The Design and Development of ZPL

Lawrence Snyder
Department of Computer Science and Engineering
Box 352350
University of Washington
Seattle, WA 98195-2350
snyder@cs.washington.edu

Abstract

ZPL is an implicitly parallel programming language, which means all instructions to implement and manage the parallelism are inserted by the compiler. It is the first implicitly parallel language to achieve performance portability, that is, consistent high performance across all (MIMD) parallel platforms. ZPL has been designed from first principles, and is founded on the CTA abstract parallel machine. A key enabler of ZPL's performance portability is its What You See Is What You Get (WYSIWYG) performance model. The paper describes the antecedent research on which ZPL was founded, the design principles used to build it incrementally, and the technical basis for its performance portability. Comparisons with other parallel programming approaches are included.

Categories and Subject Descriptors D.3.2 [**Programming Languages**]: Language Classifications – Concurrent, distributed and parallel languages; D.3.3 Languages, Constructs and Features – Concurrent programming structures, Control structures, Data types and structures; D.3.4 Processors – Compilers, Retargetable compilers, Optimization, Run-time Environments; C.1.2 Multiple Data Stream Architectures – Multiple-instruction-stream, multiple-data-stream architectures (MIMD), Interconnection architectures; C.4 Performance of Systems – Design studies, Modeling studies, Performance attributes; E.1 Data Structures – Distributed data structures; F.1.2 Modes of Computation – Parallelism and concurrency.

General Terms: Performance, Design, Experimentation, Languages.

Keywords: performance portability; type architecture; parallel language design; regions; WYSIWYG performance model; CTA; problem space promotion.

1. Starting Out

On October 7, 1992 Calvin Lin clicked on an overhead projector in a Sieg Hall classroom on the University of Washington campus in Seattle, and began to defend his doctoral dissertation, *The Portability of Parallel Programs Across MIMD Computers*. The dissertation, which was a feasibility study for a different approach to parallel programming, reported very promising results. At the celebratory lunch at the UW Faculty Club we agreed that it was time to apply those results to create a new language. We had studied the critical issues long enough; it was time to act. At 9:00 AM the following morning he and I met to discuss the nature of the new language, and with that discussion the ZPL parallel programming language was born.

It would be satisfying to report that that morning meeting was the first of a coherent series of events, each a logical descendant of its predecessors derived through a process of careful research and innovative design, culminating in an effective parallel language. But it is not to be. ZPL *is* effective, being a concise array language for fast, portable parallel programs, but the path to the goal was anything but direct. That we took a meandering route to the destination is hardly unique to the ZPL Project. We zigzagged for the same reason other projects have: language design is as much art as it is science.

What was unique to our experience was designing ZPL while most of the parallel language community was working cooperatively on High Performance Fortran (HPF), a different language with similar goals. On the one side was our small, coherent team of faculty and grad students, never more than ten, designing a language from scratch. On the other side was a distributed, community-wide effort with hundreds of participants, generous federal funding, and corporate buy-in dedicated to parallelizing Fortran 90. It had all of the features of a David and Goliath battle. Language design is much more complex and subtle than combat, however, so "battle" mischaracterizes the nature of the competition, as I will explain.

In the following I describe the design and development of ZPL. One objective is to explain the important technical decisions we made and why. Their importance derives both from their contributions to ZPL's success and their potential application in new (parallel) languages. The other objective is to convey the organic process by which ZPL was created, the excitement, the frustrations, the missteps, the epiphanies. Language design for us was a social activity, and just as a software system's structure often reflects the organization of the group that created it, the ZPL team pressed its unique profile into the resulting language.

©2007 ACM 978-1-59593-766-7/2007/06-ART8 $5.00
DOI 10.1145/1238844.1238852
http://doi.acm.org/10.1145/1238844.1238852

To follow the winding path, it is necessary to know the intended destination.

2. Goals

By the morning of that first meeting the goals were well established. They were even established before Lin started his dissertation, because the whole ZPL effort was preceded by a decade of parallel programming research. Our earlier projects had created two other parallel languages and a parallel programming environment, tested them and found them wanting. Though they failed to achieve the design goals, each effort let us explore the design space and test the efficacy of different approaches. Lin's dissertation was the transitional research moving us from exploring the problem to solving it. All of these attempts shared the same objective: To be a practical language that delivers performance, portability and convenience for programming large scientific and engineering computations for parallel computers.

2.1 Performance

The performance goal is self-evident because the main reason to program a parallel computer is to complete a computation faster than with a sequential solution. A parallel programming language that cannot *consistently* deliver performance will not be used. Consistency is essential because programmers will not spend effort programming on the *chance* that the code runs fast. It must be a certainty. Further, the degree to which the parallel solution outperforms the sequential solution is also important, since it is a measure of the programmer's success. Because of the naïve assumption that *P* processors should speed up a computation by a factor of *P*, users expect their programs to approach that performance. In consequence language designers must repress any temptation to "spend" portions of that factor-of-*P* potential for language facilities. A common example is to introduce log *P* overhead to implement the shared memory abstraction [1]. Our performance *goal* was to deliver the whole potential to the programmer.

2.2 Portability

Program portability, meaning that programs "perform equivalently well across all platforms," was the second property we wanted to achieve.[1] Portability, once a matter of serious concern, had not been a significant issue in programming language design for sequential computers since the development of instruction set architectures (ISAs) in the 1960s. Sequential computer implementations are extremely similar in their operational behavior and any programming language more abstract than assembly language could be effectively compiled to 3-address code with a stack/heap memory model, and then translated to the ISA of any sequential computer.

By contrast, parallel computer architecture had not (has not) converged to a single machine model. Indeed, when the ZPL effort began, large **SIMD** (single instruction stream, multiple data stream) computers were still on the market. As it was obvious from the beginning that SIMD machines are too limited to support general purpose computing, our focus has always been on **MIMD** (multiple instruction stream, multiple data stream) machines. Among MIMD computers, though, dozens of different architectures have been built and many more proposed during the period of our programming language research [2]. Compiling efficient code for all of them is a challenge.

Portability was an issue because, as one researcher from Lawrence Livermore National Laboratory (LLNL) lamented at a Salishan meeting[2] at the time, "Before we get the program written they roll in a new computer, and we have to start from scratch." The problem, as in the 1950s, was that programmers tried to achieve performance by exploiting specific features peculiar to the architecture, and computer vendors encouraged them to do so to secure their market advantage. For example, the popular-at-the-time hypercube communication network was often explicitly used in programs [3]. Once the topology was embedded into the code, the program either had to be rewritten or considerable software overhead had to be accepted to emulate a hypercube when the next machine "rolled in" with a different topology. This problem was extremely serious for the national labs, accustomed as they were to running programs for years, the so-called "dusty decks."

Achieving portability in parallel language design entails resolving a tension: Raising the level of abstraction tends to improve portability, while programming at a lower level, closer to the machine, tends to improve performance. Raising the level high enough to overtop all architectural features, however, may penalize performance to an unacceptable degree if layers of software are needed to map the abstractions to the hardware. The purpose of Lin's dissertation research was to validate the portability solution we intended to use in ZPL (the CTA approach, see below).

As a postscript, portability as used here, meaning portability with (parallel) performance, was not much discussed in the literature as the Lin work was being published. I think we were the only group worried about it. Soon, it became widely mentioned. It was never clear to me that the researchers who claimed portability for their languages understood the tension just described. Rather, I believe they interpreted portability in what I began thinking of as the "computability sense." Computability's universality theorem says any program can run on any modern computer by (some amount of) emulation. This fact applies to parallel programs on parallel computers. All parallel languages produce portable programs in this sense. Our portability goal required a program in language X, which delivered good performance on platform A, quantified, say, as 80% of the performance of a custom, hand-coded program for A, to deliver roughly the same good performance when compiled and run on platform B. This is the sort of portability sequential programs enjoy, and users expect it of parallel programs, too. Portability in the "computability sense" says nothing about maintaining performance.

[1] "Portability" has a range of meanings from "binary compatible" to "executable after source recompilation;" the latter applies here.

[2] The national labs have regularly held a meeting of parallel computation experts at the Salishan Conference Center on the Oregon Coast.

2.3 Convenience

Convenience, meaning ease of use, was the third goal. Inventing a language that is easy to use is a delicate matter. To aid programmers, it is important to match the language's abstractions to the primitive operations they use to formulate their computations. When we began (and it remains true) there were few "standard" parallel language constructs. What statement form do we use to express a pipelined data structure traversal? What is standard syntax for writing a (custom) parallel prefix computation? It is necessary to create new statement forms, but doing so is subtle. They must be abstract, but still give programmers effective control. Of course, picking syntax for them is guaranteed to create controversy.

On the other side of the convenience equation, the problem domain influences the choice of language facilities. Because high-end applications tend to be engineering and scientific computations, good support for arrays, floating-point computations and libraries was essential. Additionally, keeping the memory footprint contained was also essential, because high-end computations are as often memory bound than computation bound.

2.4 Additional Constraints

Our plan was to build ZPL from scratch rather than extending an existing (sequential) language. Our two previous language designs included an instance of each approach. Many benefits have been cited for extending an existing language, such as a body of extant programs, knowledgeable programmers, libraries, etc. It is an odd argument, since these resources are all sequential. To exploit them implies either that users are content to run sequential programs on a parallel computer, or that there is some technique to transform existing sequential programs into parallel programs; if that is possible, why extend the language at all?

While on the topic of techniques to transform sequential programs, notice that the "parallelizing compiler" approach is the ultimate solution to the *practical* parallel programming problem: It produces—effortlessly from the programmer's viewpoint—a parallel program from a sequential program. It seems miraculous, but then compilers are pretty amazing anyhow, so it is easy to see why the approach is so often pursued. I first argued in 1980 [4] that this technique would not work based on two facts: (1) compilers, for all their apparent magic, simply transform a computation from a higher to a lower level of abstraction without changing the computational steps significantly; (2) solving a problem in parallel generally requires a paradigm shift from the sequential solution, i.e. the *algorithm* changes. Despite considerably more progress than I might have expected, the prospects of turning the task over to a compiler remain extremely remote. ZPL was focused on writing *parallel* programs (as were all of its ancestors).

Our experience extending an existing language (Orca C, discussed below) demonstrated the all-too-obvious flaw in trying extend a sequential language: With an existing language, programmers familiar with it expect its statements and expressions to work in the usual way. How can this expectation be managed? *Preserving* the sequential semantics while mapping to a parallel context is difficult to impossible, especially considering that performance is the goal. *Changing* the semantics to morph them into a parallel context is annoying to infuriating for programmers, especially considering the potential for introducing subtle bugs; further, it renders the existing program base useless. Having done it once, I was not about to repeat the mistake. So, we started from scratch—we called it *designing from first principles.* This also has the advantage that the needs of the programmer and the compiler can be integrated into the new language structures.

Finally, we decided to let parallelism drive the design effort, avoiding object orientation and other useful or fashionable, but orthogonal, language features. If we got the parallelism right, they could be added. Before that point they would be distracting.

3. Overview of the ZPL Project

To build a superstructure around ZPL to allow direct access to the interesting parts of the research, we take a rapid tour through nearly a dozen years of the project.

3.1 The Time Line

Prior to the October 1992 meeting, several efforts in addition to Lin's dissertation produced foundations on which we built. One of these, the CTA Type Architecture, is especially critical to the language's success and will be described in detail in a later section.

Beginning in October, Lin and I worked daily sketching a language based on his dissertation work. By March, 1993, the language was far enough along that he could present it at the final meeting of the Winter Quarter's CSE590-O, *Parallel Programming Environments Seminar*. His hour-long lecture was intended to attract students to the Spring Quarter CSE590-O, which we planned to make into a language design seminar. (The language was not yet named.) We would program using the language's features, as well as try to extend its capabilities. The advertisement was successful. The project began that March with seminar students Ruth Anderson, Brad Chamberlain, Sung-Eun Choi, George Foreman, Ton Ngo, Kurt Partridge and Derrick Weathersby. A few terms later, E Chris Lewis joined to be the final project founder. See Figure 1.

As the seminar students learned the base language and began contributing to the design in Spring 1993, names were frequently suggested, usually humorously. Lin had called it Stipple in his presentation, because he had used stipple patterns liberally in his diagrams. I had called it Zippy, suggesting it produced fast programs, but that was too informal. Eventually, we gravitated to ZPL, short for Z-level Programming Language, as explained below.

With performance as the first goal, we adopted Project Policy #1: Only include in the language those features that we know how to compile into efficient code. At this point our main guidance on what could be done efficiently was Lin's dissertation. So, the language contained very little at the start. Indeed, when the design was first described in 1993 [5], ZPL was called "an array sublanguage" because so many features were missing. There was enough expressiveness, however, to cover common inner loop computations. For example, the 4-nearest neighbor stencil iteration could be expressed with these lines (plus declarations),

```
[1..n,1..n] repeat
              Temp := (A@north + A@east
                       + A@west + A@south)/4;
              err := max<<abs(Temp-A);
```

Figure 1. Snapshot from a ZPL Design Meeting, 1995. Standing, from left, Jason Secosky, E Chris Lewis, Larry Snyder, Ton Ngo, Derrick Weathersby and Judy Watson; seated, Sung-Eun Choi, Brad Chamberlain, Calvin Lin

```
    A := Temp;
until err < tolerance;
```

The new programming abstractions introduced by ZPL at this point were *regions* (the indices in brackets), *directions* (the @-constructs), and the *global-view distributed memory model*, as explained below.

As we designed we applied the general heuristic of optimizing for the common case. This would often have a profound effect on the language. Most frequently it caused us to decide *not* to adopt a too-general approach; we would table the final decision pending further inspiration.

Compiler construction began by summer 1993. Project Policy #2: The compiler must always create fast and portable code, starting at the beginning. The next year, the performance and portability results reported for ZPL [6] showed that our research goals could be achieved, at least for the sublanguage. The policy had, at least, been instantiated properly. Nearly all of the ZPL papers show performance results on several different parallel architectures.

Project meetings and implementation started out in the Blue CHiP Lab in Sieg Hall, but with the Computer Science and Engineering department critically short on space, we gave up the lab after the summer of 1994. The team never had another lab, despite the tremendous benefits of having a focal point for a research project. The grad students were distributed across several offices, most of them in a temporary manufactured building next to Sieg Hall known to the team as Le Chateau. With everyone scattered, implementation was coordinated by network, and the design discussions were conducted in conference rooms. In my opinion the absence of lab space slowed progress to a noticeable degree.

During 1995-6 the team extended ZPL's expressiveness substantially, and our few users gave us feedback on the design. Indeed, in accordance with Project Policy #3: Language design proceeded with a handful of representative operations or computations drawn from user applications. By this we hoped to match the needs of the problem domains. By 1996 a succinct, high-level stand-alone language had been designed and implemented —dubbed ZPL Classic by the designers.

Comparisons with standard benchmarks, for example, the SPLASH and xHPF suites, showed that the ZPL compiler was doing well on such parallel-specific aspects as communication performance, but that its scalar performance was only average. For these well-worked benchmark codes, programmers could incorporate hand optimizations into their low-level message passing programs to make them extremely efficient. In high-level languages like ZPL it is the compiler that is responsible for such optimizations because programmers do not write the low-level code like array-traversal loops. So, after an additional year devoted mostly to improving ZPL's scalar performance, the compiler was officially released in July 1997.

The team changed slowly over time. Jason Secosky joined in 1994. Lin joined the faculty at UT Austin in 1996. Ruth Anderson, George Forman and Kurt Partridge moved on to other dissertation topics. Taylor van Vleet, Wayne Wong and Vassily Litvinov joined the project, con-

tributed and moved on. Ton Ngo returned to IBM in 1996. In 1997 Maria Gulickson, Douglas Low and Steven Deitz joined as project members. Periodically, researchers from client fields, such as chemistry, astronomy and mechanical engineering, would join our group meetings.

One effect of Policy #1, the "add no feature until it compiles to fast code" policy, was to make ZPL Classic a rather static language that emphasized creating structures through declarations rather than building them on the fly. But creating ZPL Classic had also given us sufficient insight to be more dynamic. So, following the completion of ZPL Classic we began work on Advanced ZPL (A-ZPL) to enhance programmer flexibility. New features included built-in sparse arrays, upgrading regions and directions to first-class status, and introducing facilities for dynamically controlling data distributions and processor assignments. The resulting language was much more flexible.

Users have programmed in ZPL since 1995. ZPL programs are much more succinct than programs written in other general approaches. Experiments show that ZPL programs consistently perform well and scale well. On the NAS Parallel Benchmarks ZPL nearly matches performance against the hand-optimized MPI with Fortran or C; on user-written programs, ZPL is generally significantly faster than MPI programs. (See below.) Our ZPL-to-C compiler emits code that can be compiled for any parallel machine and typically requires little tuning. The ZPL compiler has been available open source since 2004. Concepts and approaches from ZPL have been incorporated into Cray's Chapel, IBM's X10 and other parallel languages.

4. Contributions

Though they are described in detail below, it may be helpful to list the main contributions of the ZPL project now as a link among various threads of the story.

I believe the principal contributions are

- Basing a parallel language and its compiler on the CTA machine model (type architecture).
- Creating a global view program language for MIMD parallel computers in which parallelism is inherent in the operations' semantics.
- Developing the first parallel language whose programs are fast on all MIMD parallel computers.
- Designing from first principles parallel abstractions to replace shared memory.
- Creating new parallel language facilities including regions, flooding, remap, mscan and others, all described below.
- Applying various design methodologies, including an always fast and portable implementation, designing from user problems, evolutionary design, etc.
- Creating the problem space promotion parallel programming technique.
- Inventing numerous compilation techniques and compiler optimization approaches including the Ironman communication layer, loop-fusion and array contraction algorithms, the factor-join technique, and others.

There are many other smaller contributions, but these successes most accurately characterize the advances embodied in ZPL.

5. The Programmer's View of ZPL

To illustrate ZPL programming style in preparation for a careful explanation of its technical details, consider writing a program for Conway's Game of Life. Over the years we used various introductory examples, but we eventually adopted the widely known Game of Life and joked that it solved biologically inspired simulation computations.

```
program Life;                                   Conway's Life
   /* In a toroidal world externally initialized, organisms with 2 or 3
      neighbors survive to the next generation; empty cells with 3
      neighbors give birth in next generation; all others die */

  config const n : integer = 16;                The world is n x n; default to 16; change on command line.
  region R = [1..n, 1..n];                      Index set of computation; n^2 elements.
  direction                                     Reference vectors for 8 nearest neighbors ...
    nw = [-1, -1]; no = [-1, 0]; ne = [-1, 1];  Vectors "point" in index space not Euclidean space
    w  = [ 0, -1];               e  = [ 0, 1];
    sw = [ 1, -1]; so = [ 1, 0]; se = [ 1, 1];
  var                                           Declarations for two array variables both n x n.
    TW : [R] boolean;                           Problem state, The World
    NN : [R] byte;                              Work array, Number of Neighbors

procedure Life();                               Entry point procedure; same name as program.
  begin
    -- Initialize the world here
  [R] repeat                                    Region R means all operations apply to all indices.
      NN := TW@^nw + TW@^no + TW@^ne             Add eight nearest neighbor bits (type coercion follows C)
         +  TW@^w  +          TW@^e             Caret(^) means neighbor is a toroidal reference at array edge.
         +  TW@^sw + TW@^so + TW@^se;
      TW := (TW & NN = 2) | (NN = 3);           Update world with next generation.
    until !(|<< TW);                            Continue till all die out; use an OR reduction.
end;
```

Figure 2. Conway's Game of Life in ZPL. The problem state is represented by the *n*×*n* Boolean array TW. On each iteration of the repeat loop a generation is computed: the first statement counts for each grid position the number of nearest neighbors, and the second statement applies the "rules"; the iteration ends when no more cells are alive.

5.1 *Life* As A ZPL Program

Recall that the Game of Life models birth and death processes on a two dimensional grid. The grid is toroidally connected, meaning that the right neighbors of the right edge positions are the corresponding positions on the left edge; similarly for the top and bottom edges. The game begins with an initial configuration, the 0th generation, in which some grid positions are occupied by cells. The *i*+1st generation is found from the *i*th generation by eliminating all cells with fewer than two neighbors or more than three; additionally, any vacant cell with exactly three neighbors has a cell in the *i*+1st generation. See Figure 2 for the ZPL program simulating Life.

The following ZPL concepts are illustrated by the Life program.

The `config const` is a declaration setting `n`, the length of the edge of the world, to 16. Configuration constants and variables are given default values in their declarations that can be changed on the command line later when the program runs.

Regions, which are new to ZPL, are like arrays without data. A **region** is an index set, and can take several forms. The **index range** form used in Life,

```
region R = [1..n, 1..n];
```

declares the lower and upper index limits (and an optional stride) for each dimension. (Any number of dimensions is allowed.) In this example, `R` is the set of n^2 indices {(1,1), (1,2), (1,3), ..., (*n*,*n*)}.

The next declaration statements in the Life program define a set of eight directions pointing in the cardinal directions. A **direction** is a vector pointing in index space. For example, `no = [-1, 0]` is a direction that for any index (*i*,*j*) refers to the index to its north (above), i.e. (*i*-1, *j*). Directions, another concept introduced in ZPL, provide a symbolic way to refer to near neighbors.

Regions are used in two ways. In

```
var
    TW : [R] boolean;
    NN : [R] byte;
```

the region (`[R]`) provides the indices for declaring arrays. So, the world (`TW`) is an n^2 array of Boolean values.

