[p+[0,-1]] +
                  cells[p+[0,1]] + cells[p+[1,-1]] +
                  cells[p+[1,0]] + cells[p+[1,1]];
    if (cells[p] == EMPTY && numfish == 3)
      newcells[p] = FISH;
    else if (cells[p] == FISH && (numfish < 2 || numfish > 3))
      newcells[p] = EMPTY;
    else
      newcells[p] = cells[p];
  }
  swap(cells, newcells);
  cells.synchronize(); // synchronize ghost cells
}
```

Figure 9: The actual code to run the game of life.

- A fish dies of loneliness if it has 0 or 1 neighbors
- A fish dies of overcrowding if it has 4 or more neighbors
- Other cell configurations are stable

The problem is parallelized by dividing the grid among threads. To minimize communication, the distribution is blocked in both dimensions, as shown in Figure 7. Since each cell depends on its neighbors, ghost cells of width 1 are required at processor boundaries.

Using the distributed array generator, the code for the game of life is very simple. The array specification is shown in Figure 8, and the actual code to use the array is given in Figure 9.

## 5.2 Knapsack

Given a set of $n$ books with particular weights and sizes and a sack with a given capacity $m$, the knapsack problem is to compute the maximum-weight subset of books that can fit into the sack. The problem can be solved in pseudo-polynomial time using dynamic programming. A two dimensional matrix $M$, with one dimension corresponding to books and the other to capacities ranging from zero to the sack capacity, is used, where $M(i,j)$ is the maximum weight of a subset of the first $i$ books in a capacity of $j-1$. If the weight of the $i$th book is $w_i$ and the size $s_i$, then this is just the maximum of $M(i-1,j)$ and $M(i-1,j-s_i)+w_i$. The entry $M(n,m+1)$ gives the final weight. The set of books corresponding to this weight is then computed by backtracking through the matrix. The previous entry used to get the value of entry $M(i,j)$ can be determined by comparing $M(i,j)$ with $M(i-1,j)$ and $M(i-1,j-s_i)+w_i$, and thus the set of all entries used in obtaining the final weight can be determined by starting at $M(n,m+1)$ and working backwards.

The problem is parallelized by distributing the dimension corresponding to books but not that corresponding to capacities, as shown in Figure 10. Since the value at each cell is a function of the values in the row above it, ghost cells of width 1 are required at the processor boundaries. Each cell actually only depends on the ones of equal or lesser capacity in the row above,

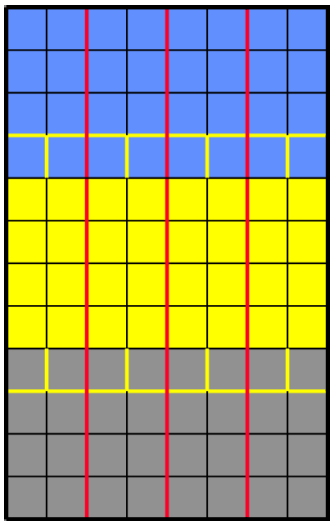

Figure 10: The knapsack algorithm uses a grid with a blocked distribution in one dimension, and single width ghost cells in that dimension. Within a thread, the local cells are accessed in a blocked fashion.

```
array matrix {
  dimension[2];
  distribution[BLOCKED(length[1] / Ti.numProcs()), NONE];
  access[NONE, BLOCKED(100)];
  boundary[GHOST(1), NONE];
}
```

Figure 11: The distributed array used in the knapsack problem.

so *pipelining* is used to maximize parallelism: thread 0 computes its first column while the other threads are idle, then thread 0 computes its second column and thread 1 computes its first column while the rest are idle, and so on, proceeding in a diagonal fashion down the matrix. In order to reduce synchronization, we compute blocks of columns instead of a single column at each step, with a block size of 100 providing the best performance [9].

Figure 11 shows the distributed array specification for the knapsack problem. Unlike in the game of life, the access pattern is specified, with each thread blocking in the capacity dimension by a size of 100. The code to compute the matrix values is shown in Figure 12. Though the code is simplified by the presence of distributed arrays, the data dependencies still have to be explicitly handled in the code. As a result, a `forall` loop does not suffice, and specifying the access pattern does not help either.

# 6 Discussion

In this section, we discuss some of the limitations of the current distributed array synthesizer and areas of future work.

## 6.1 Limitations

The array synthesizer is a very early prototype, so some details have been ignored for convenience. It does not support distributed arrays over domains with non-unit stride. The code has been written to assume that everything divides evenly, such as the block sizes into the array lengths, and unexpected errors may occur if this is not the case.

```
int startIter = Ti.thisProc();
int endIter = Ti.thisProc() + blocks-1;
for (int single iter = 0; iter < Ti.numProcs() + blocks-1; iter++) {
  if (iter >= startIter && iter <= endIter) {
    int myIter = iter - startIter;
    for (int book = startBook; book <= endBook; book++)
      for (int cap = myIter * blockSize; cap < (myIter+1) * blockSize; cap++)
        if (weight[book] <= cap) {
          int newProfit = profit[book] + matrix[book-1,cap-weight[book]]
          matrix[book,cap] = max(matrix[book-1,cap], newProfit);
        }
  }
  matrix.synchronize();
}
```

Figure 12: The actual code for computing the matrix values in the knapsack problem.