The other use of regions is to control the extent of array computations. For example, by prefixing the `repeat`-statement with `[R]` we specify that all array computations in the statement are to be performed on indices from `R`. That is, the operations of the two assignment statements in the body of the `repeat` and the loop condition test, all of which involve expressions whose operands are arrays, will be performed on the n^2 values whose indices are given in the region `R`. In this way programmers have global (declarative) control over which parts of arrays are operated upon.

The computation starts by generating or inputting the initial configuration. Within the `repeat`-loop the first statement

```
NN := TW@^nw + TW@^no + TW@^ne
    + TW@^w           + TW@^e
    + TW@^sw + TW@^so + TW@^se;
```

counts the number of neighbors by type-casting Booleans to integers. This is an array operation applying to all elements in the region `R`. Therefore `TW@^no`, for example, refers to the *array* of elements north of the elements in `R`; `@^` is verbalized **wrap at**. The indices for these north neighbors are found by adding the direction vector to the elements of `R`, i.e. `R+[-1,0]`. The translated indices (with wraparound) reference an $n \times n$ array of "north neighbors," and similarly, for the other direction-modified addition operands. The additions are performed element-wise, producing an $n \times n$ result array whose indices are in `R`. The result, the count of neighbors, is assigned to `NN`.

The next statement

```
TW := (TW & NN = 2) | (NN = 3);
```

applies the Game of Life rules to create the `TW` Boolean array for the next generation. Thus, any position where a cell exists in the current generation and has two neighbors, or the position has three neighbors, becomes true in the next generation. As before, these are array operations applying to all elements of `R` as specified by the enclosing region scope.

Finally, the loop control

```
until !(|<< TW);
```

performs an OR-reduction (`|<<`) over the new generation; that is, it combines the elements of `TW` using the OR-operation. If the outcome is not true, no cells exist and the repetition stops.

It is a small computation and only requires a small ZPL program to specify.

5.2 Parallelism in *Life*

Although the Life program contains some unfamiliar syntax and several new concepts, it is composed mostly of routine programming constructs: declarations, standard types, assignment statements, expressions, control structures, etc. Indeed, it doesn't look to be parallel at all, since the familiar concepts retain their standard meanings: declarations state properties without specifying computation, statements are executed one at a time in sequence, etc.

The Life program is **data parallel**, meaning that the concurrency is embodied in the array operations. For example, when the eight operand-arrays are added element-wise to create the value to be assigned to `NN`, the order of evaluation for the implementing scalar operations is not specified, and can be executed in parallel.

ZPL programmers know much more about the program's parallelism than that. Using ZPL's WYSIWYG Performance Model (explained below), programmers have complete information about how the program is executed in parallel: They know how the compiler will allocate the data to the processors, they know a small amount of point-to-point communication will be performed at the start of each loop iteration, they know that operations of each statement will be fully concurrent, and that the termination test will require a combination of local parallel computation followed by tournament tree and broadcast communication. (Details are given below.) Though programmers do not write the code, they know what the compiler will produce.

6. Antecedents

To set the context for the ZPL development, recall that in the 1980s and early 1990s views on how to program parallel machines partitioned into two categories that can be dubbed: No Work and Much Work. The No Work commu-

nity expected compilers to do all of the parallelization. Most adherents of this view believed a sequential (Fortran) program was sufficient specification of the computation, although a sizeable minority believed one of the more "modern" functional or logic or other languages was preferable. The Much Work community believed in programming to a specific machine, exploiting its unique features. Their papers were titled "The *c* Computation on the *d* Computer," for various values of *c* and *d*, much as technical papers had been in the early days of computing. And conference series such as the Hypercube Conference were devoted to computations on parallel computers defined by their communication topology (binary *n*-cube). Given the state of the art, the No Work community could have portability but little prospect for performance, while the Much Work community could have performance but little prospect of portability. One without the other is useless in practice. We thought we knew how we might get both, but we were exploring new territory.

6.1 The Importance of Machines

I take as a fundamental principle that programming languages designed to deliver performance should be founded on a realistic machine model [1]. Since the main point of parallel programming is performance, it follows that the principle applies to all parallel languages. I didn't always know or believe the principle; it emerged in work on ZPL's language antecedents. The first of these was the Poker Parallel Programming Environment, a product of the Blue CHiP Project and my first real attempt at supporting parallel programming [7].

The Blue CHiP Research Project designed and developed the Configurable, Highly Parallel (CHiP) computer [8], a MIMD parallel machine with programmable interconnect like FPGAs of today. That is, the processors are not permanently connected to each other in a fixed topology; the programmer specifies how the processors are to connect. The programmer designing a "mesh algorithm" connects the processors into a mesh topology, and when the next phase requires a tree interconnect the processors are reconfigured into a tree. The CHiP machine was my answer to the question, "How might VLSI technology change computers?"

While in the throes of machine design in January 1982, I organized a team of a dozen grad students and faculty to begin work on its programming support. Poker was a first attempt at an Interactive Development Environment (IDE) for parallel programming; it was intended to support every aspect of CHiP computer programming.[3] Because the CHiP computer is MIMD, Poker provided the ability to write code for each processor; because the CHiP computer is configurable, it provided the ability to program how the processors should be interconnected; because parallel programming was seen as occurring in phases, it provided the ability to orchestrate the whole computation, globally.

Though we thought of it as a general-purpose programming facility, Poker was actually tightly bound to the underlying CHiP architecture, a 2D planar structure designed more to exploit VLSI than to provide a general-purpose parallel computing facility. Poker was perfectly matched to computations that could run efficiently on the CHiP platform, that is, VLSI-friendly computations such as systolic algorithms and signal processing computations. But the language became more difficult to use in proportion to how misaligned the computation was with the CHiP architecture. The tight binding between the CHiP machine and the programming language limited Poker's generality, but the successful cases clearly illustrated the value of knowing the target machine while programming. This was an enduring contribution of the Poker research: Programmers produce efficient code when they know the target computer, but directly exploiting machine features embeds assumptions about how the machine works and can impose unnecessary requirements on the solution.

If not the CHiP architecture, then what? At the other end of the spectrum from specific architectures are the abstract concurrent execution environments used for functional languages, logic languages, PRAM computations, etc., all of which were extremely popular in the 1980s [9-11]. These very abstract "machines" are powerful and flexible; they do not constrain the programmer and are easy to use. But how are they implemented efficiently? They did not (and do not) correspond to any physical machine, and emulating them requires considerable overhead. It was evident that fundamental issues concerning memory reference limit the feasibility of all of these approaches. A solution might one day be found, but in the meantime the practical conclusion I came to was: It is just as easy to be too general about the execution engine as it is to be too specific.

6.2 Type Architecture

The sum of what I thought I knew in the summer of 1985 could be expressed in two apparently contradictory statements:

- A tight binding between a programming facility and a machine leads to efficient programs, but by using it programmers embed unnecessary assumptions into the solution.
- A weak or non-existent binding between a programming facility and a physical machine leads to easy-to-write computations that cannot be efficiently implemented.

[3] Of course, there were no IDEs to build on at the time, but the Alto work at Xerox PARC [65] illustrated a powerful interactive graphic approach, which we did try to borrow from with Poker. We designed Poker around a bit-mapped display attached to a VAX 11/780, our proto-workstation. Since another VAX 11/780 supported the entire Purdue CS department, our profligate use of computer power to service a single user was deemed ludicrous by my faculty colleagues. It is quaint today to think that dedicating a 0.6 MIPS computer with 256K RAM is wasteful; the surprising aspect of it was how quickly (perhaps 3-4 years) it became completely reasonable. Our Poker system used windows and interactive graphics, but we didn't understand the windows technology well enough. The resulting system was very cumbersome and fragile. Trying and failing to make a technology work gives one a deep appreciation and respect for those who do get it right [66].

I decided to explain these two "facts" in an invited paper that eventually appeared under the title *Type Architecture, Shared Memory and the Corollary of Modest Potential* [1]. The paper was purposely provocative at the request of the *Annual Reviews* editors. Accordingly, rather than presenting the two "facts" directly, it argued that parallel models based on a flat, shared memory—the usual abstraction when implementation is ignored and the assumption in the functional/logic/PRAM world—are best avoided on the grounds of poor practical parallel performance. (The point was made more rigorously sometime later [12].) The paper went on to present an alternative, the CTA Type Architecture.

A **type architecture** is a machine model abstracting the performance-critical behavior of a family of physical machines. The term "type" was motivated by the idea of a "type species" in biology, the representative species that exhibits the characteristics of a family, as in the frog *Rana rana* characterizing the family Ranidae. Alas, given how overused the word "type" is in computer science, it was an easily misinterpreted name.

Machine models have been common since Turing introduced his machine in 1936 [13]. But type architectures are not just machine models. They are further constrained to describe accurately the key performance-critical features of the machines, usually in an operational way. For example, in the familiar Random Access Machine Model (*a.k.a.* the von Neumann machine), there is a processor, capable of executing one instruction at a time, and a flat memory, capable of having a randomly chosen, fixed-size unit of its storage referenced by the processor in unit time; moreover, instructions are executed in sequence one per unit time. These characteristics impose performance constraints on how the RAM model can work that inform how we program sequential computers for performance. (Backus described some of these features as the von Neumann Bottleneck [14], and used them to motivate functional languages.) Notice that most features of physical machines are ignored —instruction set, registers, etc.—because they affect performance negligibly.

The idea of a type architecture derived from several months' reflection on how we write efficient sequential programs to be used as an analogy for how we should write efficient portable parallel programs. The result of this thought-experiment was essentially making explicit what we all understand implicitly: Usually, programmers don't know what their target machine will be and it changes frequently, yet they are perfectly effective writing fast programs in languages like C. Why? Because

- they understand the performance constraints of the RAM model,
- they understand generally how C maps their code to the model, and
- when the program runs on real hardware it performs as the model predicts.

What is essential about the RAM model is that it gives a true and accurate statement of what matters and what doesn't for performance. For example, programmers prefer binary search to sequential search because random access is allowed, and 1-per-unit-time data reference and 1-per-unit-time instruction execution implies sequential search is slower when the list is longer than several elements. It's not enough to give a machine model; Turing gave a machine model. It's essential that the model accurately describe the performance that the implementing family of machines can deliver. A type architecture does that.

Notice that Schwartz's idealistic but otherwise brilliant "Ultracomputer" paper [15] had tried to ensure some degree of realism in a parallel model of computation. His work influenced my thinking, though he was more concerned with algorithmic bounds than production parallel programs, and he was prescriptive rather than descriptive.

So, in the same way that the RAM model abstracts sequential machines and is the basis for programming languages like C, enabling programmers to know the performance implications of their decisions and to produce portable and efficient sequential programs, a type architecture abstracting the class of MIMD parallel machines could be used to define a language enabling programmers to write code that runs well on all machines of that type. MIMD parallel machines, unlike sequential computers, exhibit tremendous variability. What is their type architecture like?

6.3 CTA – Candidate Type Architecture

Abstracting the key characteristics of MIMD parallel machines was somewhat easier than one might have expected given the wide variability of real or proposed architectures. Clearly, it has some number, P, of processors. What's a processor? Because the parallel machines are MIMD, the processors each have a program counter, program and data memory, and the ability to execute basic instructions; so the RAM type architecture is a convenient (and accurate) definition of a processor. This gives the CTA the attractive property that a degenerate parallel computer, that is, $P = 1$, is a sequential computer, establishing a natural relationship between the two.

Besides processors, the other dominant characteristic of parallel machines is the communication structure connecting the processors. Because connecting every pair of processors with a crossbar is prohibitively expensive, parallel architectures give up on a complete graph interconnection and opt for a sparser topology. I had expected that the parallel type architecture would specify some standard topol-

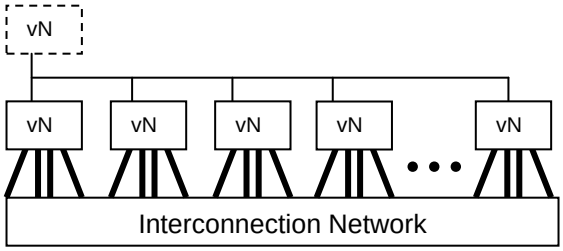

Figure 3. CTA Type Architecture Schematic. The CTA is composed of P von Neumann processors each with its own (program and data) memory, program counter and d-way connection to the communication network; the dashed processor is the "global controller" (often only logical) with a thin connection to all machines; not shown is the key constraint that on-processor memory reference is unit time; off-processor memory reference is λ time.

ogy [16] such as the shuffle-exchange graph or perhaps a 3D torus, but with so many complex issues surrounding interprocessor communication, I was uncertain as to which topology to pick. So, I decided to leave the topology unspecified, and call the machine the *Candidate* Type Architecture, pending a final decision on topology and possibly other features. In retrospect this was an extremely lucky decision, because I eventually decided topology is not any more important to parallel programmers than bus arbitration protocols are to sequential computer programmers. (Also, leaving topology unspecified prevents programmers from embedding topological assumptions in their code, as was criticized under the Antecedents discussion above.) What does matter to programmers, and what was specified with the CTA, is the performance implications of whatever topology the architect ultimately chooses.

The key performance implication of the communication structure is that there is a large latency for communicating between any pair of processors. Though there is also a modest bandwidth limit, it is latency that affects performance—and therefore programming—the most. **Latency**, eventually quantified as **λ**, cannot be explicitly given as a number for a long list of obvious reasons even for a specific topology: Costs grow with *P*; the paths connecting different pairs of processors have different lengths in most topologies; certain routing strategies have no worst-case delivery time [17]; there is network congestion and other conflicts; costs are greatly affected by technology and engineering, etc. But, it is a fact that compared to a local memory reference—taken to be the *unit* of measurement—the latency of a reference to another processor's memory is very large, typically 2-5 orders of magnitude. The λ value has a profound effect on parallel programming languages and the parallel programming enterprise. The number is so significant that the actual details of the communication network don't matter that much: If a program is well written to accommodate λ in this range, then it will be well written for a value 10 times as large or one tenth as large.

There are a few other conditions on the CTA: The interconnection network must be a low-degree, sparse topology if it is to be practical to build machines of interesting sizes. A distinguished processor known as the controller has a narrow connection to all processors, suitable for such things as **eurekas** in which one processor announces to all others. Though largely deprecated in many programmers' minds because processor 0 can act as the controller, various parallel computers have had controllers, and I think it captures an important capability. Notice also that the CTA does not specify whether a parallel machine has hardware support for shared memory. Not specifying shared memory support is consistent with the family of computers being abstracted: some have it and some don't. Overall, the CTA is a simple machine, as shown in Figure 3.

The CTA, despite its designation as a *candidate* type architecture, has remained unchanged since its original specification 20 years ago, and it has been useful. It provided the basis for demonstrating the inherent weakness in shared memory programming abstractions [12]. It is the machine model used for ZPL. It is the machine model message passing programmers use. It is the machine of the LogP model [18]. Further, the type architecture approach—but not the CTA—has been used for sequential computers with FPGA attached processors [19].

6.4 Preparatory Experiments

From 1982, the genesis of Poker, to 1992, the genesis of ZPL, the Blue CHiP project expended considerable effort trying to solve the parallel programming problem. We tried to enhanced Poker. We designed and implemented a language (SPOT) based on the lessons of Poker, but it was unsatisfactory. After creating the CTA, we designed and implemented a language (Orca-C) based on it, but it too was unsatisfactory. Realizing how difficult the problem is, we decided to run a series of experiments (Lin's feasibility studies) to test out the next language (ZPL) before implementing it. This section skips all of the details, focusing only on the work leading up to Lin's dissertation.

One derivative from the Poker work was a classification of **parallel programming levels**. Recall a description from three sections back:

> *Because the CHiP computer is MIMD, Poker provided the ability to write code for each processor; because the CHiP computer is configurable, it provided the ability to program how the processors should be interconnected; because parallel programming was seen as occurring in phases, it provided the ability to orchestrate the whole computation, globally.*

These three aspects of Poker programming were eventually labeled **X**, **Y**, **Z**, where X referred to programming processors, Y referred to programming the communication, and Z referred to programming the overall orchestration of the computation. (These were temporary labels pending better names, which were never chosen.) The classification influenced our thinking about programming, and the top level, Z, was particularly important, being the global view of the computation. Z gave ZPL its name: **Z-level Programming Language**.

Two language efforts contributed to a maturing of our ideas. The first was SPOT [20], an effort to raise the level of abstraction of Poker while remaining tightly bound to a CHiP-like architecture. SPOT introduced array abstractions, but mostly it contributed to the recognition, explained above, that coupling too tightly to a specific machine is too constraining. The second effort, known as Orca-C, was an attempt to extend C to a CTA-based model of computation.[4] The programming was processor-centric, i.e. very much X-level. A prototype was produced, and several application programs, including the Simple benchmark, were written and run on parallel hardware [21]. Being astonished to discover that more than half of the Simple code was devoted to handling boundary conditions, we committed ourselves to attending to the "edges" of a computation in future abstractions. Though Orca-C gave us experience designing and implementing a CTA language and helping users apply it, the low programming level implied we badly needed higher-level abstractions.

Finally, there was the question of whether programming in a language built on the CTA resulted in fast and portable parallel programs. In principle it should, as argued above. But our line of reasoning had been based on only a single success—imperative languages like C and the RAM model.

[4] We expected to generalize Fortran to produce Orca-F, and had the idea been successful, we might be discussing Orca-J!

That seemed like the only way to program, and was so familiar that perhaps we'd missed something. Before creating a new language, we needed experience programming with the CTA as the machine model. So, we ran a series of Orca-C experiments examining whether performance and portability could both be achieved.

These were the experiments reported in Calvin Lin's dissertation, *The Portability of Parallel Programs Across MIMD Computers* [22]. He used Orca-C and the CTA to write several parallel programs and measured their performance across five of the popular parallel architectures of the day: three shared memory (Sequent Symmetry, BBN Butterfly GP1000, BBN Butterfly TC2000), and two distributed memory (Intel iPSC-2, nCUBE/7). Though the space of results was complicated, they suggested to us that it was possible to write one generic program based on the behavior of the CTA and produce reasonable performance consistently across vastly different parallel machines.

A further issue concerned the memory model for a new language. The CTA says, in essence, that parallel programming should be attentive to exploiting locality, but the undifferentiated global shared memory that is the logical extension of the von Neumann model's memory ignores locality, and the convenience of ignoring it is the rationale for including hardware support for shared memory. Lin's results said that respecting locality was good for performance and portability, that is, CTA-based programs run well on machines with or without shared memory hardware. But there is the subtle case of shared memory programs written for shared memory hardware computers. It might be that programs developed assuming global shared memory running on shared memory hardware are superior to generic programs focusing on locality on that same hardware. If so, CTA-focused programming might be penalized on such machines. Lin's experiments suggested CTA-based programs were actually better, but not definitively. Ton Ngo [23, 24] addressed the question more deeply and found that for the cases analyzed the CTA was a better guide to efficient parallel programming than global shared memory for shared memory hardware computers. Though it is almost always possible to find an application that uses a hardware feature so heavily that it beats all comers, the opposite question—can the applications of interest exploit the shared memory hardware better using the shared memory abstraction?—was answered to our satisfaction in the negative.

Thus, with the knowledge that performance and portability could be achieved at once and the knowledge that respecting locality produces good results, we began to design ZPL.

7. Initial Design Decisions

Following the celebratory lunch in October 1992, we launched the language design, which allowed us to apply the results of several years' worth of experiments. It was a great relief to me to finally be designing a language. For several years I'd been attending contractors' meetings, workshops and conferences, and listening to my colleagues present their ideas for ways to program parallel computers. Their proposals, though well intended and creative, were just that: proposals. They were generally unimplemented and unproved. More to the point, they appeared totally oblivious to the issues that were worrying us, making the lack of verifying data even more frustrating.[5] Now, the preparation was behind us. Our formulation of a high-level language would soon emerge.

7.1 Global View versus Shared Memory

With the results of the Lin and Ngo experiments in mind, we adopted a global view of computation. That is, the programmer would "see" the whole data structures and write code to transform entire arrays rather than programming the local operations and data needed by the processors to implement them. The compiler would handle the implementation. By "seeing" arrays globally and transforming them as whole units, programmers would have power and convenience without relying on difficult-to-implement global shared memory.

To be clear, in shared memory, operands can reference any element or subset of elements of the data space, whereas in the global view operations are applied to entire data aggregates, though perhaps selectively on individual elements using regions.

The problem with shared memory is the "random access," that is, the arbitrary relationship between the processor making the access and the physical location of the memory referenced. The Lin/Ngo results emphasized the importance of locality. When a processor makes an access to a random memory location, odds are that it is off-processor, requiring λ time to complete according to the CTA. Global shared memory gives no structure to memory accesses, and therefore, no way[6] to maximize the use of the cheaper local memory references.

Some years earlier we had tested the idea of using APL as a parallel programming language. The results of those experiments [25] showed that although most of APL's facilities admitted efficient parallel implementations, the notable exception was indexing. APL permits arrays to be indexed by arrays, $A[V]$, which provides capabilities more powerful than general permutation. Based on usage counts, indexing is heavily used, and although its full power is rarely needed, it would be very difficult to apply special-case implementations to improve performance. We decided at the time that APL's indexing operator was its Achilles' heel. We wanted more control over array reference.

Another indexing technique is "array slices," `A[2:n-1:2]`, as used in, say Fortran 90 [26] and, therefore, in High Performance Fortran [27]. These references are much more structured with their lower bound, upper bound, stride triple. Slices give a simple specification of the data being referenced for a single operand. An effective compiler needs that information, but it also needs to know the relative distribution between the referenced items for (pairs of) operands because it must assign the work on some

[5] At one point Lin and I tried to engage the community in testing the shared-memory question mentioned above, but it was noticed by only a few researchers [67].

[6] Many, many proposals have been offered over the years to provide scalable, efficient shared memory, both hardware and software. In 1992 none of these seemed credible.

processor(s), and if the data is not local to the processors doing the work, it must be fetched. So, for example, in the Fortran 90 expression

```
… A[1:n] + B[1:n] …                    (1)
```

corresponding elements are added; because of the usual conventions of data placement, this typically would be implemented efficiently without communication. But, the similar expression

```
…  A[1:n] + B[n:1:-1] …                (2)
```

sums elements with opposite indices from the sequence. This likely has serious communication implications, probably requiring most of one operand to reference non-local data.

ZPL solved this problem with regions, which give a set of indices for statement evaluation that applies to all array operands; any variation from common specification is explicit. So, in ZPL the computation of expression (1) would be written

```
[1..n]  …  A + B …
```

since the operands reference corresponding elements. If the relationship is offset, say by one position, as in `A[1:n-1] + B[2:n]`, then ZPL simply uses

```
[1..n-1] … A + B@right …
```

where the `@right` effectively adds 1 to each index, assuming the direction `right` has the value `[1]`. This implies (a small amount of) communication, as explained later in the WYSIWYG performance model. With regions, expression (2) requires first that one of the operands be remapped to new indices. Accordingly, the equivalent ZPL is

```
[1..n]  … A + B#[n-Index1+1] …
```

We explicitly reorder using the remap (`#`) operator and a description of the `n` indices in reverse order (`n-Index1+1`), because the memory model does not permit arbitrary operand references.[7] Programmers must explicitly move the data to give it new indices.

A mistaken criticism of regions asserts that they are only an alternate way to name array slices, that is, reference array elements. But as this discussion shows, the difference is not embodied in naming, though that is important. Rather, the main difference is in controlling the index relationships among an expression's operands: Regions specify one index set per statement that applies to all operands, modulo the effects of array operators like `@`; slices specify one index set per operand.

It may appear that this is a small difference since both approaches achieve the same result, and indeed, it may appear that ZPL's global reference is inferior because it requires special attention. But, as argued below in the discussion of the WYSIWYG performance model, ZPL's policy is to make all nonlocal (λ-cost) data references explicit so programmers can know for certain when their data references are fast or slow and what its big-O communication complexity is.

We eliminated general subscripting because we couldn't figure out how to include it and still generate efficient code relative to the performance constraints of the CTA. Because all operands use the same reference base, regions make possible a variety of simplifications that we wanted:

- The compiler could know the relationship among the operands' indices without depending on complex analysis or the firing of a compiler optimization, and could therefore consistently avoid generating unnecessarily general code
- The common case of applying operations to corresponding array elements allows local, 100% communication-free code
- When operands are not corresponding array elements, programmers specify transformations such as the `@` or `#` operators that embody the communication needed to move the elements "into line"
- "Array-wide" index transformations such as remap often admit optimizations like batching, schedule reuse, etc. that use communication more efficiently
- Understanding the performance model and compiler behavior is much more straightforward and, we recognized later, expressing it to programmers is easier (see WYSIWYG model below)

Finally, one happy convenience of regions is that they avoid the repetitious typing of subscripts that occurs with array slices in the common case when subscripts of each operand correspond.[8]

Unquestionably, adopting the global view rather than shared memory drove almost all of the language design. It made the work fun and very interesting. And as we progressed, we liked what we created. Our programs were clear and succinct and errors were rare. It was easy not to miss subscripts.

7.2 Early Language Decisions

Beginning in March 1993 we taught the base language to the Parallel Programming Environments seminar (CSE590-O) students. (The type architecture and CTA had been covered during fall term.) Soon, we worked as a group to extend the language.

As language design proceeded, the question of syntax arose continually. Syntax, with its ability to elicit heated (verging on violent) debate carries an additional complication in the context of parallel languages. When syntax is familiar, which usually means that it's recognizable from some sequential language, it should have the familiar meaning. A `for`-loop should work as a `for`-loop usually works. Accordingly, parallel constructs will require new or

[7] Index*i* is a compiler generated array of indices for dimension *i*; here it is 1, …, *n*.

[8] Once, to make the point that repeating identical or similar index references using slices is potentially error prone, we found a 3D Red/Black SOR program in Fortran-90, filled with operands of the form `A[loX+1:hiX+1:2, loY-1:hiY-1:2, loZ-1:hiZ-1:2]`, and then purposely messed up one index expression. Our plan was to ask readers which operand is wrong. It was nearly impossible to recognize the correct from the incorrect code, proving the point but making the example almost useless once we had forgotten which program was messed up. Which was which? In frustration, I eventually dumped the example.

distinctive syntax, motivating such structures as `parabegin`/`paraend`. The new semantics have a syntax all their own, but this further raises the barrier to learning the new language. In our design discussions we tended to use a mix of simplified Pascal/Modula-based syntax with C operators and types. It was an advantage, we thought, that Pascal was by then passé, and was thus less likely to be confused with familiar programming syntax. With only a little discussion, we adopted that syntax.

Perhaps the two most frequently remarked upon features of ZPL's syntax are the use of `:=` for assignment and the heavy use of `begin`/`end` rather than curly braces. Of course, the use of `:=` rather than `=` was intended to respect the difference between assignment and equality, but it was a mistake. ZPL's constituency is the scientific programming community. For those programmers such distinctions are unimportant and `:=` is unfamiliar. Programmers switching between ZPL and another language—including the implementers—regularly forget and use the wrong symbol in both cases. It's an unnecessary annoyance.

The grouping keywords `begin`/`end` were adopted with the thought that they were more descriptive and curly braces would be used for some other syntactic purpose in the language. Over the years of development curly braces were used for a number of purposes and then subsequently removed on further thought. In the current definition of the language, curly braces are used only for the unimportant purpose of array initialization. In retrospect, their best use probably would have been for compound statements.

7.3 Early Compiler Decisions

Language design continued during the summer of 1993 in parallel with the initial compiler implementation. To solve the instruction set portability problem we decided to compile ZPL to C, which could then be compiled using the parallel platform's native C compiler to create the object code for the processor elements. Making the interprocessor communication portable requires calls to a library of (potentially) machine-specific transport routines. Though this problem would ultimately be solved by the Ironman Interface [28], it was handled in 1993 by emitting *ad hoc* message passing calls.

To get started quickly, we began with a copy of the Parafrase compiler from the University of Illinois [29]. This gave us instant access to a symbol table, AST and all the support routines required to manipulate them. With a small language and borrowed code, the compiler came up quickly. The compiler's runtime model followed the dictates of the CTA, of course, and benefited greatly from the Orca-C experience.

One measure of success was to demonstrate performance and portability from our high-level language. Though there is a natural tendency in software development to "get something running" and worry about improving performance (or portability) later, we took as a principle that we would deliver performance and portability from the start, Policy #2. We reasoned that this would ensure that the language and compiler would always have those properties: Preserving them would be a continual test on the new features added to the language; any proposed feature harming performance or limiting portability wouldn't be included.

In September, 1993, the basic set of features was reported in "ZPL: An Array Sublanguage" at the Workshop on Languages and Compilers for Parallel Computers (LCPC) [5]. The idea of building an array sublanguage—contained, say, in Orca-C or other general parallel facilities—was crucial initially because it meant that we could focus on the part of parallel computation we understood and could do well, such as the inner loops of Simple and other benchmarks. *We didn't have to solve all the problems at once.* Because we had dumped shared memory and adopted the demanding CTA requirements, there was plenty to do. Over time, the language got more general, but by late fall of 1993—a year after Lin's dissertation defense—we were generating code, and by the next summer, when 1994 LCPC came around, we were demonstrating both performance and portability on small parallel benchmarks [6].

7.4 Other Languages

As noted above, we had already determined some years earlier [25] that APL was an unlikely candidate as a parallel programming language. Nevertheless, because it is such an elegant array language we often considered how our sample computations might be expressed in APL. On hearing that ZPL is an array language, people often asked, "Like APL?" "No," we would say, "ZPL is at the other end of the alphabet." The quip was a short way to emphasize that parallelism had caused us to do most things quite differently. (A related quip was to describe ZPL as the "last word in parallel programming.")

C*, however, was a contemporary array language targeted to parallel computers, and therefore had to deal with similar issues [30]. A key difference concerned the view of a "processor." Being originally designed for the Connection Machine family, C* took the virtual processor view: that is, in considering the parallel execution of a computation programmers should imagine a "processor per point." The physical machine may not have so much parallelism, but by multiplexing, the available parallelism would be applied to implement the idealization. The idea is silent on locality. The CTA implies that a different processor abstraction would be preferable, namely, one in which a set of related data values all reside on the same processor. The CTA emphasizes the benefits of locality since nonlocal references are expensive.

Of course, the processor-per-point (1ppp) view and the multiple points per processor (mppp) view are identical when the number of data values and the number of processors are the same ($n=P$), which never happens in practice. Still, because implementations of 1ppp models bundle m values on each processor for $n = mP$ and multiplex the 1ppp logic on each value, it has been claimed the 1ppp models are equivalent to ZPL's mppp model. They are not. As one contradicting example, consider the array `A` declared over the region `[1..n]` and consider the statement

```
[1..n-1] A := A@east;
```

which shifts the values of `A` one index position to the left, except for the last. The 1ppp model abstracts this operation as a transmission of every value from one virtual processor to its neighbor, while ZPL's mppp model abstracts it as a transmission of a few "edge elements" followed by a local copy of the data values. (See the WYSIWYG model discussion below for more detail.) All communications in both models conceptually take place simultaneously and are therefore equivalent. But the data copy is not captured by the 1ppp model, though it is probably required by the im-

plementation; the data copy is captured by the mppp model. This repositioning of the data may seem like a small detail, especially considering our emphasis on communication, but it can affect the choice of preferred solutions. In the Cannon *v.* SUMMA matrix multiplication comparison [31] described below, repositioning has a significant effect on the analysis. This distinction between the virtual processor and the CTA's multiple points per processor kept us from using C* as a significant source of inspiration.

The High Performance Fortran (HPF) effort was contemporaneous with ZPL, but it had no technical influence. HPF was not a language design,[9] but a sequential-to-parallel compilation project for Fortran 90. Because we were trying to express parallelism and compile for it, that is parallel-to-parallel compilation, we shared few technical problems. And we were building on the CTA and they were not: On the first page of the initial HPF design document [27] the HPF Forum expressed its unwillingness to pick a machine model—"Goal (2) Top performance on MIMD and SIMD computers with non-uniform memory access costs (while not impeding performance on other machines)." At the time I assumed that the Goal had more to do with the need to design by consensus than any neglect of the literature. Either way, attempting to parallelize a sequential language and doing so without targeting the CTA were blunders, preventing us from having much interest in its technical details. But, with most of the research community apparently signed on, we came to see ourselves as competitors because of the divergent approaches. We compared performance whenever that made sense. We were proud to do better nearly always, especially considering the huge resource disparities.

Finally, ZPL is apparently a great name for a language. In the summer of 1995 we received a letter from attorneys for Zebra Technologies Inc. alleging trademark infringement for our use of "ZPL" for a programming language, and threatening legal action. Zebra is a Midwest-based company making devices for UPC barcoding including printers, and they had trademarked their printer control language, Zebra Programming Language, ZPL. We had never heard of them, and we guessed they had never heard of us until MetaCrawler made Web searches possible and they found us via our Web page. Our response pointed out that theirs is a control language for printers and ours was a supercomputer language, and that the potential audiences do not overlap. Nothing ever came of the matter, but every once in a while we get a panic question from some harried barcode printer programmer.

8. Creating ZPL's Language Abstractions

In this section we recap the high points of the initial design discussions and the important decisions that emerged during the first 18 months of project design.

[9] The statement may be unexpected, but consider: Beginning with Fortran 90 and preserving its semantics, HPF focused on optional performance directives. Thus, all existing F-90 programs compute the same result using either an F-90 or HPF compiler, and all new programming in HPF had the same property, implying no change in language, and so no language design.

8.1 Generalizing Regions

Regions as a means of controlling array computation were a new idea, and working out their semantics percolated through the early design discussions. It was clear that scoping was the right mechanism for binding the region to the arrays. That is, the region applying to a *d*-dimensional array is the *d*-dimensional region prefixing the statement or the closest enclosing statement. So in

```
[RegOuter] begin
                Statement 1;
 [RegInner]     Statement 2;
                Statement 3;
           end;
```

d-dimensional arrays in *Statement 2* are controlled by region `[RegInner]`, while *d*-dimensional arrays in statements 1 and 3 are controlled by region `[RegOuter]`. This region is known as the **applicable region** for the *d*-dimensional arrays of the statement. Regions had been strongly motivated by scientific programming style, where it is common to define a problem space of a fixed rank, e.g. 2D fluid flow, and perform all operations on that state space. With large blocks of computation applying to arrays with the same dimensionality, scoped regions were ideal.

Because regions define a set of indices of a specific dimensionality, arrays of different dimensionalities require different regions. ZPL's semantics state that the region defines the elements to be updated in the statement's assignment, so initially it seemed that only one region was needed per statement. (Flooding and partial reduction soon adjusted this view, as explained below.) But, because statements within a scope may require different regions, the original language permitted regions of different ranks to prefix a single statement, usually a grouping statement, as in

```
[1..n] [1..n, 1..m] begin ... end;
```

Though this feature seemed like an obvious necessity, it was rarely if ever used during ZPL's first decade and was eventually removed.

Because the team's *modus operandi* was to use a few specific example computations to explore language facilities and compiler issues, we showed the language to campus users from an early stage as a means of finding useful examples. This engendered a surprising (to me) modification to the language. Specifically, we had defined a region to be a list of index ranges enclosed in brackets, as in the declaration

```
region R = [1..n, 1..n];
```

and required that to control the execution of a statement the name appear in brackets at the start of the line, as in

```
[R] A := ... ;
```

It seemed obvious to us as computer scientists that when a region was specified literally on a statement that it should have double brackets, as in

```
[[1..n, 1..n]] A := ... ;
```

Users were mystified why this should be and were not persuaded by the explanation. We removed the extra brackets. Curiously, computer scientists have since wondered why ZPL doesn't require double brackets for literal regions!

8.2 Directions

The original use of directions as illustrated in Figure 2 was to translate a region to refer to nearby neighbors, as in `A@west`. It is a natural mechanism to refer to a whole array of adjacencies, i.e. the array of western neighbors, and contrasts with single-point alternatives such as `A[.,.-1]`. Generally, directions were a straightforward concept at the semantic level.

The compiler level was another matter. Directions refer to elements some of which will necessarily be off-processor. (A complete description of the runtime model is given in the WYSIWYG section below.) This presents several problems for a compiler. First, communication with the processor storing the data is required, and the compiler must generate code to access that data, as illustrated in Figure 4. Obviously, the direction is used to determine which is the relevant processor to communicate with. Second, in order to maximize the amount of uninterrupted computation performed locally, buffer space—called **fluff** in ZPL and known elsewhere as shadow buffers, ghost regions or halo regions—was included *in position* in the array allocation. Again, it is obvious that the direction indicates where and how much fluff is required. So, for example, with `north = [-1,0]`, the (direction) vector has length 1 and so only a single row can be referenced on a neighboring processor. When the direction is `west2 = [-2,0]` the compiler introduces a double row of fluff.

Unlike regions, however, directions were not originally allowed to be expressed literally, as in `A@[0,-1]`. Furthermore, not allowing literal directions prevented the temptation to allow variables in a direction, as in `A@[x,y]`. The problem with `A@[x,y]` is that we don't know how much fluff to allocate, and we don't know how much communication code to generate. For example, `A@[1,1]` requires fluff on the east, southeast and south edges of the array and three communications (generally) to fill them, while `A@[0,1]` requires only an east column of fluff and one communication. Effectively, ZPL's @-translations were only statically specified constant translations. By knowing the off-processor references exactly, fluff could be allocated at initialization, avoiding array resizing, and the exact communication calls could be generated. Though these restrictions were (not unreasonably) criticized by programming language specialists, they were only rarely a limitation on using the language. One such example was the user whose fast multipole method stencil required 216 neighbor references.

As indicated in Figure 4, the `@` operator will reference whole blocks of data, e.g. columns, and these can be transmitted as a unit. Batching data values is a serious win in parallel communication, because of the overhead to set up the transfer and the advantages of the pipelining of the transmission itself. ZPL gets this batching for free.[10] But we could do even better knowing at compile time what communication is necessary. For example, we move communication operations earlier in the control flow to prefetch data; we also determine when data for several variables is moving between the same processors, allowing the two transmissions to be bundled, as described below. These scheduling adjustments to the default communication lead to significant performance improvements [32] (see Ironman, below). In essence, the language enforced restrictions on `@` to give a fast common case.

Having been conservative with directions originally allowed us to work out fully the compilation issues. Confident that we knew what we were doing, we added literal directions.

8.3 Region Operators

A key question is whether a procedure must specify its own region, or whether it can inherit its region from the calling context. That is, are regions supplied to procedures statically or dynamically? Inheriting was chosen because it is more general and provides a significant practical advantage to programmers in terms of reuse. Since portions of the inherited region will be used in parts of the computation, it is obviously necessary to define regions from other regions.

Specifically, let region `R = [1..n, 1..n]` and direction `west = [0,-1]`. Then to refer to boundary values of the region beyond `R`'s west edge, i.e. a 0th column, is specified `[west of R]` (see Figure 5). The *region operators*—informally known as the *prepositional operators* because they include `of`, `in`, `at`, `by` and `with`—

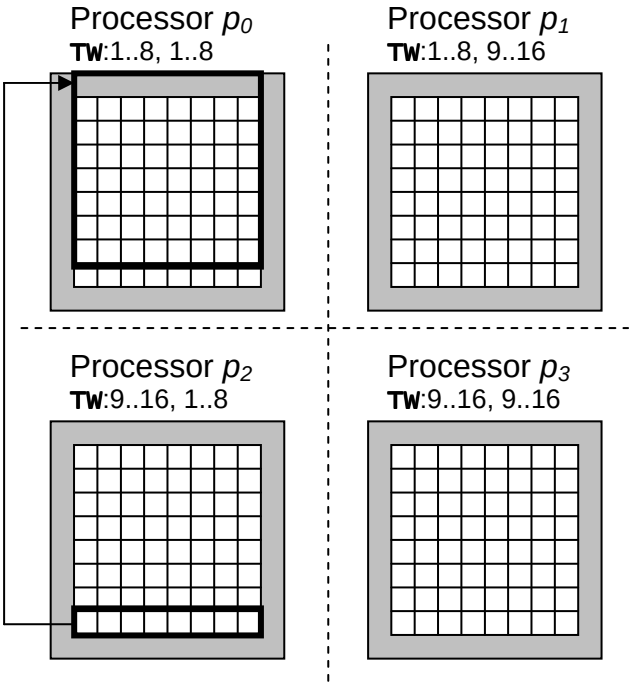

Figure 4. Schematic of the block allocation of array TW from Figure 2, where `n`=16 and the P = 4 processors have a 2 × 2 arrangement. The array is shown as a grid. The fluff buffers, shown shaded, completely surround the local portion of the array because there are @-references in all eight directions. In the diagram, local portion of the (operand) array `TW@^no` for processor p_0 is shown in heavy black; because the north neighbors of the top row wrap to reference the bottom row on processor p_2, communication from p_2 to p_0 is required to fill the fluff buffer.

[10] There was a flutter of compiler research at one point focused on techniques to determine when coalescing communications is possible [68]; raising the semantic level of the language eliminates the problem.

generally take a direction and a region of the same rank and produce a new region, which can be used with other region operators for region expressions, as in `[east in north in R]` for the upper right corner element of a 2D region `R`. The operator `at` translates regions just as `@` translates indices for operands; `by` strides indices, as in `[1..n by 2]`, which selects odd indices in this example. Such operators allow regions to be defined in terms of other regions without the need to know the actual limits of the index range. Similarly, we included the *ditto* region, as in `[ " ]`, meaning to inherit the dynamically enclosing region, and *empty dimensions*, as in `[1..n, ]`, meaning to inherit the index range of that dimension from the enclosing region. Defining the prepositional operators and considering regions in general led to a 4-tuple descriptor for index ranges that probably has applicability beyond ZPL [33].

The `with` operator, as in `[R with mask]`, uses a Boolean array expression (`mask`) rather than a direction with a region as the other prepositional operators do. The variable is a Boolean array of the same rank as `R` specifying which indices to include, and making it similar to the `where` of languages like Fortran 90. Though there are ample cases where masking is useful and sensible, the ZPL team, especially Brad Chamberlain, was always bothered by the fact that the execution of a masked region would be proportional to the size of `R` rather than the number of the true values in the mask. This ultimately motivated him to develop a general sparse array facility; `by` was also a motivation to develop multiregions (see below).