## 6.2 Future Work

There are many possible areas of future work, involving adding support for more array types, analyses, and optimizations.

### 6.2.1 Array Types

The code generator currently only supports simple distributions of regular arrays. More distribution types can be added, such as blocked-cyclic, and other boundary conditions besides ghost cells can be provided. Irregular arrays can also be added, such as those used in adaptive mesh refinement, with support for refining subsets of an array. It may even be possible to extend the generator to automatically distribute an array so that load is balanced evenly over all processors.

### 6.2.2 Analyses and Optimizations

In this project, we focused on functionality over speed, so the Titanium compiler currently makes no attempt to analyze and optimize code that uses distributed arrays. As a result, it is likely that the distributed array implementation is not as fast as it could be.

Under the current implementation, accessing a distributed array requires a locality check or thread owner computation. It is often the case that a programmer writes his code such that no remote accesses occur. The compiler should be able to infer this and eliminate the check.

The code generator currently only uses the provided access pattern to provide extra functions to the user, which may not be useful, as in §5.2. The generator could also use this information to modify data layout in order to take full advantage of the machine's cache and other characteristics.

As was shown in §5.2, data dependencies must be explicitly reflected in the structure of the code to operate over a distributed array. Not only does this complicate the code, it also is error-prone, and programmers sometimes don't even know what the dependencies are. With powerful enough analyses, the compiler may be able to determine what they are and remove the need to explicitly specify them, as well as optimize the data layout and access patterns accordingly.

Though distributed arrays simplify code, it is still necessary for the programmer to determine the optimal parameters to specify. It should be possible to develop a system that automatically tunes the parameters, perhaps by running the program using different parameters to determine the best ones.

# 7 Related Work

There has been extensive work on distributed arrays in various parallel languages, and we only highlight some of it here. Most parallel languages support distributed arrays in some fashion, though many languages such as Unified Parallel C [4] and Co-Array Fortran [11] only allow them to be distributed in a single dimension. Others such as High Performance Fortran [8]

and High Performance Java [5] allow distribution in all dimensions, like we do here. To our knowledge, HPJava is the only language that provides ghost cells in distributed arrays.

Data-parallel languages like ZPL, NESL, HPF and pC++ go even further than distributed arrays in simplifying programs [2, 3, 8, 12]. In such languages, the degree of parallelism is determined by the data structures in the program, and need not be expressed directly by the user. These languages include array operators for element-wise arithmetic operations, e.g., $C = A+B$ for matrix addition, as well as reduction and scan operations to compute values such as sums over arrays. However, it is very difficult to express many parallel algorithms in these languages.

To our knowledge, no language currently allows specification of access patterns or allows multiple refinement levels. For the latter, libraries such as Chombo have been built in both C++ and Titanium [1, 13].

# 8 Conclusion

In this paper, we presented a framework for specifying and using distributed arrays in Titanium. The framework allows specification of array dimension, distribution, access patterns, and replication, including ghost cells and multiple refinement levels. We showed that the resulting code could easily be used to implement two algorithms in Titanium, the game of life and a pseudo-polynomial knapsack solver.

This work only scratches the surface of the problem of distributed data structures. It only supports simple options for each array feature, some of which could not be completed due to time limitations, such as multiple refinement levels. Whole classes of structures are not yet supported, such as irregular grids and pointer-based structures, and many analyses and optimizations are yet to be implemented. The problem of distributed data structures is far from solved, but this work makes significant progress towards that goal.

# References


[1] Chombo website: `http://seesar.lbl.gov/ANAG/software.html`. Applied Numberical Algorithms Group (ANAG), Lawrence Berkeley National Labratory, Berkeley, CA.

[2] G. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-95-170, Carnegie-Mellon University, September 1995.

[3] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.

[4] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[5] B. Carpenter, H.-K. Lee, S. B. Lim, G. Fox, and G. Zhang. Parallel Programming in HPJava, April 2003. Pervasive Technology Labs, Indiana University.

[6] B. L. Chamberlain, S. J. Deitz, and L. Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[7] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an NPB experimental study. In *Languages and Compilers for Parallel Computing*, 2005.

[8] High Performance Fortran Forum. High Performance Fortran Language Specification. http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf-v20, Jan. 1997.

[9] A. Kamil, M. Ngo, Y. Patel, F. Gennari, G. Díez-Cañas, and C. Chang. Parallel knapsack. `www.cs.berkeley.edu/~kamil/cs267/hw3/hw3.pdf`.

[10] R. W. Numrich, J. Reid, and K. Kim. Writing a multigrid solver using co-array fortran. In *Proceedings of the Fourth International Workshop on Applied Parallel Computing, Umea, Sweden*, June 1998.

[11] R. Numwich and J. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.

[12] L. Snyder. *The ZPL Programmer's Guide*. MIT Press, 1999.

[13] T. Wen and P. Colella. Adaptive mesh refinement in Titanium. *19th International Parallel and Distributed Processing Symposium (IPDPS)*, 01:89a, 2005.

[14] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *Workshop on Java for High-Performance Network Computing*, Stanford, California, February 1998.