8.4 Reduce and Flood

Since APL, reduce—the application of an associative operator to combine the elements of an array—has been standard in array programming languages. (Scan is also common and is discussed below.) The key property from our language design standpoint is the fact that reduce takes an array as an operand but produces a scalar as a result. This is significant because ZPL has been designed to combine like-ranked arrays to produce another array of that rank. (The reasons concern the runtime representation and the WYSIWYG model, as explained below.) The original ZPL design only included a "full" reduction (*op* <<**)** that combined all elements into a scalar, where *op* is `+`, `*`, `&`, `|`, `min`, `max`. We had plans to include a partial reduction, but had not gotten to that part of the design. Thus, initially, the fact that reduce reduces the rank of its operand was simply treated as an exception to the like-ranked-arrays rule. (Scalars were exceptional anyway, because they are replicated on all processors whereas arrays are distributed.) When it came time to add partial reduction, the issue of an "input shape" and an "output shape" had to be addressed.

Partial Reduce A **partial reduction** applies an associative operator (`+`, `*`, `&`, `|`, `max`, `min`) to reduce a *subset* of its operand's dimensions. The original ZPL solution, motivated by APL's +/[2]*A*, was to specify the reducing dimension(s) by number. This is a satisfactory solution for APL because the **reduced dimensions** are removed. In ZPL they remain and are assigned a specific index to produce a **collapsed dimension**; a dimension is collapsed if the index ranges over a single value, as in `[1..n, 1]`, which is accepted shorthand for `[1..n, 1..1]`.

The consequence of these semantics is that a region is needed to specify the indices of the operand array and another region is needed to specify the indices of the result array. Thus, we write

```
[1..m, 1] B := +<< [1..m, 1..n] A;
```

to add the elements of `A`'s rows and store the results in `B`'s

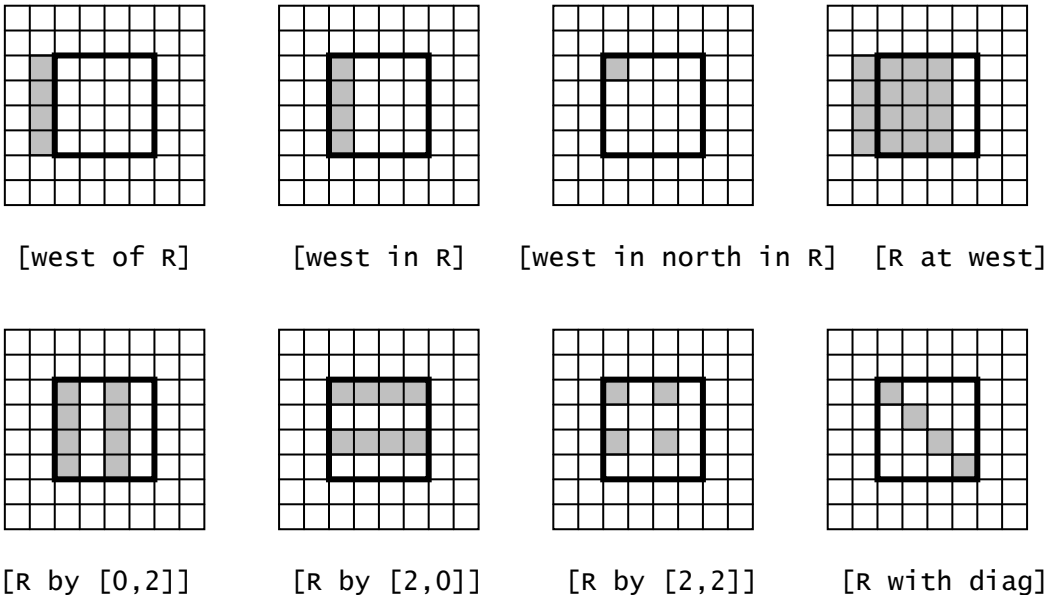

Figure 5. Schematic diagram of various region operators. Referenced elements are shaded; `R` is a region, `west = [0,-1]`, `north = [-1,0]`, `diag` is a Boolean array with trues on the diagonal.

first column. The compiler knows this by comparing the two regions, noting that the first dimension is unchanged and the second is collapsed and so must be combined; the reduction operator (+<<) provides the addition specification.

The reason to depart from standard practice by keeping the reduced dimensions as collapsed dimensions rather than removing them and thereby lowering the dimensionality of the result array relative to the operand array was to manage allocation and communication efficiently. When, for example, a 2D array collapses to a 1D array by a partial reduction, how do we align the result relative to the operand array? Both the compiler writer and the user need to know. Normally, the result will interact in subsequent computations with other arrays having the operand shape. So aligning our 1D result with 2D arrays is rational. But how?

Aligning the result with columns is possible; aligning it with rows is possible; treating it as a 1D array is also possible, though because ZPL uses different allocations for 1D and 2D arrays (in its default allocations), the operand and result arrays would misalign. The decision matters because in addition to the time required to compute the partial reduction, each choice entails different amounts of communication to move it to its allocated position, ranging from none, to an additional transpose, to the very expensive general remap.

By adopting collapsed dimensions, however, the result aligns with the operand arrays in the most efficient and natural way: it aligns with the "remaining" shape; further, the user specifies (by the index(es)) exactly how. Most importantly, collapsed dimensions allow us to bound the communications requirements to O(*log* P_i) where P_i is the number of processors spanning the reduced dimension *i*. Because of such analysis, the powerful partial reduction operator is also an efficient operator. (This type of analysis—worry about communication, worry about how the user controls the computation—characterized most of our design discussions about most language features.)

This "two region" solution is elegant, but we didn't figure it out immediately.

Creating Flood Indeed, we came to understand partial reduction by trying to implement its inverse, flood. **Flood** replicates elements to fill array dimensions. Flood is essential to producing efficient computations when arrays of different rank are combined, e.g. when all columns of an array (2D) are multiplied by some other column (1D). In the case of flood the need for two regions is probably more obvious because there must be a region to specify the source data and another region to specify the target range of indices for the result. Thus, to fill A with copies of B's first column, write

```
[1..m, 1..n] A := >> [1..m, 1] B;
```

The duality with partial reduction is obvious.

The use of two regions—one on the statement to control the assignment and the overall behavior of the computation, and one on the source operand specifying its indices—was a difficult concept to develop. Flooding was a new operation that I more or less sprung on the team at a ZPL retreat in 1994; it was introduced as a new operator and engendered spirited debate. The duality with partial reduction was not immediately recognized because partial reduction still had its "APL form." Flooding was not adopted at the retreat, and Brad Chamberlain and I continued to kick it around for a couple of weeks more after the rest of the team had tired of the debate. In our conversations the solution emerged, the duality emerged, the elimination of partial reduction's dimensions-in-brackets definition was adopted, and the idea of recognizing which dimensions participate in the operations by comparing the two regions and finding which dimensions are collapsed/expanded was invented.

This experience with flood represented in my view the first time we had attacked a problem "caused" by trying to build a global view language for the CTA, and had produced a creative solution that advanced the state of the art in unexpected ways. Regions, directions, etc. were new ideas, too, but they "fell out" of our approach. Flooding posed significant language and compiler challenges, and we'd solved them with creativity and hard work. Flooding, which motivated a new algorithmic technique (problem space promotion [34], discussed below), continues to be a source of satisfaction.

Flood Dimensions Shortly after inventing the two region solution, we invented the concept of the flood dimension. To appreciate its importance, consider why parallel programmers are interested in the flooding operation in the first place. Imagine wanting to scale the values in the rows of a 2D array to the range [-1,1]. This is accomplished if for each row the largest magnitude is found and all elements of the row divided by that value. ZPL programmers write

```
[1..m,1..n] A := A /
        (>> [1..m,1] max<< [1..m,1..n] abs(A));
```

which says, use as the denominator the flooded values of the maximum reduction of each row. That is, find the maxima, replicate those values across their rows, and perform an element-wise division of the replicated values into the row. The statement allows the ZPL compiler

- to treat the row maxima as a column and transmit it to the processors as a unit, achieving the benefits of batched communication rather than repeated transmission of single values,
- to multicast rather than redundantly transmit point-to-point, and
- to cache, since each processor will store its section of the column and repeatedly use the values for the multiple references local to it.

In effect, the programmer's use of flood described the computation in a way that admitted extremely efficient code.

The odd feature of the last statement is the use of "1" in the flood,

```
(>> [1..m,1] max<< [1..m,1..n] abs(A))
          ^
```

specifying that the result of the max reduction is to be aligned with column 1 before flooding. Why column 1? In fact there is no reason to assign the result of the reduction

to any specific column.[11] What we want to say is that a dimension is being used for flooded data, and so no specific index position is needed—or wanted. This is the flood dimension concept, and is expressed by an asterisk in the dimension position, as in `[1..m,*]`. In such regions any column index conforms with the flood dimension.

The flood dimension extends regions with an intellectually clean abstraction, but there is a greater advantage than that. The compiler can implement a flood dimension extremely efficiently by allocating a single copy of that portion of the replicated data needed by a processor. So, for example, if `CellCountPerRow` is declared as the flood array `[1..n, *]` for the allocation shown in Figure 4, the assignment

```
[1..n, *] CellCountPerRow := +<< [1..n, 1..n] TW;
```

adds up the number of cells in each row and stores the relevant part of `CellCountPerRow` on each processor. That is, referring to Figure 4, both processors p_0 and p_1 would store only the first eight elements of `CellCountPerRow`, and p_2 and p_3 would store the last eight. These "logically populated" arrays are an extremely efficient type of storage, and do not require a reduction or flood to assign them. The next section gives an elegant example.

8.5 Flooding the SUMMA Algorithm

The power and beauty of flooding became clear in 1995 when we applied it to matrix multiplication. We had written several row-times-column matrix product programs in ZPL to test out language features, when Lin reported a conversation with Robert van de Geijn of UT Austin about his new algorithm with Jerrel Watts, which they called SUMMA—scalable, universal matrix multiplication algorithm [35]. Their idea was to switch from computing dot-products as a basic unit of a parallel algorithm to computing successive terms of all dot-products at once. They claimed that it was the best machine independent matrix multiplication algorithm known.

To see the key idea of SUMMA and why flooding is so fundamental to it, recall that in the computation $C = AB$ for 3x3 matrices A and B, the result is

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} + A_{1,3}B_{3,1} \quad C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} + A_{1,3}B_{3,2}$$
$$C_{1,3} = A_{1,1}B_{1,3} + A_{1,2}B_{2,3} + A_{1,3}B_{3,3}$$
$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} + A_{2,3}B_{3,1} \quad C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2} + A_{2,3}B_{3,2}$$
$$C_{2,2} = A_{2,1}B_{1,3} + A_{2,2}B_{2,3} + A_{2,3}B_{3,3}$$
$$C_{3,1} = A_{3,1}B_{1,1} + A_{3,2}B_{2,1} + A_{3,3}B_{3,1} \quad C_{3,2} = A_{3,1}B_{1,2} + A_{3,2}B_{2,2} + A_{3,3}B_{3,2}$$
$$C_{3,2} = A_{3,1}B_{1,3} + A_{3,2}B_{2,3} + A_{3,3}B_{3,3}$$

Notice that the first term of all of these equations can be computed by replicating the first *column* of A across a 3x3 array, and replicating the first *row* of B down a 3x3 array, that is, flooding A's first column and B's first row, and then multiplying corresponding elements; the second term results from replicating A's second column and B's second row and multiplying, and similarly for the third term.

The ZPL program for SUMMA (Figure 6) applies flooding to compute these terms. It begins with declarations, including `Col` and `Row`, flood dimension arrays. These two arrays can be assigned a column and a row, respectively, which is replicated in the `*` dimension. After initializing `C`, a `for`-loop steps through the columns and rows of the common dimension, flooding each and then accumulating their element-wise product, implementing SUMMA.

```
var  A : [1..m,1..n] double; Left operand
     B : [1..n,1..p] double; Right operand
     C : [1..m,1..p] double; Result array
   Col : [1..m,*]    double; Col flood array
   Row : [*,1..p]    double; Row flood array
…
[1..m,1..p]  C := 0.0;        Initialize C
        for k := 1 to n do
[1..m,*]   Col := >>[ ,k] A; Flood kth col of A
[*,1..p]   Row := >>[k, ] B; Flood kth row of B
[1..m,1..p]  C += Col*Row;    Multiply & accumulate
         end;
```

Figure 6. ZPL matrix multiplication program using the SUMMA algorithm.

We were delighted by the van de Geijn and Watts algorithm, because it was not only the best machine-independent parallel matrix product algorithm according to their data, it was the shortest matrix multiplication program in ZPL. (The three statements in the `for`-loop can be combined,

```
[1..m,1..p] for k := 1 to n do
                C += (>>[ ,k] A )*( >>[k, ] B);
            end;
```

eliminating the need for the `Col` and `Row` arrays.) When the best algorithm is also the easiest to write, the language's abstractions match the problem domain well.[12]

8.6 Epic Battle: Left-hand Side @s

At the 1994 ZPL retreat mentioned above, the issues surrounding flood were not resolved, first because they were deep and complex, requiring thought and reflection, and second because we spent most of our time arguing over left-hand side @s. All (early) project members agree that this was our most contentious subject, and the magnitude of the disagreement has now grown to mythic proportions. Here's what was at issue.

ZPL uses @-references to specify operands in expressions, as in

```
[2..n-1,2..n-1] B := (A@north + A@east
                    + A@west + A@south)/4;
```

where directions translate the region's indices to a new region to reference the operand. Programming language design principles encourage the consistent use of constructs, and so because @-translations are allowed on the right-hand side of an assignment, they should be allowed

[11] ZPL recognizes the reduction-followed-by-flood as an idiom and permits >> *op* << with no dimension specified, but that doesn't solve the general problem.

[12] The original SUMMA is blocked. ZPL's is not, and blocking complicates the ZPL significantly.

on the left-hand side as well. The semantics of left-hand side @ are intuitive and are natural for programmers, as in

```
[2..n,2..n] B@nw := A@north & A & A@west;
```

which flows information into the corner of an array when used iteratively, for example, in a connected components algorithm [36]. Of course, left-hand side @s are not required, since the programmer could have written

```
[1..n-1,1..n-1] B := A@south & A@se @ A@east;
```

But he or she may think of the computation from the other point of view. Another principle of language design says to avoid placing barriers in the programmer's way.

The compiler writer's viewpoint noted that ZPL's semantics use the region to specify which indices are updated in the assignment. Processors are responsible for updating the values of the region that they store. By using a left-hand side @ the indices that a processor is to update are offset. Which processor is responsible for updating which elements—only those it owns or those it has the data for? Should the value for a location be computed on another processor and then shipped to the processor that stores it; or should every processor simply handle the values it stores? The presence of such a construct in the language complicates the compiler and has serious optimization implications.

Like many issues that engender epic design battles, left-hand side @s are an extremely small matter. How and why they excited so much controversy is unclear. It may simply be that the particular circumstances of the situation provided the accelerant to heat up the discussion, and that on another day the issue would have hardly raised comment.[13] Whatever the reason, left-hand side @s were ZPL's biggest controversy. In this case, however, everyone won: The language includes left-hand side @s and the compiler at that time automatically rewrote any use of left-hand side @s to remove them, leaving the remainder of the compiler unaffected. In the years since a direct implementation has been developed.

8.7 A Bad Idea From Good Intentions

As mentioned above, the Simple benchmark written in Orca-C devoted more than half its lines to computing boundary values [21]. This represents a huge amount of work for programmers. A high-level language should help. ZPL's regions, directions and wrap-@ constructs do help significantly, and almost eliminate concern about boundaries. But we added one more feature, automatic allocation of boundary memory. Automatic allocation was a mistake.

Recall that the `of` prepositional operator refers to a region beyond the base region, that is,

```
[west of R] A := 0.0;    -- Initialize west boundary
```

Programmers were confused by this unfamiliar idea, not knowing when it did or didn't apply to an `of`-region. Defining when auto-allocation made sense was complicated because it applied only to `of`-regions extending regions used in array allocation, though `of`-regions are used in many other circumstances. This produced a hard-to-follow set of definitional conditions. Even when users understood them, auto-allocation seemed not to be used. Basically, it violates what should be a fundamental principle of language design: *Make the operations clear and direct with no subtle side effects.* Automatic allocation was a side effect to an array expression. It was deprecated and eventually removed.

8.8 Working With Users

Throughout the ZPL development we had the pleasure of working with a group of very patient and thoughtful users, mostly from UW. Often their comments and insights were amazing, and they definitely had impact on the final form of the language.

Adding Features Direct complaints about the language were the least likely input from users, though that may have reflected their discretion. The most direct comments concerned facilities that ZPL didn't have. "ZPL needs a complex data type," was a common sentiment among users and, like good computer scientists, we replied that users could implement their own complex arithmetic using existing facilities. But such responses revealed the difference between tool builders and tool users, and being the builders, we added `complex`. Quad-precision was another example. Also, Donna Calhoun, along with others from UW Applied Math, strongly encouraged our plans to add sparse arrays; when Brad Chamberlain implemented them, she was the first user.

Shattered control flow is an example of a language construct that emerged by working with a user, George Turkiyyah of UW Civil Engineering. Shattered control flow refers to the semantics of using an array operand in a control flow expression. Normally, when a control statement predicate is a scalar, say `n`, ZPL executes one statement at a time, as in

```
if n < 10000 then baseCase(n, A);
```

but if the control predicate evaluates to an array of values, as it would if `A` is an array in

```
[R] if A < 0 then A := -A;
```

the control flow is said to shatter into a set of independent threads evaluating the statement(s) in parallel for each index in `R`. In the example, the conditional is applied separately for each index in the region; for those elements less than 0, the `then`-clause is executed; for the others it is skipped. Turkiyyah and I were having coffee, discussing how his student might solve a problem in ZPL and making notes on a napkin. In the discussion I had used several such constructs without initially realizing that they didn't actually match current ZPL semantics, which didn't allow non-scalar control expressions. But they were what was required for his problem, and knowing a solution [30, 37], I'd used it. When I thought about it later, it was clear that this was an elegant way to include a weak `parabegin`/`paraend` construct in ZPL's semantics. By the time the student's program was written, shattered control flow was in the language.

The Power of Abstraction The best user experiences were of the "satisfied customer" type, and two stand out as more notable for their elegance. The first was a graduate

[13] Perhaps not. An early project member reviewing this manuscript complained that my characterization of it as a "small matter" revealed my biases, and then went on to emphasize its depth by reiterating all of the "other side's" arguments … a dozen years later!

student in biostatistics, Greg Warnes, who came by one day with a statistics book containing a figure with a half dozen equations describing his computation. He said that his sequential C program required 77 lines to implement the equations, while his ZPL program required only 11 lines. Moreover, he emphasized, the ZPL was effectively a direct translation from the math syntax of the book to the syntax of ZPL. He may have been even more delighted with his observation that day than we were.

In the second case, I'd given a lecture about ZPL to an audience in Marine Sciences on Thursday, and by the next Monday one of the faculty emailed a ZPL program for a shallow water model of tidal flow. Reading over the several pages of the program, I was astonished to notice that he'd used several named regions, all containing the same set of indices. Most programs use multiple regions, of course, but one name per index set is sufficient. What the shallow-water program revealed was the programmer's use of regions as abstractions in the computation—different region names referred to different phenomena. The appropriate assignment of indices made them work, and the fact that they might name the same set of indices was irrelevant.

Tool Developers It was a surprise to realize that a programming language is an input to other projects. At the University of Massachusetts, Lowell, ZPL was revised for use in a signal processing system. At the University of Oregon, ZPL was instrumented by Jan Cuny's group to study program analysis and debugging. At Los Alamos National Laboratory, ZPL was customized to support automatic check-pointing, an important concern for long running programs and a challenge for decentralized parallel machines.

David Abramson and Greg Watson of Monash University in Melbourne, Australia applied the interesting concept of relative debugging to ZPL programs [38]. **Relative debugging** uses a working program to specify correct behavior for finding errors in a buggy program. Relative debugging is a powerful tool in parallel programming because parallel programs often implement the computation of a working sequential program. Errors in the parallel program execution can be tracked down *relative* to the sequential program's execution by running both simultaneously and asserting where and how their behaviors should match. The tool is very convenient.

The demonstrations of relative debugging were impressive. Watson found a bug in the ZPL version of Simple, one in the sequential C version and one in both! The error in ZPL was an extra delta term in computing the heat equation; this problem had been in every parallel version of Simple we had ever written. (The only effect fixing the bug would have had on published timing results would have been to improve them trivially.) The bug in the C program was to leave out the last column in one array computation. Watson said, "Whoever wrote the ZPL code obviously noticed this and corrected for it (or the ZPL syntax corrected it automatically)." But such "off-by-1" errors are hard to make once the named regions are set up correctly. Finally, the programs disagreed on how they computed the error term controlling convergence, and in this regard they were both wrong! Referring to the original Simple paper [39] allowed Watson to fix both.

9. WYSIWYG Performance Model

Though the CTA was always the underlying "target" computer for ZPL, the idea of explicitly handing programmers a performance model was not originally envisioned. In retrospect, it may be ZPL's best contribution. Notice that as the compiler writers, we had had the performance model in our heads the whole time. The achievement was that we recognized that it could be conveyed to programmers.

9.1 The Idea

How the idea emerged has been lost, but I believe it was an outgrowth of a seminar, *Scientific Computation in ZPL*, which we ran every winter quarter beginning in 1996. These seminars were a chance for us to present parallel programming to scientists and engineers "properly," i.e. in ZPL. The format was for project members to teach ZPL and for students to write a program of personal interest to be presented at the end of the term. The students, unlike researchers and seasoned users who grabbed our software off the Web, needed to be told how to assess the quality of a ZPL program. Conveying the execution environment motivated our interest in a model.

The champion of WYSIWYG was E Chris Lewis, who had joined the project a year after the initial team and saw ZPL with fresh eyes. He spoke of a concept "manifest cost" for an operation, which seemed pretty obvious to the implementers. But he persisted and as we worked the idea over, it became clear how valuable specifying costs could be for users. The model was documented in Chapter 8 of *A Programmer's Guide to ZPL* [40], which was the main language documentation at the time and was eventually published in 1999.

The **What You See Is What You Get** (**WYSIWYG**) performance model is a CTA-based description of the performance behavior of ZPL language constructs expressed with syntactically visible cues [31]. The WYSIWYG model specifies what the parallelism will be and how the scalar computation and communication will be performed, telling programmers how ZPL runs on the CTA and by extension on the machines the CTA models. This performance specification cannot be given in terms of time or instruction counts because the programs are portable and every platform is different; rather, it gives relative performance.

To illustrate how it works, consider the two assignment statements from the Life program (Figure 2):

```
NN := TW@^nw + TW@^no + TW@^ne
    + TW@^w           + TW@^e
    + TW@^sw + TW@^so + TW@^se;

TW := (TW & NN = 2) | (NN = 3);
```

How do they perform? They look similar, and the semantics say that element-wise arithmetic operations on arrays are performed atomically in a single step. But, we know that "step" will be implemented using many scalar operations on a CTA. How will this be done?

Programmers know from the WYSIWYG model that although both statements will be performed fully in parallel, and that the n^2 scalar operations implementing each of them are only negligibly different in time cost (seven binary ops versus four), the two statements actually have noticeably different performance. This is because the assignment to

NN requires point-to-point communication, thereby incurring at least λ additional cost, while the assignment to TW uses no communication at all.

How is this known? *Every use of @ entails point-to-point communication* according to the WYSIWYG model, *and the absence of @ (and other designated array operators) guarantees the absence of communication.* By looking for these cues, programmers know the characteristics of the statements: The first statement requires communication, the second does not. Further, the amount of communication—total number of values transferred and where—is also described to the programmer based on other program characteristics. Knowing when and how much communication a statement requires is crucial, because although it cannot be avoided, alternative solutions can use vastly different amounts of it. ZPL programmers can analyze performance as they write their code, judge the alternatives accurately and pick the best algorithm. This is the key to writing efficient parallel programs.

Notice that this ability to know when and what communication is incurred is not a property of High Performance Fortran. Recall our earlier discussion of array slices, in which we asserted that it would be reasonable for expression (1) above

```
… A[1:n] + B[1:n] …
```

not to require communication, but we cannot for several reasons be sure. By contrast, we can be sure that the equivalent ZPL expression

```
[1..n]    … A + B …
```

does not.

9.2 Specifics of the Model

To describe the WYSIWYG model, we start on the command line as we invoke a compiled ZPL program. The user specifies along with the computation's configuration values the number of processors available and their arrangement; for our running example, P=4, in 2 rows of 2. This is the processor grid, and though in the early years it was thus set for the entire computation, it has since come under programmer control. See Figure 7 for the allocation of a 16×16 problem on the four processors.

During initialization, the regions, and by extension the arrays, are allocated in blocks (by default) to processors, as shown in Figure 7. A key invariant of this allocation is that every element with index [i, j, …, k] of every array is always allocated to the same processor (though programmers can override it). So, NN's allocation will match that of TW. This allows all "element-wise" computations to be performed entirely locally. Thus, seeing

```
TW := (TW & NN = 2) | (NN = 3);
```

users know that each processor computes its portion of TW using only the locally stored values of NN and TW. Programmers know they are getting the best performance the CTA has to offer, and the available parallelism (P=4) is fully utilized. Such element-wise computation is the sweet spot of parallel computation.

As mentioned, @-communication requires nonlocal data reference. Thus, prior to executing

```
NN := TW@^nw + TW@^no + TW@^ne
   +  TW@^w           + TW@^e
   +  TW@^sw + TW@^so + TW@^se;
```

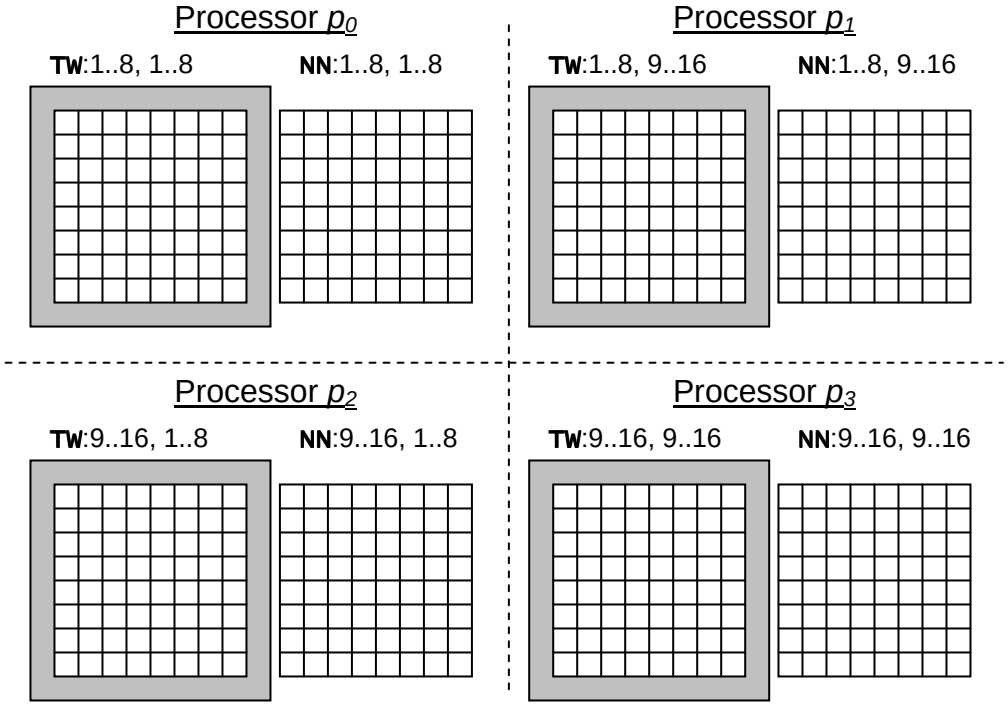

Figure 7. Allocation of the Life program to a 2×2 (logical) processor grid. Notice that array elements having the same index are allocated to the same processors. TW has fluff allocated because some of its references are modified by @-translations; NN does not require fluff.

Table 1. **Sample WYSIWYG Model Information**. Work is the amount of computation for the operator measured as implemented in C; *P* is number of processors. The model is more refined than suggested here.

Syntactic Cue	Example	Parallelism (*P*)	Communication Cost	Remarks
`[R]` *array ops*	`[R]` ... `A+B` ...	full; work/*P*		
`@` *array transl.*	... `A@east` ...		1 point-to-point	xmit "surface" only
`<<` *reduction*	... `+<<A` ...	work/*P* + log *P*	2log *P* point-to-point	fan-in/out trees
`<<` *partial red*	… `+<<[ ] A` …	work/*P* + log *P*	log *P* point-to-point	
`||` *scan*	… `+||` …	work/*P* + log *P*	2log *P* point-to-point	parallel prefix trees
`>>` *flood*	… `>> [ ] A`…		multicast in dimension	data not replicated
`#` *remap*	... `A#[I1,I2]` ...		2 all-to-all, potentially	general data reorg.

communication is required to fill the fluff buffers with values from the appropriate neighbors. For example, to reference `TW@^no`, the array of north neighbors illustrated in Figure 4, the fluff along the top of the local allocation must contain values from the appropriate neighbor. Figure 8 shows the source of all fluff values required by processor p_0. Notice from the figure that `@`-communication does not transmit all values, only those on the "surface" of the allocation—edges and corners—and that *d* direction references will typically require communication with *d* other processors. (This example looks especially busy because the instance is so small.) Once a processor has the necessary data, it executes at a speed comparable to the statement without any `@`-communication.

To conclude the analysis of the Life program, notice that the iteration uses an OR-reduce in the loop control, `!(|<<TW)`. Reduce is another operation that according to the WYSIWYG model entails communication to compute a global result from values allocated over the entire processor grid. The local part of the reduction performs an OR of the locally stored values. The nonlocal part percolates these intermediate values up a combining tree to produce a final result. Because that global result controls the loops executing on each processor, it is broadcast to all processors. These two operations—combine and broadcast—are specified as taking log *P* communication steps in the WYSIWYG model, because the combining tree can be implemented on any CTA machine. However, some machines may have hardware for the combining operation as the CM-5 did; alternatively, all processors could fan into a single one if the hardware favored that type of implementation.

The WYSIWYG performance model specifies the characteristics of ZPL's constructs. Table 1 shows a simplified specification.

The specification of the table is simplified in the sense that true program behavior interacts with allocation, which programmers are fully aware of and control. For example, the programmer *might* have allocated whole columns of an array to the processors, implying there will be no communication in the north/south direction for `@` or >> operations. So in the case, when analyzing, say, the SUMMA algorithm in this context, the flood of `A`'s columns requires a broadcast to all processors, but the flood of `B`'s rows requires no communication at all. Programmers know all of this information (they specify it), and so can accurately estimate how their programs will behave.

Because the remap (`#`) operator can restructure an array into virtually any form, it requires two all-to-all communications, one to set up where the data moves among the processors and one to transmit the data. Remap is (typically) the most expensive operator and the target of aggressive optimizations [41]. Returning to the example above where the elements of a vector were added to their opposite in sequential order,

```
[1..n]  … A + B#[n-Index1+1]  …
```

the presence of remap implies by the WYSIWYG model that data motion is necessary to reorder the values relative to their indices. Though two all-to-all communications are specified, the complier-generated specification for "reverse indices" (`n-Index1+1`) allows the first of the two to be optimized away; the remaining communication is still potentially expensive. Of course, the actual addition is fully parallel.

The key point is that the WYSIWYG performance properties are a *contract between the compiler and programmers*. They can depend on it. Implementations may be better because of special hardware or aggressive optimizations by the compiler, but they will not be worse. Any surprises in the implementation will be good ones.

9.3 The Contract

To illustrate how the contract between the compiler and the programmer can lead to better programming, return to the topic of slices versus regions, and consider an operation on index sequences of unknown origin, namely

```
... A[a:b] + B[c:d] ...
```

Such a situation might apply where there are "leading values" that must be ignored (in `B`). Because the values of `a`, `b`, `c` and `d` are unknown, programmers in Fortran 90 will likely use this general solution.

If absolutely nothing is known about the range limits, the ZPL programmer will write

```
[a..b]  ... A + B#[c..d] ...
```

But using the WYSIWYG model, and knowing that remap is expensive, the programmer might further consider whether there are special cases to exploit. If `a=c` and `b=d`, then the ZPL

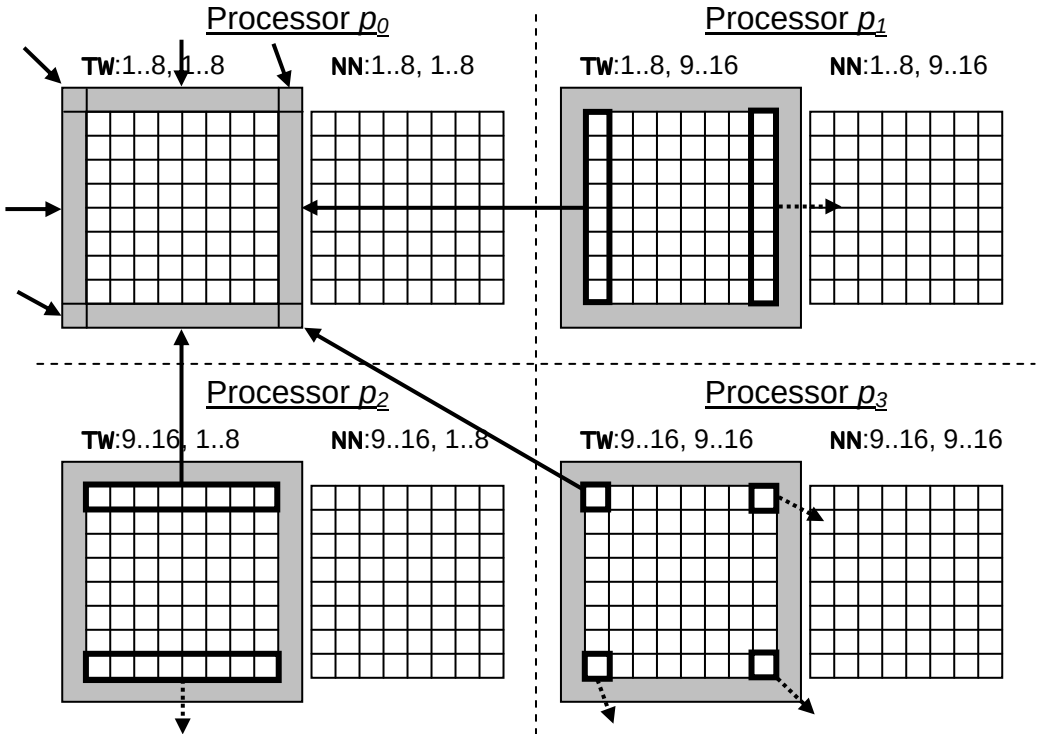

Figure 8. Communication required to load the fluff for `TW` on processor p_0 to support wrap `@`-translations. Internal array adjacencies are shown by direct arrows from the source values to the appropriate fluff; in all other cases, the reference wraps around the array edge, and the transmission is shown leaving on a dotted arrow and arriving at a solid arrow *of the same angle*.

```
[a..b]  ... A + B ...
```

will be an entirely local computation with full parallelism and a great enough "win" that it is worth applying, even occasionally. Further, if the limits are occasionally only "off by 1", that is, if `c=a+1` and `d=b+1`, then the ZPL statement

```
[a..b]  ... A + B@right ...
```

where `right = [1]` is equivalent and also more efficient, despite requiring some communication.

The ZPL compiler generates the exact code required in each case, guaranteed, and because the λ penalty is so large for general communication, it may pay to special-case the situation even if the cases apply in only a fraction of the situations.

9.4 Verifying The Predictions

Creating the WYSIWYG model was easy enough, because it simply describes how the compiler maps source code to the run-time environment coupled with the properties of the CTA. However, its predictive qualities had to be demonstrated. We chose to illustrate the analysis on two well known and highly regarded parallel matrix multiplication algorithms, the SUMMA algorithm [35] and Cannon's algorithm [42], because they take quite different approaches. (Though we had programmed them often, we hadn't ever compared them.)

The methodology was to write the "obvious" programs for the two algorithms and then analyze them according to the dictates of the WYSIWYG model; see Figure 6 for SUMMA and Figure 9 for Cannon. Both computation and communication figure into the analysis.

The analysis divides into two parts: setup and iteration. Only Cannon's algorithm requires setup: The argument arrays are restructured by incremental rotation to align the elements for the iteration phase. This would seem to handicap Cannon and favor SUMMA, but the situation is not that simple. Ignoring the actual product computation, which is the same for both algorithms, SUMMA's iteration is more complex because it uses flood rather than `@`-communication. Floods are implemented with multicast communication, which is generally more expensive than the `@`-communication. On the other hand, flood arrays have much better caching behavior and the rotation operations of Cannon's require considerable in-place data motion. So, which is better?

We predicted that SUMMA would be better based on the WYSIWYG model, though completing the analysis here requires a bit more detail about the model than is appropriate [31]. Then we tested the two algorithms on various platforms and confirmed that SUMMA is the consistent winner. Not only was this a satisfying result, but with it we began to realize its value in achieving our goals. *One way to make a language and compiler look good is to empower programmers to write intelligent programs!*

Once we discovered the importance of the WYSIWYG performance model, all subsequent language design was influenced by the goal of keeping the model clear and easy to apply.

```
var  A : [1..m,1..n] double;   Left operand
     B : [1..n,1..p] double;   Right operand
     C : [1..m,1..p] double;   Result array
direction  right = [0,1];      Ref higher row elements
           below = [1,0];      Ref lower col elements
        ...
  for i := 2 to m do           Skew A, cyclically rotate
[i..m,1..n]  A := A@^right;      successive rows
  end;
  for i := 2 to p do           Skew B, cyclically rotate
[1..n, i..p] B := B@^below;      successive columns
  end;
[1..m, 1..p] C := 0.0;         Initialize C
  for i := 1 to n do           For common dimension
[1..m, 1..p] C += A*B;         Accumulate product terms
[1..m, 1..n] A := A@^right;    Rotate A
[1..n, 1..p] B := B@^below;    Rotate B
  end;
```

Figure 9. Cannon's matrix multiplication algorithm in ZPL. To begin, the argument arrays are skewed so rows of A are cyclically offset one index position to the right of the row above, and the columns of B are cyclically offset one index position below the column to its left; in the body both argument arrays are rotated one position in the same directions to realign the elements.

10. Compiler Milestones

The ZPL compiler uses a custom scanner with YACC as its front end, and we initially relied on symbol table and AST facilities from the Parafrase compiler. (As the compiler matured the Parafrase code was steadily eliminated in favor of custom ZPL features; it is not known how much Parafrase code remains, if any.) As new language features were added to ZPL, the corresponding new compiler technology was usually implemented by additional passes over the AST. A final code generation pass emitted ANSI C to compete the compilation process.[14] The overall structure is shown in Figure 10.

Several aspects of the compiler are significant.

10.1 Ironman – The Compiler Writer's Communication Interface

A compiler that seeks to generate machine-independent parallel code must decide how to handle interprocessor communication, which is as particular to a computer as its instruction set. Standard compiler theory suggests solving the problem by targeting machine independent libraries or producing a separate backend for each platform. The latter is time consuming (and platforms come and go with astonishing rapidity), so writers of parallel language compilers target the lowest common denominator: the machine independent message-passing library. All parallel computer communication can be reduced to message passing. Two

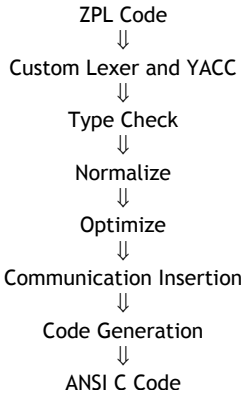

Figure 10. Principal components of the ZPL compiler.

libraries are in wide use: Parallel Virtual Machine (PVM) [42] and Message Passing Interface (MPI) [43]. But, in our view compiling to the message passing level introduces too much "distance" between compiled code and the computer.

Specifically, message passing is a communication protocol with a variety of forms, ranging from a heavyweight synchronous form with multiple copying steps, to lighter-weight asynchronous forms. Without delving into details, all schemes have a sender that initiates the communication (`send`) and must wait until the data is "gone," and a receiver that explicitly receives it (`recv`) when it "arrives" to complete the transmission; further, when noncontiguous data addresses are sent—think of a column of data for an `A@west` transmission in a row-major-order allocation—data must be marshaled into a message, i.e. brought together into a separate buffer. But there are other kinds of communication: In shared-memory computers data is transmitted by the standard load/store instructions; other machines have "one-sided" get/put communication. Both schemes are extremely lightweight compared to message passing, e.g. neither requires marshalling. Message passing is too thick a layer as the one-size-fits-all solution.

The ZPL commitment to targeting all parallel computers made us aware of these differences early. We had originally generated *ad hoc* message passing calls, but then realized that we needed a way to generate generic code that could map to the idiosyncrasies of any target machine. Brad Chamberlain with Sung-Eun Choi developed the Ironman interface, a compiler-friendly set of communication calls that were mechanism independent [44]. Unlike message-passing commands, the Ironman operations are designed for compilers rather than people.

Ironman conceptualizes interprocessor communication as an assignment from the source to the target memory, and the four Ironman calls are the def/use protocol bounding the interval of transmission (see Figure 11):

- `destination_ready()` marks the place where the buffer space (fluff) that will be written into has had its last use
- `source_ready()` marks the place where the data to be transmitted has been defined
- `destination_needed()` marks the place where computations involving the transmitted data are about to be performed – impending use

[14] One of our first and most patient users was an MIT physicist, Alexander Wagner, who used ZPL for electromagnetic modeling [69]. His was a complex 3D simulation with large blocks of very similar code. Ironically, he wrote a C program to generate the ZPL for the blocks, which we then compiled back to C!

- `source_volatile()` marks the place where the data is about to be overwritten – impending def

The procedures' parameters say what data is involved and name the destination/source memory. Pairs of calls are placed inline wherever communication is needed.

This compiler-friendly mechanism specifies the interval during which transmission must take place, but it subordinates all of the details of the transmission to whatever implementation is appropriate for the underlying machine. The appropriate Ironman library is included at link time to give a custom communication implementation for the target machine.

Table 2 shows bindings for various communication methods. Notice that the four Ironman routines serve different functions for different methods. For example, in message-passing protocols that copy their data, the `DR()` and `SV()` functions are no-ops, because when the data is ready to be sent it is copied from the source, allowing the source to be immediately overwritten; when the receive operation is performed, it must wait if the data has not arrived.

One-sided and shared-memory implementations of Ironman can transfer the data immediately after it is available, that is, in `SR()`, or just before it is needed in `DN()`. The former is preferred because if the optimizer is able to find instructions to perform on the destination processor, that is, $k > 0$ in Figure 11, then communication will overlap with computation. Such cases are ideal because they hide the time to communicate beneath the time to compute, eliminating (some portion of) the time cost for that communication.

Developing these optimizations was the dissertation research of Sung-Eun Choi [45]. Although there are numerous variations, one underlying principle is to spread the interval over which communication could take place; that is, move `destination_ready()` and `source_ready()` earlier in the code, and `destination_needed()` and `source_volatile()` later, causing k to increase. This usually improves the chances that communication will overlap in time with computation, hiding its communication overhead. (She also implemented a series of other optimizations including combining, prefetching and pipelining [45].)

The proof that the message passing abstraction is "too distant" was nicely demonstrated on the Cray T3E, a computer with one-sided (Shmem) communication. A ZPL program was written and compiled. That single compiled program was loaded and run with ZPL's Ironman library implemented with Shmem (get/put) primitives and with the Ironman library implemented with Cray's own (optimized) version of MPI (send/recv) functions. The one-sided version was significantly faster [44], implying that a compiler that reduces communication to the lowest common denominator of message passing potentially gives up performance.

10.2 The Power of Testing

The ZPL compiler development was conducted like other academic software development efforts. All programmers were graduate students who were smart, well schooled compiler writers; we used available tools like RCS and CVS, and we adopted high coding standards. But, like all research software, the goal changed every week as the language design proceeded. The emphasis was less on producing robust software and more on keeping up with the language changes. This meant that the compiler was an extremely dynamic piece of software, and as it changed it

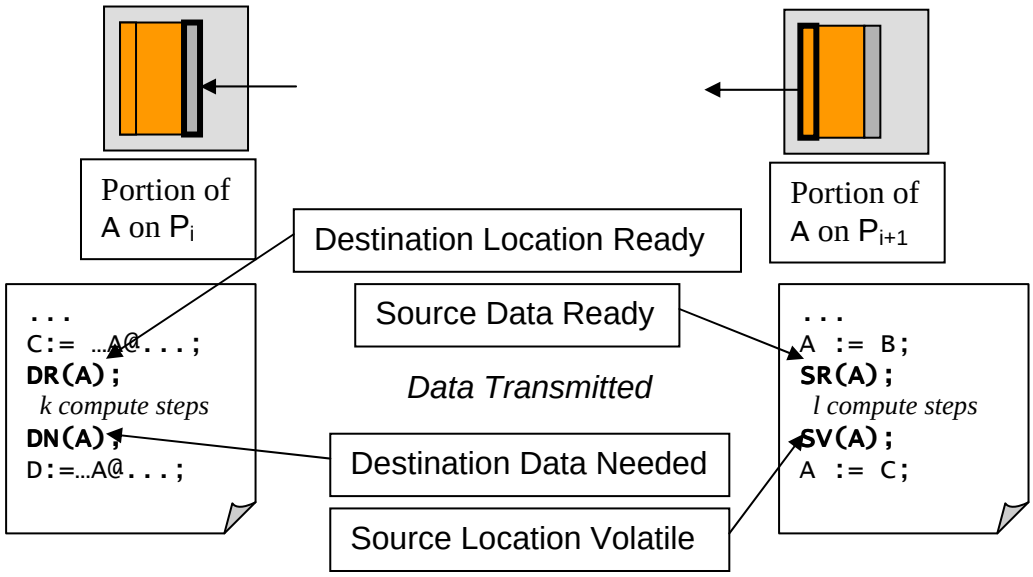

Figure 11. Schematic of Ironman call usage for a single transmission. As a result of an `A@east` operand reference, a column of `A` is transmitted from P_{i+1}, the source, to P_i, the destination. (Because ZPL generates SPMD code, P_i will also be a source, i.e. include `SR()` and `SV()` at this point, and P_{i+1} will also be a destination, i.e. include `DR()` and `DN()` at this point, to other transmissions.) The four Ironman calls are inserted by the compiler into the program text.

accreted: Features were added, then eliminated, leaving dead code, or worse, nearly dead code; implementers, uncertain of all interactions among routines, would clone code to support a new feature, leaving two very similar routines performing separate tasks, etc. And the compiler was fragile for a variety of reasons, mostly having to do with incomplete testing.

Eventually, perhaps in late 1995 or early 1996, Jason Secosky, who was taking the software engineering class, decided ZPL needed a testing system, and he built one. The facility woke up every night and ran the entire suite of ZPL test programs, reporting the next morning to the team on which of these regression tests failed.

Immediately, the testing system resulted in two significant changes to the compiler. First, the programmers became bolder in their modifications to the compiler. It was possible to make changes to the code and if something broke, it would be announced the next morning in complete detail. Second, the compiler began to shrink. Procedures with common logic were removed and replaced by a more general and more robust routines. Very quickly, the compiler tightened up.

Though the worth of continual testing is well known, it is easy to put off applying the tool too long. So significant was the improvement in the ZPL compiler that we all became evangelists for regression testing.

10.3 Scalar Performance

ZPL is a parallel language, the beneficiary of a long list of failed attempts to create an effective parallel programming system. As described above, our focus was on generating high-quality parallel code. By 1996 we were generating quality code and had been demonstrating performance with portability for some time. We were not, however, doing well enough (in our opinion) against Fortran + MPI benchmarks, and the problem was not with the parallel part of our generated code, but with the scalar portion. We were not generating naïve code, but we were not pulling out all stops either. Though we had knocked off the parallel compilation problem, it is a fact that virtually all of the "parallel part" is (necessary) overhead relative to the scalar part that actually does the work. We spent a year improving programs, especially for scalar performance.

One simple example was the Bumpers and Walkers optimization, a mechanism for walking an array using pointer arithmetic rather than direct index calculations. Bumpers move through successive elements, walkers advance through higher dimensions. Though it was somewhat subtle to get the logic fully general considering that ZPL has fluff buffers embedded in arrays used in `@` references, the improvement was significant given how common array indexing is in ZPL.

A large list of optimizations focused on applying standard compiler techniques in the ZPL setting. For example, eliminating temporaries is a tiny optimization in the scalar world, but in the ZPL world temporaries are arrays; removing them can be significant. Temporaries are introduced automatically whenever the compiler sees the left-hand side operand on the right-hand side in an `@`-translation, as in

```
[R] A := A@west;
```

This specific example, performed in place, would overwrite itself using the standard row traversal indexing from 1 to *n*, forcing the temporary. However, no temporary is needed if the references proceed in the opposite direction, *n* to 1. A compiler pass can analyze the directions to determine when temporaries can be eliminated. Because the compiler does not distinguish between whether the temporary was placed by the user or the compiler, it usually finds user temporaries to eliminate.

Another very effective improvement for ZPL's scalar code is loop-fusion. Compilers recognize when two consecutive loops have the same bounds and stride, and combine them into a single loop in an optimization known as **loop-fusion**. Fusing not only reduces loop control overhead and produces longer sequences of straight line code in the loop body, but it can, if the loops are referencing similar sets of variables, improve data reuse in the cache. (The reason is that in the double-loop case a value referenced in the first loop may have been evicted from the cache by the time it is referenced in the second loop, though the situation is quite complex [46].) ZPL's loop fusion research, chiefly by E Chris Lewis, advanced the state of the art on the general subject and improved ZPL code substantially. His work also advanced **array contraction**, an optimization in which an array that cannot be eliminated is reduced in size, usually to a scalar. Replacing an array with a scalar reduces the memory footprint of the computation, leading both to better cache performance and to the opportunity to solve larger problems [47].

Yet another class of improvements, the dissertation research of Derrick Weathersby [48], involved optimizing reductions and scans. Reduction operations (*op* <<) are often used on the same data and in tight proximity, as in

Table 2. **Sample bindings to the four Ironman procedures**. Standard communication facilities are used to implement the four Ironman routines.

	Copying message passing (nCUBE, iPSC)	Asynchronous message passing MPI Asynch	Put-based 1-sided communication (Cray T3D, T3E)	Shared memory with coherency, SMPs
Destination Loc Ready		`mpi_irecv()`	`post_ready`	`post_ready`

Source Data Ready	`csend()`	`mpi_isend()`	`wait_ready`
			`shmem_put`
			`post_done`
				`wait_ready`
				`fluff:= A`
				`post_done`

Destination Data Needed	`crecf()`	`mpi_wait()`	`wait_done`	`wait_done`
Source Volatile		`mpi_wait()`		

```
[R] range := (max<< A) - (min<< A);
```

Such cases are ideal cases for merging. A reduction usually involves three parts: combining local data on each processor in parallel, a fan-in tree to combine the *P* intermediate results, and a broadcast tree notifying all *P* processors of the final result. The last two steps involve point-to-point communications of single values to implement a tournament tree. Since two or a small number of values fit in the payload of most communication facilities, that is, a few values can be transmitted together at the cost of transmitting a single one, it makes sense to merge the fan-in/fan-out parts of reductions to amortize the cost over several operations [48].

One other group of important optimizations included Maria Gulickson's array blocking optimization to improve cache performance, Steven Deitz's stencil optimizations [49] to eliminate redundant computation in the context of @-references, and Douglas Low's partial contraction optimizations to reduce the size of array temporaries when they cannot be removed or reduced to a scalar.

The efforts were essential and rewarding—it's always satisfying to watch execution time fall—but it also seemed to be slightly orthogonal to our main mission of studying parallel computing.[15]

10.4 Factor-Join Compilation

In the same way that the WYSIWYG model called the user's attention to communication in the parallel program, it called our attention to the interactions of communication and local computation, too. For example, recognizing adjacent uses of the same communication structure usually offered an opportunity to combine them at a substantial savings, and "straight-line" computation between communication operations defined basic blocks that were prime opportunities for scalar optimization. This motivated us to abstract the compilation process in terms we described as Factor-Join [50].

In brief the idea is to represent the computation as a sequence of factors—elements with common communication patterns—and then join adjacent factors as an optimization. The approach was not a *canonical form* like Abrams' APL optimizations (Beating and Drag-along [51]); ZPL probably doesn't admit a canonical form because of the distributed nature of the evaluation. But Factor-Join regularized certain communication optimizations in such a way that they could in principle be largely automated. Exploiting these ideas would be wise when building the next ZPL compiler, but our compiler was never retooled to incorporate these ideas cleanly.

10.5 Going Public

Finally, in July 1997 ZPL was released. Though there was much more we wanted to do, the release was a significant milestone for the team. Users had worked with the language for years, so we were confident that it was a useful tool for an interesting set of applications. It might not be finalized, but it was useful.

There was a ripple of interest in ZPL, but not a lot. We had anticipated that the release would draw only modest interest for a set of obvious reasons: First, users are notoriously reluctant to adopt a new language; they are even less likely to adopt a university-produced language; and we were unabashedly a research language, which users had to expect would change and have bugs. Second, we were releasing binaries, not the sources, and they were targeted to the popular but not all extant parallel machines. Third, the software came as is, with no promises of user support or 800 number.[16] Finally, the entire community was focused on High Performance Fortran.

I don't know if the team was disappointed in the response or not—it never came up. But I was happy for the interest we got and relieved it was no greater. We didn't have the funding to provide user support, and I could not use the team for that purpose beyond our current level of expecting team members to be responsible for fixing their own bugs. Further, much more remained to do, and I wanted to preserve the option of changing the language. For the moment, we only needed enough users to keep us honest, and no more. We had that many. Over the next few years while we worked on the language, we made no effort to popularize ZPL, though users regularly found us and grabbed a copy. Periodically, we would upgrade the public software, but that was all.

11. After Classic ZPL

The language released has become known as ZPL Classic, but at the time we saw ourselves moving on to Advanced ZPL, which we abbreviated A-ZPL to suggest its generality. Indeed, we were sensitive to the fact that although ZPL Classic performed extremely well for the applications that exploited its abstractions, these abstractions needed to be more flexible and more dynamic. We'd tamed regular data parallelism; now it was time to be more expressive.

Though I'm not aware of any study proving the effect, I believe that during the five years of ZPL development 1992-1997, there was a significant advancement in parallel algorithms for scientific computation: They became much more sophisticated. If true, it was likely the result of substantial funding increases (High Performance Computing Initiative) and collaboration between the scientific computing community and computer scientists. The bottom line for us was that although ZPL Classic was expressive enough for 1992-vintage algorithms, 1997 algorithms wanted more flexibility. We were eager to add it.

[15] The effort was vindicated several years later by George Forman, a founding ZPL project member who had graduated years earlier and was working on clustering algorithms at Hewlett-Packard. While gathering numbers for a paper on performance improvement through parallelism, he wanted to compare their ZPL program to a sequential C program they had written during algorithm creation. Their measurements showed that ZPL's C "object code" was running significantly faster than their C program on one processor. George called to complain about our timer routines, but in the end, the numbers were right. The aggressive scalar optimizations were the reason. ZPL simply produces tighter, lower-level (uglier?) C code than humans are willing to write, perhaps making ZPL a decent sequential programming language.

[16] Nevertheless, the team took it as a matter of pride to respond rapidly to users' email.

This section enumerates important additions to the language.

11.1 Hierarchical Arrays

Multigrid techniques are an excellent example of a numerical method that became widely popular in the 1992-97 period, though it was known before then [52]. A typical multigrid program uses a hierarchy of arrays, the lowest level being the problem state; each level above has 1/4 as many elements in the 2D case as the array below it; a single iteration "goes up the hierarchy" projecting the computation on the coarser grids, and then it "goes down the hierarchy" interpolating the results; convergence rates are improved. For ZPL, hierarchical arrays greatly complicated regions, directions and their interactions.

Renumbering In an array programming language, multigrid computations require manipulating arrays with different, but related, index sets. The significant aspect of supporting multigrid is whether the indices are renumbered with each level. Figure 12 shows a 2D example illustrating the two obvious cases: not to renumber (12(a)), ZPL's choice, and to renumber (12(b)). That is, assuming the lower array has dense indices `[1..8,1..8]`, not renumbering implies that the level above has sparse indices `[1..8 by 2, 1..8 by 2]`, making the second element in its first row (1,3); renumbering implies that the level above has dense indices `[1..4,1..4]`, making the second element in its first row (1,2).

We struggled with the decision of whether to renumber for some time. Our colleagues in applied math who were eager to use the abstraction were initially stunned when we asked about renumbering indices. They never thought about it, and simply assumed the indices were dense at every level. For us, it seemed natural to express projecting up to the next level by the intuitive (to ZPL programmers) statement

```
[1..n, 1..n by [2,2]]  AUP :=
                    (A+A@east+A@se+A@southeast)/4;
```

which would imply no index renumbering. Though the "more intuitive" argument was strong, it was nowhere near as compelling as the WYSIWYG argument.

In ZPL an index position describes the allocation of its value completely, allowing the WYSIWYG model to describe when and how much communication takes place. For example, in Figure 12 the WYSIWYG model tells us the communication required for each scheme: In Figure 12(a) no communication is required to project up the hierarchy; in Figure 12(b) there is communication between the bottom level and the level above, though not for the next level; exact details depend on the stencil, sizes and processor assignment. In general, renumbering will force considerable communication at every level until the entire renumbered array fits on one processor. Further, the WYSIWYG model tells us that the load is balanced in Figure 12(a) but unbalanced in 12(b). The issues are somewhat more complicated "going down," but the point is clear: better performance can be expected with no renumbering.

The decision to not to renumber has stood the test of time: I remain convinced that it is the only rational way to formulate multigrid in a region-based language. We pointed out to our colleagues from Applied Math that our decision hardly mattered to them, because indices are so rarely used once the regions are set up correctly: They could continue to think of arrays as being renumbered as they programmed the computation. The actual indices matter only when analyzing performance, and if for some reason it would be better to make the arrays dense, a remap solves the problem.

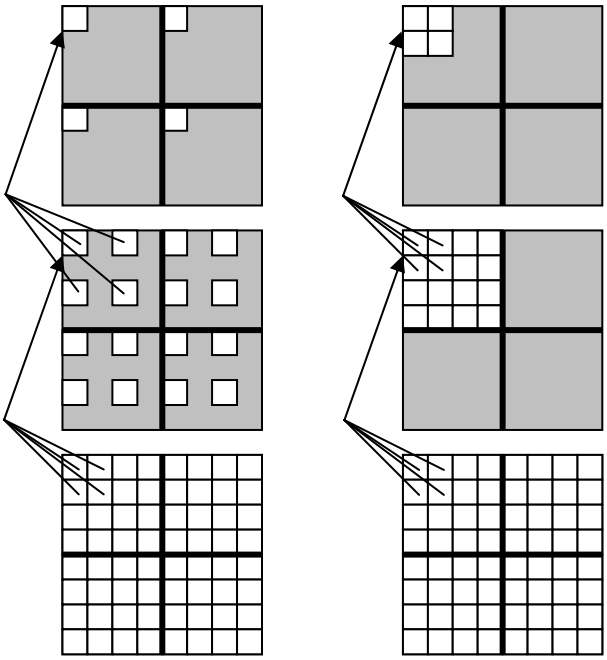

Figure 12. Multigrid. Three levels of a hierarchical array are used to express a multigrid computation; the index positions are preserved and the (logical) size remains constant in (a); the index positions are renumbered and the footprint of the array shrinks in (b).

Curly Braces Multigrid was on the cusp between ZPL and A-ZPL. Reviewing what we'd created as we moved on to A-ZPL, we were pleased that we'd figured out how to incorporate array abstractions for multigrid computations into an array language; it was surely a first. Further, the abstraction was compatible with WYSIWYG, also a major accomplishment. We were getting good performance on the NAS MG benchmark with many fewer lines of code; see Figure 13. And our first user, Joe Nichols, a UW mechanical engineering graduate student, got his combustion program to work without massive amounts of help from us; it was also faster and shorter than the C + MPI equivalent.

But there was much to criticize. ZPL programs had always tended to be clean, but multigrid computations were not. Our approach for specifying multi-regions, arrays and directions, which must get progressively sparser, was to parameterize them by sequences in curly braces. Without going into the details, the arrays for Figure 12(a) and 12(b) would have the form

```
region
  Fig12A{0..2} =
        [1..8,1..8] by [2^{},2^{}];   strides 1, 2, 4
  Fig12B{0..2} =
        [1..8/(2^{}),1..8/(2^{})];    renumbered
var     A{}: [Fig12A{}] float;
```

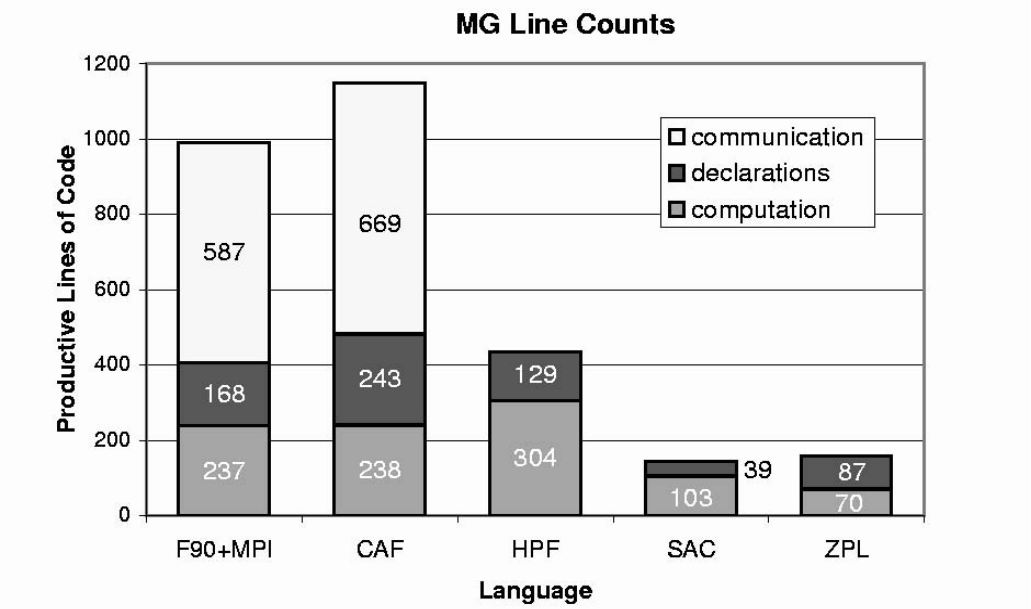

Figure 13. Line Counts for the NAS Multigrid Benchmark [63]. Lines after removing comments and white space are classified into user-specified communication, declarations or computation; published data for Fortran-90 using the Message Passing Interface, Co-Array Fortran (restricted at the time to Cray Shmem machines) [70], High Performance Fortran, Single Assignment C (restricted to shared memory computers) [59] and ZPL.

```
B{}: [Fig12B{}] float;
```

which was decidedly not beautiful. Curly braces were epidemic in all expressions involving hierarchical arrays, as this (3D) projection code:

```
procedure rprj3(var S,R: [,,] double);
 begin
  S:=0.5000 *  R +
     0.2500*(R@^dir100{}+R@^dir010{}+R@^dir001{} +
             R@^dirN00{}+R@^dir0N0{}+R@^dir00N{})+
     0.1250*(R@^dir110{}+R@^dir101{}+R@^dir011{} +
             R@^dir1N0{}+R@^dir10N{}+R@^dir01N{} +
             R@^dirN10{}+R@^dirN01{}+R@^dir0N1{} +
             R@^dirNN0{}+R@^dirN0N{}+R@^dir0NN{})+
     0.0625*(R@^dir111{}+R@^dir11N{}+
             R@^dir1N1{}+R@^dir1NN{}+
             R@^dirN11{}+R@^dirN1N{}+
             R@^dirNN1{}+R@^dirNNN{});
 end;
```

from the NAS MG illustrates. Further, as discussions with Nichols and users from Applied Math indicated, parameterized regions and directions were not intuitive.

Over time, we came to this conclusion [33]: The complications of this hierarchical array formulation with its parameterization were inevitable because ZPL's regions and directions were not first class. To be **first class** means they should be treated like any other value. (The fact that ZPL's regions and directions were not first class was an oft-criticized aspect of ZPL Classic.) We had generalized them to support hierarchical structures, but this was too situation-specific. If we made regions and directions first class—thereby permitting arrays of regions and arrays of directions—then a clear, orthogonal formulation of hierarchical arrays with all the desirable WYSIWYG properties would fall out. And the cumbersome direction references (also evident in the foregoing code) would be fixed. Chamberlain presented this analysis in his dissertation [33], and that is how multigrid is now computed in A-ZPL.

11.2 User-defined Scans and Reduces

Scan is a parallel prefix operator, which, like reduce, uses the base list of associative operators (`+`, `*`, `&`, `|`, `max`, `min`); unlike reduce, scan returns all intermediate results. ZPL always included scan, but few numerical programmers use it.[17] Nevertheless, researchers like Blelloch [53] have argued that scanning is a powerful parallel problem solving technique. We agree and were motivated to make it a powerful tool in ZPL.

The idea of user-defined reductions and scans was first discussed in the early days of APL. They have been implemented sequentially, and on shared-memory parallel machines, but the full application of the parallel prefix algorithm [54] on the CTA architecture remained unsolved. The problem was that in the standard sequential implementation the reduce/scan proceeds through the elements first to last, processing one result at a time directly. In the ZPL formulation of the parallel prefix algorithm each processor performs a local reduce/scan, and then these local intermediate results are combined; finally, for the scan the local values must be updated based on intermediate results computed on other processors. Steven Deitz worked out the logic for generalized reduction and scan [55, 56]. Our algo-

[17] The exception proving the rule was a solar system simulation testing planetary stability written with UW Astronomers by Sung-Eun Choi; the inner loop was a scan over records each of which was a descriptor of the orbits of the planets for an (Earth) year.

rithm must be broken into five parts: initialize, local, combine, update (for scan only) and output.

Further, the input and output data types processed by these routines are potentially different. For example, the `Third_Largest_Value` reduce of a numeric array is an example of a user-defined reduction; it returns the input array's third largest value. The base input data type is numeric, but the intermediate routines pass around records of numeric triples, and the output is again numeric. The four components for this reduction have the following characteristics:

- Initialize starts the local reduction on each processor, emitting a record triple containing identity values.
- Local takes a numeric value and a triple and returns a new updated next triple.
- Combine takes two numeric triples and returns the triple of their three largest numbers.
- Output takes the global triple and returns the final numeric result.

User defined reductions and scans enjoy the same benefits (mentioned earlier) as their built-in counterparts, such as participating in combining optimizations.

As a postscript, notice two curious properties of scan.

- As mentioned earlier, ZPL started out following APL by specifying that partial reduce and scan use dimension numbers in brackets, as in the APL, +/[2]*A*. Partial reduction was changed to use the "two regions" solution as a result of formulating flood, but partial scan continues to specify the dimensions by number, as in

  ```
  [R] Temp := +||[-2] A;
  ```

  where negative numbers indicate that the scan is to be performed from high indices to low. The key reason for the apparently inconsistent syntax for apparently similar operations is that scan doesn't change rank.[18]
- A moment's thought reveals that the compiled code for the procedure calls and interprocessor communication for higher-dimensional scans is potentially very complex. But as several researchers have noted, higher-dimensional scans can be expressed in terms of a composition of lower-dimensional scans that can be implemented in a source-to-source translation. Deitz [55] observed that the key to this technique in the context of user-defined scans is to choose exclusive rather than inclusive scans. The difference is whether each element is included in its replacement value or not, as in

  ```
  +|| 2 4 6  ⇔  2  6 12      Inclusive
  +|| 2 4 6  ⇔  0  2  6      Exclusive
  ```

  where the first element in the exclusive scan is the identity for the operation. The inclusive form can be found from the exclusive by element-wise combining with the operand. Though APL and most languages use the Inclusive form, A-ZPL ultimately adopted the more general exclusive scan, because generating code for high-dimension user defined scans is extremely difficult otherwise.

Scan is an operation worthy of much wider application.

11.3 Pipelines and Wavefronts

It is not uncommon in scientific computation to compute values that depend on just computed values elsewhere in the array. For example, the new value of $A_{i+1,j+1}$ may depend on $A_{i,j}$, $A_{i+1,j}$ and $A_{i,j+1}$. This looks more like a scan in two dimensions (simultaneously) than it looks like the other whole-array operations of ZPL. But it is a bigger and more complex scan. Our solution quickly took on the name **mighty scan**.

Whereas standard scan and its user-defined generalizations apply to array elements in one dimension (at a time) with a single update per element, wavefront computations must proceed in multiple dimensions at once (three is most typical) and perform much more complex computations. The solution, developed by E Chris Lewis [47], involved two additions to the language: *Prime ats* and *scan blocks*.

The `@`-translations were extended to add a prime, as in

```
[1..n, 2..n] A := A + A'@west;
```

The **prime at** indicates that the referenced elements refer to the *updated* values from earlier iterations in the evaluation of *this* statement. Thus, in the illustrated statement, the first column of `A` would be unchanged (note region), the second column would be the sum of the first two, the third would be the sum of the first three, because it combines the third column (`A`) and the newly updated second column `A'@west`, etc. (Of course, this simple one-dimension example can also be expressed as a partial scan.)

Production applications perform complex calculations in wavefront sweeps, updating several variables at once. Sweep3D, a standard wavefront benchmark, updates five 3D arrays, four as wavefronts, in its inner loop. In order to permit such complex wavefronts, the scan block was added as a statement grouping construct

```
[R] scan
      A := A*B + A@w*B@w + A@nw*B@nw + A@n*B@n;
      B := r*B +   s*B@w +    t*B@nw +   u*B@n;
    end;
```

Essentially, the scan block fuses the loop nests of all the array assignments into a single loop nest, allowing the updates to be propagated quickly [57]. Notice that because these operations are performed in parallel on values resident on other processors, only a subset of the processors can start computing at first. Once edge data can be sent, adjacent processors can begin computation. This raises interesting scheduling questions relative to releasing blocked processors quickly.

[18] Notice the three related operators reduce (*op* <<), scan (*op* ||) and flood (>>) have the properties that the result is smaller than, the same as, or larger than the operand, motivating the choice of operator syntax.

The prime at and scan block were extremely successful in the parallel benchmark Sweep3D.[19] The core of the Sweep3D Fortran code is 115 lines, while it is only 24 in ZPL. Further, ZPL's performance is competitive (+/- 15%) with the Fortran + MPI on the measured platforms [47].

11.4 Sparse Regions and Arrays

ZPL Classic has dense regions, strided regions, and regions with flood dimensions. Arrays can be defined for all of them. Further, it has a `with` region operator for masking. But, as our numerical computation friends kept reminding us, most huge problems are not dense. Brad Chamberlain was especially concerned about the matter, and ultimately cracked the problem. There were several issues, and since MATLAB was the only language offering sparse arrays—but interpreted sequentially rather than compiled in parallel—he had a lot of details to work out.

The first concerned **orthogonality**, meaning that because exploiting sparseness is simply an optimization, the use of sparse arrays and regions should match the use of ZPL's other forms of arrays and regions, differing possibly only by the sparse modifier on the declarations. Chamberlain's formulation achieved that, but there is a wrinkle to be explained momentarily.

The second concerned performance. Computer scientists and users have created a multitude of sparse array representations, and each has its advantages. We wanted a solution that worked well—defined to mean that the overhead per element was a (small) constant for all sparse configurations. Further, the representation must work well in ZPL, meaning that it fits within the specifications of the WYSIWYG model. Since there is no universal sparse representation, the internal representation was a difficult task; our one advantage was that the operators and data-reference patterns of the rest of the language were known, and they were the only ones that could manipulate sparse arrays.

The third issue was, what does "sparse" mean? Numerical analysts think of a sparse array as containing a small number of nonzero elements, where "small" is in the eye of the beholder, but is surely a value substantially less than half. "Zero" means a double precision floating-point 0. But couldn't a galaxy photo be a sparse pixel array where "zero" means "black pixel", i.e. an RGB triple of zeroes? We agreed that a sparse array was any array in which more than half of the elements were *implicitly represented values* (IRV), and the user can define the IRV.

The unexpectedly difficult aspect of sparse arrays, the wrinkle, concerned initialization. The issue is that, given a dense array, it is easy to produce a sparse region from it efficiently, say by masking (`with`). And given a sparse region, it is easy to produce another sparse region from it. But how to create a huge sparse region in the first place without creating a huge dense region to cover it? It could be read in, but in ZPL I/O must be executed in some region context, which will either be a dense region or lead to a circular solution. Another approach is to create the initial sparse region by parallel computation, say with a random number generator, but how is this computation specified if the region does not yet exist? In the end the general problem was never solved. ZPL has full support for sparse regions, but they are either created from dense regions or set up with tools external to the language. This latter is not wholly unsatisfactory, because sparsity patterns often have odd sources that require external support to use.

Chamberlain advanced the state of the art dramatically, producing the first array language abstraction and first comprehensive compiler support for sparse arrays. It is unquestionably one of the premiere accomplishments of the ZPL research. More remains to be done: the sparse array initialization problem is a deep and challenging topic worthy of further research.

11.5 Data Parallelism and Task Parallelism

ZPL Classic is a data-parallel language, and A-ZPL adds features that go well beyond data parallelism. But it is widely believed that task parallelism is also an essential programming tool, and we were eager to include such facilities in A-ZPL. Several data-parallel languages have been generalized to include task parallelism, and vice versa. Steven Deitz investigated which of these was the most successful, and concluded that none had been. Essentially, his analysis seemed to imply that once a language acquires its "character"—data parallelism or task parallelism—fitting facilities for the other paradigm into it cleanly and parsimoniously is difficult, we guessed impossible. This result was not good news.

Nevertheless, Deitz continued to investigate how to add more flexibility and dynamism in ZPL's assignment of work and data to processors. Of several driving applications, the adaptive mesh refinement (AMR) computation was one he considered closely.

To appreciate the solution, recall that the parallel execution environment of ZPL is (by default) initially defined in an extra-language way. That is, the number of processors and their arrangement are specified on the command line. Parameters (configuration constants and variables) are also defined on the command line, and are used to set up regions and arrays when the computation starts. These inputs define the default allocation of arrays to processors, which is the basis of understanding the WYSIWYG performance model. ZPL is always initialized this way, and previously, none of it could be changed in the program.

New Features To add flexibility two new abstractions were added to the language, *grids* and *distributions* [55]. They are both first class, and with regions elevated to first-class status, programmers can completely control all the features originally provided by default.

Grids are a language mechanism for defining the arrangement of processors. The values of grid names are dense arrays of distinct processor numbers 0 to *P*-1 of any rank. The assignment of numbers to positions is under user control, giving considerable flexibility in arranging logical processor adjacencies. In concept, the initial (command line) setup defines *P* and sets the first arrangement.

Distributions are a mechanism for mapping regions to grids. The user specifies the range of indices to be assigned in each dimension. Further, the type of allocation scheme—block, cyclic, etc.—is also specified for each dimension. Users can default to compiler-provided implementation routines for the allocation, or program their own. Again,

[19] Indeed `scan/end` are sufficiently powerful that they have been generalized and renamed as `interleave/end`.

the initial setup defines the initial (block) assignment of the regions.

The typical use of these facilities to redefine the runtime environment is as follows. Within the current parallel context, determine how the data should be reallocated. Then

- ⇓ Restructure the processors into a new grid, if necessary,
- ⇓ Specify how the data should be allocated in the new arrangement, if necessary,
- ⇓ Define a new region to assign appropriate indices to the problem, if necessary, and
- ⇓ Say whether the array reallocation is to preserve or destroy the data contained therein.

If these operations are specified in consecutive statements, they are bundled together and performed as one "instantaneous" change in environment. This strategy minimizes data motion and the use of temporaries. The statements can realize a complete restructuring of the computational setting.

Notice that although there is considerable flexibility, one feature—that all arrays with the same distribution and a given index, say *i, j, …, k,* will have that element allocated to the same processor—is maintained by these ZPL facilities. This is a property of the base WYSIWYG model, and preserving it is key to preserving the programmer's sanity. After all, once grid-distribution-region-array operations set up a new parallel context, the user must morph his or her understanding of the WYSIWYG model to accord with it. This isn't really difficult, because the WYSIWYG ideas were probably used to create the new context in the first place. Nevertheless, treating the indices consistently helps.

In Place Restructuring The distribution facilities are extremely interesting and worthy of much greater study. However, there are two curious aspects worth mentioning.

When a distribution binds a region to a grid, there is a question of what happens to the data stored in the arrays when that region had a different structure. It could be lost, or it could be preserved as far as possible, i.e. if an index in the old region is also in the new region, then the array values with that index could be preserved in the arrays. It seemed initially that preserving the values was essential, but as the facilities were used in sample programs, it became clear that both schemes—destroy or preserve—were useful. Though preserve subsumes destroy—the preserved values can be overwritten—it is a waste to spend communication resources moving the data when it is not needed. So, ZPL has both a destructive assignment (`<==`) and a preserving assignment (`<=#`) to support region change [41].

The ability to reallocate the data to the processors so conveniently has motivated new algorithms. For example, the standard solution for a 3D FFT specifies that a 1D FFT be performed along each of the three dimensions. The standard way to program this is to set up an allocation of the 3D array over a 2D arrangement of processors, keeping the dimension along which the 1D FFT is performed local to the processors. (This saves shipping data around in the complex butterfly pattern.) The array is then transposed to localize a different dimension, the 1D FFT is applied to that dimension, and another transpose is performed to localize the last dimension. The alternate way to program this in ZPL is to avoid the transposes and simply redistribute the region across the processors. That is, the 3D region is mapped differently to the 2D processor grid. The details are complex, but result in better performance chiefly because the data is not repacked into the local portions of the array. That is, it saves local computation.

We continue to develop a better understanding of these facilities.

12. Problem Space Promotion

One aspect of ZPL not yet mentioned is that it has contributed to new parallel computation techniques, analogous to serial techniques like divide-and-conquer. One interesting example is Problem Space Promotion (PSP) [58]. **Problem Space Promotion** sets up a solution in which data of *d* dimensions is processed in a higher dimensionality that only logically exists in the same way the "tree" is only logical in the divide-and-conquer paradigm. The advantage of the PSP strategy is that it dramatically increases the amount of parallelism available to the compiler.

To explain PSP, consider the task of rank sorting *n* distinct values by making all n^2 comparisons, counting the number of elements less than each value, and then storing the item in the position specified by the count. (This is illustrative of an all-pairs type computation, and is not the recommended sorting algorithm, which is Sample Sort [58].) All-pairs computations, though common, would not normally be solved using n^2 intermediate storage for *n* inputs. Such a large intermediate array, even composed of one-bit values, would likely not fit in memory, and being lightly used has very poor cache performance.

Because of the flood operator and flood dimensions of ZPL, it is possible to perform this algorithm *logically* without actually creating the n^2 intermediate storage. The solution *promotes* the problem to a logical 2D space, by adopting the declarations

```
region R = [1, 1..n];          A "row" region
var V, S : [R] integer;        Input (V) and Output (S)
     FIR : [*,1..n] integer;   Flood array
     FIC : [1..n,*] integer;   Flood array
```

`V` and `S` are the input and output, respectively, and `FIR`—mnemonic for flood in rows—and `FIC` are helper arrays. They are logically 2D, but physically only contain (a portion of) the row or column.

The logic begins by initializing the two helper arrays using flood

```
[1..n,*] FIC := >>[1..n,1] V#[1, Index1];
```

where to set the `FIC` array, the input must first be transposed using remap (`#`) to change dimensions. If the input (in `V`) were `2 6 7 1 5`, we would have these arrays:

```
FIR:   2  6  7  1  5      FIC:   2  2  2  2  2
       2  6  7  1  5             6  6  6  6  6
       2  6  7  1  5             7  7  7  7  7
       2  6  7  1  5             1  1  1  1  1
       2  6  7  1  5             5  5  5  5  5
```

The sorting—actually the construction of the permutation that specifies where the input must go to produce a sorted order—is a "one-liner":

```
[R] S := +<<[1..n,1..n](FIC<=FIR);
```

which can be deconstructed as follows. The (logical) Boolean array showing which elements are smaller than another element,

```
FIC<=FIR:      1  1  1  0  1
               0  1  1  0  0
               0  0  1  0  0
               1  1  1  1  1
               0  1  1  0  1
```

is reduced to the vector to reorder the data by adding up the columns:

```
+<<[1..n,1..n](FIC<=FIR):    2  4  5  1  3
```

The columns are reduced because a comparison of the applicable region `R=[1,1..n]` and the expression region in the partial reduction `[1..n,1..n]` reveals that the first, i.e. columnar, dimension collapses.

A remap,

```
[R] S := V#[1,S];
```

produces the reordered data. A stable sort, i.e. a sort preserving the position of non-distinct input, only requires the addition of two subexpressions in the "one-liner".

The key point about the PSP is that the processors are logically allocated in a 2D grid, so that the logical work is assigned to them all. With full parallelism and with the logically flooded arrays "banging" on the portion of the single row and column assigned to their processor, the performance is quite satisfactory [58] and the apparently impractical solution becomes practical for many n^2 and higher computations, including 3D matrix multiplication, n^2 *n*-body solutions, Fock matrix computations and more.

13. ZPL And HPF Comparisons

ZPL and HPF shared the same goal: Provide a global-view programming language for parallel computers and rely on the compiler to generate the right (communication, synchronization, etc.) code.

From that common goal the two efforts diverged, ZPL creating a new language and HPF parallelizing Fortran 90. Because these are such dissimilar approaches, there is little basis for comparison. The features in ZPL are the result of a from-first-principles design responding to the dictates of the CTA type architecture. The features in HPF came with Fortran 90, which is based on the RAM type architecture. Though we have observed, for example, that regions are better than array slices for data-parallel computation, the observation gives little insight about the two efforts because slices were given; HPF researchers studied different issues.

Though attempting such comparisons is not productive, I can say what benefits I think we got by our approach. Our type architecture approach and the CTA specifically gave the ZPL language designers and compiler writers the key parameters of the computers they were targeting: scalable *P*, unit time local memory reference, $\lambda >> 1$ nonlocal memory references, etc. We could formulate programming abstractions to accommodate those properties. As compiler writers we could target the generic CTA, confident that the object code would run well (with comparably good performance) on any MIMD parallel computer. Further, ZPL programmers could analyze their code using the WYSIWYG model as a means of predicting program performance and then observe that performance in fact. None of this was available to HPF.

One curious similarity is that neither language had a significant installed base of programs. ZPL was new, but in 1992 so was Fortran 90. In the early years it was difficult to find benchmark HPF programs to compare against.

Of course, it was not an advantage to the ZPL effort that most of the research community, nearly all of the funding, and most of the obvious corporate players were actively directed towards a different approach. The "group think" was pervasive. One difficult problem, common in reviews of papers and proposals, was for ZPL work to be dismissed because it didn't address problems faced by HPF, that is, parallelizing sequential programs. A typical remark was "Of course you get outstanding performance; you changed the language to make it easy." The result was that many referee reports did not speak to the merits (or lack thereof) of our papers.

Incidentally, the most foolish of the new-language criticisms was, "Design of new programming languages, including parallel programming, ceased to be even slightly interesting many, many years ago (circa 1980)." This was so narrow-minded (and wrong as Java, C#, Perl, Python, etc. prove) that the phrase "ceased to be interesting circa 1980" became a joke put-down among the team members. It always got a laugh.

We knew that ZPL was on the right track based on the performance numbers we were getting, but with so much of the community invested in HPF, ZPL's success wasn't welcome news. One paper—"ZPL vs HPF: A Comparison of Performance and Programming Style" [59]—demonstrated experimentally in 1995 that the ZPL approach was more effective than HPF. But the paper was not accepted *for partisan reasons*, as the conference program committee chairman admitted to me, and has never been published. It is interesting to speculate how a public debate in 1995 on the merits of the two approaches might have changed subsequent history.

The most serious consequence for ZPL of so much of the field being focused on one approach occurred towards the end of the 1990s, when we were extending A-ZPL and pushing on its flexibility: Essentially all funding for parallel programming research dried up, making it extremely difficult for us to complete our work. Speculating as to why this might have occurred, we can acknowledge unrelated events such as changes at funding agencies, the distraction of the "dot com boom," etc. But, based on discussions at the time, the most likely explanation was the dawning realization by many in the field—researchers, funders, users—as to what the expenditure of so much talent and treasure on HPF was likely to produce. Parallel programming research quickly became *technologia non grata*.

Despite the difficulties there is today, for a "David-size" fraction of the people and money, a publicly available open source ZPL compiler, which seems to be used for classroom and research purposes. Further, ZPL concepts have significantly influenced the next generation of parallel languages, Cray's Chapel [60] and IBM's X10 [61].

14. Assessment

The ZPL Project started with three goals: performance, portability and convenience, and it is essential to ask: how did we do? Though readers can consult the literature for detailed evaluations and judge for themselves, I am ex-

tremely pleased. As evidence consider data from a detailed analysis of the NAS MG benchmark [62, 63]. Our paper focuses on languages with implemented compilers as of 2000 with published results for the MG benchmark over the standard range of datasets. It contains evidence supporting all three goals.

Figure 13 gave the convenience data from the paper, as measured by lines of code. ZPL is nearly the most succinct of the programs. As explained earlier, this version of ZPL did not have first-class regions or directions, resulting in code less elegant than it should be; with the revisions to hierarchical arrays [33], however, the program would be more readable, and perhaps slightly shorter. The key point to notice is that the computation portion of the Fortran 90 + MPI and CAF programs are not only much longer, but that the main constituent of the difference is communication code that is extremely difficult to write and debug. So, although these programs can give good performance, only ZPL, HPF and SAC truly simplify the programmer's task. (SAC is a functional language targeting only shared-memory machines.)

Figure 14 reproduces two pages of performance graphs from the analysis of the MG benchmark programs [63]. These tables reveal data about the performance the pro-

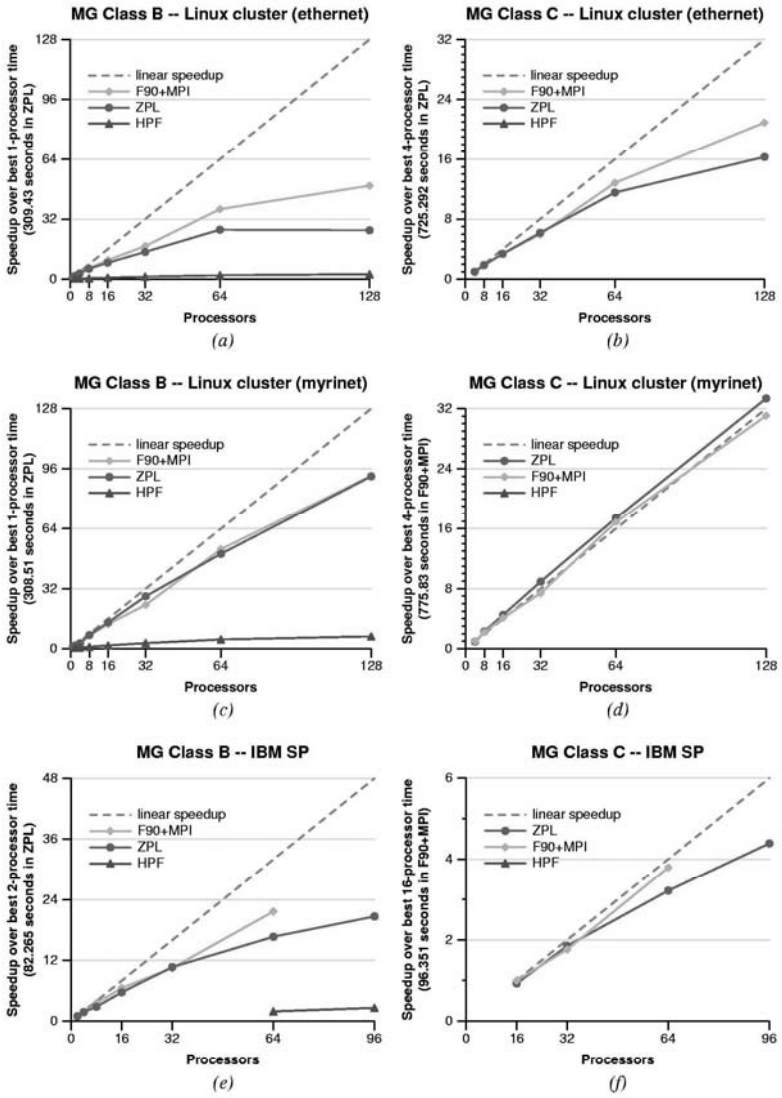

Figure 4: Speedup curves for classes B and C of MG on the Linux cluster using both ethernet and myrinet, and on the IBM SP. Note that there is not enough memory to run these problem sizes on small processor sets. Thus, we compute speedups for each graph using the fastest execution time on the smallest number of processors (indicated in the *y*-axis label). Due to its excessive memory allocation, the HPF version is unable to run on most configurations. Note that the F90+MPI version cannot run on 96 processors since it is not a power of two.

Figure 14. Performance Graphs from "A comparative study of the NAS MG benchmark across parallel languages and architectures" [63]. Speedup is shown; the dashed line is linear speedup.

grams can achieve across a series of machines, i.e. their portability.

Regarding portability, the figures show data for six different MIMD machines: two shared memory machines (Enterprise, SGI Origin), two distributed memory machines (Cray T3E, IBM SP-2), and two Linux clusters (Myrinet, Ethernet. These computers represent all the major architectures at the time except the Tera MTA [64], for which no MG program was available. (Performance numbers are missing in many cases, usually because the language does not target the machine or, for some HPF cases, because there was not enough memory.)

The performance numbers show that ZPL produces code that is competitive with the handwritten message passing code of F90 + MPI; further experiments and more detailed analyses are found in the Chamberlain [33], Lewis [47] and Deitz [55] dissertations. Not only is the performance portable across the machines, ZPL scales well, generally doing better on larger problems. This is gratifying considering the programming effort required to write message-passing code compared to ZPL. It seems that for this sampling of computers, ZPL is comparable to the hand-coded benchmark, which was the performance-with-portability goal.

In closing, notice that the results are not given for the F90+MPI message passing program for processor values *P* that are not a power of 2. This is due to the fact that the programmers had to set up the problem by hand without the benefit of any parallel abstractions, and in doing so, they embedded the processor count in the program; making it a power of 2 was a simplifying assumption. In ZPL *P* is

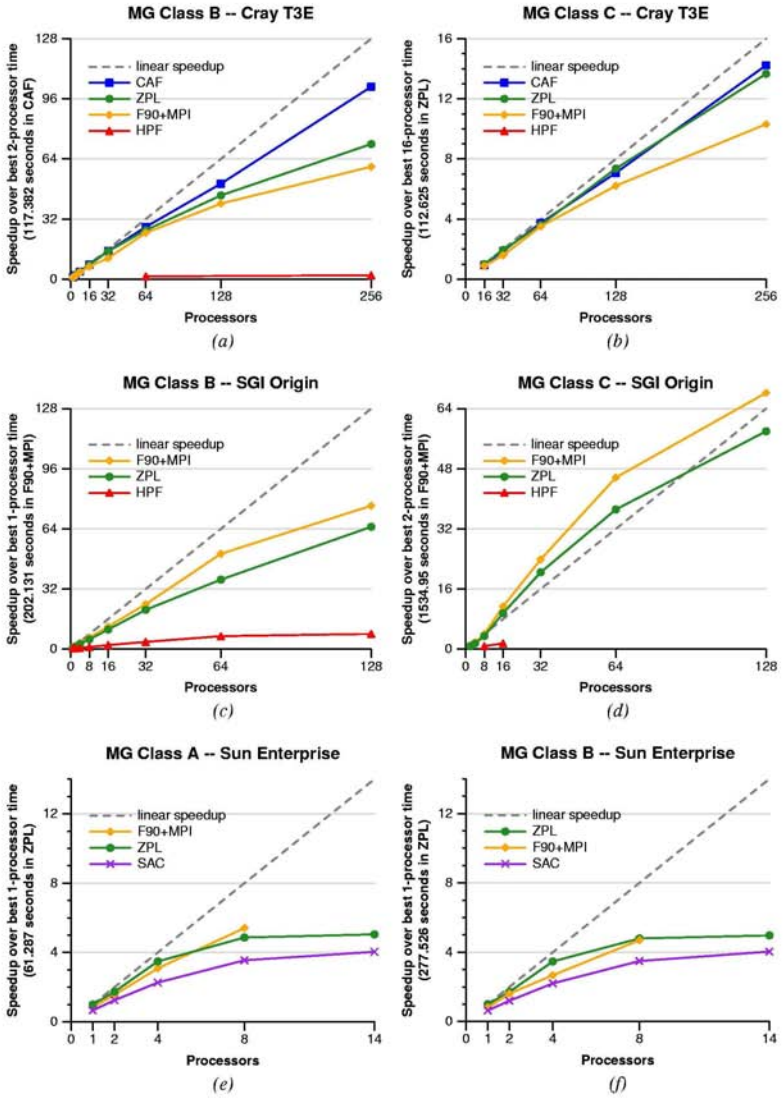

Figure 5: Speedup graphs for MG classes B and C on the Cray T3E and SGI Origin, and for classes A and B on the Sun Enterprise 5500. As in the previous figure, speedups for each graph are computed using the fastest execution time on the smallest number of processors. The superlinear speedup on the Origin is due to the memory traffic required to run a class C problem on 2 processors. We were unable to obtain reasonable timings on more than 128 Origin processors due to the network traffic involved in crossing between machines. See Appendix B for details.

Figure 14 (continued). **Performance Graphs** from "A comparative study of the NAS MG benchmark across parallel languages and architectures" [63].

assigned on the command line and the compiler sets up the problem anew for each run. This flexibility seems like a small thing. But it is symptomatic of a deeper problem: Building the parallel abstractions manually, as is necessary in message passing, is error prone. A year after the Figure 14 results were published, while running the ZPL Conjugant Gradient (CG) program on a 2000 processor cluster at Los Alamos, we discovered that for $P \geq 500$, the F90+MPI program didn't pass verification, though the ZPL did. What was wrong? The NAS researchers located the problem quickly; it was an error in the set-up code that was only revealed on very large processor configurations. This reminded us that one advantage to a compiler generating the code for an abstraction is that it only has to be made "right" once.

15. Summary

Though the ZPL project achieved its goals of performance, portability and convenience—making it the first parallel programming language to do so—the accomplishments of greatest long-term value may be the methodological results. The type architecture approach, used implicitly in sequential languages like Fortran, C and Pascal, has been applied explicitly to another family of computers for the first time. With silicon technology crossing the "multiple processors per chip" threshold at the dawn of the 21st century, there may be tremendous opportunities for architectural diversity; the type architecture methodology, we are beginning to see [19], allows a straight path to producing languages for new families of machines. Further, the WYSIWYG performance model transfers the compiler writer's know-how to the programmer in a way that has direct, visible and practical impact on program performance. As long as faster programs are better programs, models like WYSIWYG will be important.

Acknowledgments

It is with sincere thanks and with deepest appreciation that I acknowledge the contributions of my colleagues on the ZPL project: Ruth Anderson, Bradford Chamberlain, Sung-Eun Choi, Steven Deitz, Marios Dikaiakos, George Forman, Maria Gulickson, E Chris Lewis, Calvin Lin, Douglas Low, Ton Ngo, Jason Secosky, and Derrick Weathersby. There could not have been a research group with more energy, more smarts, more creativity or a higher "giggle index" than this ZPL team. Other grad student contributors included Kevin Gates, Jing-ling Lee, Taylor van Vleet and Wayne Wong. Judy Watson, project administrator, was a steady contributor to ZPL without ever writing a single line of code. There is a very long list of others to whom I'm also indebted, including grad students on my other parallel computation projects, faculty colleagues, research colleagues at other institutions, and scientists both at UW and elsewhere in the world. It has been a great pleasure to work with you all. Finally, for this paper I must again thank Calvin, Brad, E and Sung for their help, thank the tireless HoPL3 editors, Brent Hailpern and Barbara Ryder, for their patience, and the dedicated anonymous referees, who have so generously contributed to improving this paper.

References

[1] Lawrence Snyder. Type architecture, shared memory and the corollary of modest potential. *Annual Review of Computer Science*, 1:289-317, 1986.

[2] Gil Lerman and Larry Rudolph. *Parallel Evolution of Parallel Processors*, Plenum Press, 1993.

[3] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon and D. W. Walker. *Solving Problems on Concurrent Processors*, Prentice-Hall, 1988.

[4] Lawrence Snyder, The Blue CHiP Project Description. Department of Computer Science Technical Report, Purdue University, 1980.

[5] Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1993.

[6] Calvin Lin and Lawrence Snyder. SIMPLE performance results in ZPL. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1994.

[7] Lawrence Snyder. Parallel programming and the poker programming environment. *Computer*, 17(7):27-36, July 1984.

[8] Lawrence Snyder. Introduction to the configurable, highly parallel computer. *Computer*, 15(1):47-56, January 1982.

[9] Richard O'Keefe, *The Craft of Prolog*, MIT Press, 1994.

[10] Simon Peyton Jones and P.L. Wadler. A static semantics for Haskell. University of Glasgow, 1992.

[11] Jorg Keller, Christoph W. Kessler and Jesper Larsson Traff. *Practical PRAM Programming*, John Wiley, 2000.

[12] Richard J. Anderson and Lawrence Snyder. A comparison of shared and nonshared memory models of parallel computation. *Proceedings of the IEEE*, 79(4):480-487, April 1991.

[13] Alan M.Turing. On computable numbers, with an application to the Entscheidungsproblem, Proc. London Math. Soc., 2(42):230-265, 1936.

[14] John Backus. Can programming be liberated from the von Neumann style? *Communications of the ACM* 21(8):613-641, 1978.

[15] J. T. Schwartz. Ultracomputers. ACM Transactions on Programming Languages and Systems, 2(4):484-521, 1980.

[16] .I.D. Scherson and A.S. Youssef. *Interconnection Networks for High-Performance Parallel Computers*. IEEE Computer Society Press, 1994

[17] K. Bolding, M. L. Fulgham and L. Snyder. The case for Chaotic adaptive routing. *IEEE Trans. Computers* 46(12): 1281-1291, 1997.

[18] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation, *ACM Symposium on Principles and Practice of Parallel Programming*, 1993.

[19] Benjamin Ylvisaker, Brian Van Essen and Carl Ebeling. A Type Architecture for Hybrid Micro-Parallel Computers. In IEEE *Symposium on Field-Programmable Custom Computing Machines*, 2006.

[20] David G. Socha. *Supporting Fine-Grain Computation on Distributed memory Parallel Computers.* PhD Dissertation, University of Washington, 1991.

[21] Calvin Lin, Jin-ling Lee and Lawrence Snyder. Programming SIMPLE for parallel portability, In U. Banerjee, D. Gelernter, A. Nicolau and D. Padua (eds.), *Languages and Compilers for Parallel Computing*, Springer-Verlag, pp. 84-98, 1992.

[22] Calvin Lin. *The Portability of Parallel Programs Across MIMD Computers*. PhD Dissertation, University of Washington, 1992.

[23] Ton Ahn Ngo and Lawrence Snyder. On the influence of programming models on shared memory computer performance. In *Proceed-*

ings of the Scalable High Performance Computing Conference, 1992.

[24] Ton Anh Ngo. *The Role of Performance Models in Parallel Programming and Languages*. PhD Dissertation, University of Washington, 1997.

[25] Raymond Greenlaw and Lawrence Snyder. Achieving speedups for APL on an SIMD distributed memory machine. *International Journal of Parallel Programming*, 19(2):111—127, April 1990.

[26] Walter S. Brainerd, Charles H. Goldberg and Jeanne C. Adams. *Programmer's Guide to Fortran 90*, 3rd Ed. Springer, 1996.

[27] High Performance Fortran Language Specification Version 1.0 (1992) High Performance Fortran Forum, May 3, 1993. [34] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD Dissertation, University of Washington, 2001.

[28] Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. A compiler abstraction for machine independent parallel communication generation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1997.

[29] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477—518, 1996.

[30] J. R. Rose and G. L. Steele Jr. C*: An extended C language for data parallel programming. In *Proceedings Second International Conference on Supercomputing*, 1987.

[31] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL's WYSIWYG performance model. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1998.

[32] Sung-Eun Choi. *Machine Independent Communication Optimization*. PhD Dissertation, University of Washington, March 1999.

[33] Bradford L. Chamberlain. *The Design and Implementation of a Region-based Parallel Programming Language*. PhD Dissertation, University of Washington, 2001.

[34] Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder. Problem space promotion and its evaluation as a technique for efficient parallel computation. In *Proceedings of the ACM International Conference on Supercomputing*, 1999.

[35] Robert van de Geijn and Jerrell Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience*, 1998.

[36] R.E. Cypher. J.L.C. Sanz and L. Snyder. Algorithms for image component labeling on SIMD mesh connected computers. *IEEE Transactions on Computers*, 39(2):276-281, 1990.

[37] MasPar Programming Language (ANSI C-compatible MPL) Reference Manual Document Part Number: 9302-0001 Revision: A3 July 1992.

[38] Gregory R. Watson. *The Design and Implementation of a Parallel Relative Debugger*. PhD Dissertation, Monash University, 2000.

[39] W. P. Crowley et al. The SIMPLE code. Technical Report UCID 17715, Lawrence Livermore Laboratory, February 1978.

[40] Lawrence Snyder. *A Programmer's Guide to ZPL*. MIT Press, Cambridge, MA, 1999. (The language changed in small ways and has been extended; it is now most accurately described in Chapter 2 of Chamberlain [33].)

[41] Steven J. Deitz, Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. The design and implementation of a parallel array operator for the arbitrary remapping of data. In *Proceedings of the ACM Conference on Principles and Practice of Parallel Programming*, 2003.

[42] Adam Beguelin, Jack Dongara, Al Geist, Robert Manchek , and Vaidy Sunderam. User guide to PVM. Oak Ridge National Laboratory, Oak Ridge TN 378 316367, 1993.

[43] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: the Complete Reference*. MIT Press, Cambridge, MA, USA, 1996.

[44] Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. A compiler abstraction for machine-independent parallel communication generation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1997.

[45] Sung-Eun Choi. *Machine Independent Communication Optimization*. PhD Dissertation, University of Washington, March 1999.

[46] E Christopher Lewis, Calvin Lin, and Lawrence Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1998.

[47] E Christopher Lewis. *Achieving Robust Performance in Parallel Programming Languages*. PhD Dissertation, University of Washington, February 2001.

[48] W. Derrick Weathersby. *Machine-Independent Compiler Optimizations for Collective Communication*. PhD Dissertation, University of Washington, August 1999.

[49] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the ACM International Conference on Supercomputing*, 2001.

[50] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. Factor-join: A unique approach to compiling array languages for parallel machines. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1996.

[51] Philip S. Abrams. *An APL Machine*, PhD Dissertation. Stanford University, SLAC Report 114, 1970.

[52] Ulrich Ruede. Mathematical and computational techniques for multilevel adaptive methods, *Frontiers in Applied Mathematics*, 13, SIAM, 1993.

[53] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[54] R. E. Ladner and M. J. Fischer. Parallel prefix computation. In Proceedings of the IEEE International Conference on Parallel Processing, 1977.

[55] Steven J. Deitz. *High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations*. PhD Dissertation, University of Washington, February 2005.

[56] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. High-level language support for user-defined reductions. *Journal of Supercomputing*, 23(1), 2002.

[57] E Christopher Lewis and Lawrence Snyder. Pipelining wavefront computations: Experiences and performance. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, May 2000.

[58] Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder. Problem space promotion and its evaluation as a technique for efficient parallel computation. In *Proceedings of the ACM International Conference on Supercomputing*, 1999.

[59] Calvin Lin, Lawrence Snyder, Ruth Anderson, Brad Chamberlain, Sung-Eun Choi, George Forman, E. Christopher Lewis, and W. Derrick Weathersby. ZPL vs. HPF: A Comparison of Performance and Programming Style, TR # 95-11-05 (available online from the University of Washington CSE technical report archive).

[60] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *9th International Workshop*

on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004), pp 52-60, April 2004.

[61] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *20th OOPSLA*, pp. 519-538, 2005.

[62] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, AlexWoo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS–95–020, Nasa Ames Research Center, Moffet Field, CA, December 1995.

[63] Bradford L. Chamberlain, Steven J. Deitz, and Lawrence Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Proceedings of the ACM Conference on Supercomputing*, 2000.

[64] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. *ACM SIGARCH Computer Architecture News*, 18(3):1 - 6, 1990.

[65] Michael A. Hiltzik. Dealers in Lightning: Xerox PARC and the Dawn of the Computer Age, Harper Collins, 1999.

[66] Eric Steven Raymond and Rob W. Landley. *The Art of Unix Usability*, Creative Commons, 2004. `http://www.catb.org/~esr/writings/taouu/html/ch02.html`.

[67] Calvin Lin and Lawrence Snyder. A comparison of programming models for shared memory multiprocessors. In *Proceedings of the IEEE International Conference on Parallel Processing*, 1990.

[68] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam and N. Shenoy. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM Transactions on Programming Languages and Systems* 21(6):1251-1297, 1999.

[69] A. J. Wagner, L. Giraud and C. E. Scott. Simulation of a cusped bubble rising in a viscoelastic fluid with a new numerical method, *Computer Physics Communications*, 129(3):227-232, 2000.

[70] Robert W. Numrich, John Reid, and Kieun Kim. Writing a multigrid solver using Co-array Fortran. In *Proceedings of the Fourth International Workshop on Applied Parallel Computing*, 1998.

[71] S. B. Scholz. A case study: Effects of WITH-loop-folding on the NAS benchmark MG in SAC. In *Proceedings of IFL '98*, Springer-Verlag, 1998.