AFRL-IF-RS-TR-2001-220
Final Technical Report
October 2001

MIGRATING MATLAB TO ZPL

University of Washington

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D515

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-220 has been reviewed and is approved for publication.

APPROVED: Joseph A Carozzoni

JOSEPH A. CAROZZONI
Project Engineer

FOR THE DIRECTOR: James A. Collins

JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

MIGRATING MATLAB TO ZPL

Lawrence Snyder

Contractor: University of Washington
Contract Number: F30602-97-1-0152
Effective Date of Contract: 1 January 1997
Contract Expiration Date: 31 December 1999
Program Code Number: D515
Short Title of Work: Migrating MATLAB to ZPL

Period of Work Covered: Jan 97 – Dec 99

Principal Investigator: Lawrence Synder
Phone: (206) 543-9265
AFRL Project Engineer: Joseph A. Carozzoni
Phone: (315) 330-7796

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Joseph A. Carozzoni AFRL/IFTB, 525 Brooks Road, Rome, NY 13441-4505.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 2001	Final Jan 97 - Dec 99

4. TITLE AND SUBTITLE

MIGRATING MATLAB TO ZPL

5. FUNDING NUMBERS

G - F30602-97-1-0152
PE - 62301E
PR - HPSW
TA - 00
WU - 03

6. AUTHOR(S)

Lawrence Snyder

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Washington
Grants and Contract Services
3935 University Way, NE
Seattle WA 98105-6613

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFTB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2001-220

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Joseph A. Carozzoni/IFTB/(315) 330-7796

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report documents the task of migrating MATLAB programs to ZPL so that the computations can run on parallel platforms and achieve significant performance improvements. It entails three tasks: Upgrade ZPL to support sparse representations, provide an interface to a parallel scientific library, and provide a mechanism by which programmers can know when their MATLAB programs have limited parallelism. The project achieved all three goals. The prototype was fully implemented and made available over the Internet. In benchmark tests ZPL programs were shown to perform as well as or better than programs written by experts using C and message passing. ZPL programs are fully portable running well on any UNIX platform. Additionally, the language is convenient, automatically producing all concurrency, all communication and very aggressive scalar optimizations. This research produced two dozen technical papers and four PhD dissertations.

14. SUBJECT TERMS

Data parallel programming, metacomputers, memory hierarchy simulator

15. NUMBER OF PAGES

16

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Migrating MATLAB to ZPL

Abstract

The task of migrating MATLAB programs to ZPL so that the computations can run on parallel platforms and achieve significant performance improvements entails three tasks: Upgrade ZPL to support sparse representations, provide an interface to a parallel scientific library, and provide a mechanism by which programmers can know when their MATLAB programs have limited parallelism. The project achieved these three goals, although the original proposal's plan to solve the latter problem on the "MATLAB side" was replaced by a more effective solution that solves it on the "ZPL side." The software is fully implemented and available free of charge over the WWW. There is a small ZPL user community. In benchmark tests ZPL programs are shown to perform as well as or better than programs written by experts using C and message passing. ZPL programs are fully portable running well on any UNIX platform. And the language is convenient, automatically producing all concurrency, all communication and very aggressive scalar optimizations. Additionally, a substantial amount of research was conducted into parallel languages, parallel compilation and compiler optimizations. This research produced two dozen technical papers and four PhD dissertations.

Overview

Generally, MATLAB is a forgiving, powerful and slow (to execute) means of expressing scientific computation. Generally, ZPL is an exacting, powerful and fast means of expressing scientific computations. The forgiving vs exacting and the slow vs fast tradeoffs are embodied in the differences between an interpreted vs compiled language. The goal of this research was to discover how to transition from the forgiving to the exacting in order to replace the slow with the fast. That is, to have the convenience and the speed too.

There are three fundamental challenges to converting MATLAB programs to ZPL programs. The first is the ability to execute the MATLAB language constructs efficiently. Since the effort began with ZPL already performing most of the MATLAB operations faster, in parallel and with more generality than MATLAB itself, the main challenge was MATLAB's support for sparse arrays. MATLAB was the first, and when this project started the only, language supporting sparcity. Its sparse matrices were largely transparent to the user, so the performance advantages were limited. The challenge of providing a language level sparse array capability that was both parallel and high performance had never been achieved. This project has accomplished this goal, as explained below. The solution not only provides fast, parallel sparse arrays as a fundamental data type of ZPL, but the technology goes well beyond what is available in MATLAB and is general enough to apply to any language, parallel or sequential. Thus, ZPL "covers" MATLAB in the sense of running all of its abstractions fast and in parallel.

The second challenge is to provide a parallel interface to library routines, since most of MATLAB's value comes from its convenient interface to powerful scientific software. The problem in the parallel context is that parallel scientific software is an enormous research area in its own right. We interacted with two of the most well known scientific software groups, Jack Dongarra's SCALapack group and Robert van de Geijn's PLAPACK group. Though ZPL can interface to both, and we have worked out the details for both, we have demonstrated our solution using PLAPACK. It is the work of a week for a SCALapack expert to interface to that library.

The third challenge concerns the fact that the MATLAB language is a sequential language, but to run it fast enough for serious scientific computations, it must be run in parallel. Being sequential means that

whenever programmers are not using the scientific software, i.e. when the are programming directly in the language, they are writing code that may or may not have efficient parallel execution. This problem was described in depth in the proposal, and it was noted there that it couldn't be solved. That is, to solve it is tantamount to claiming a "general purpose automatic parallelization" technology, which has been promised by many researchers for decades and not achieved. We believe that general automatic parallelization is not a realistic goal. So, the plan in the proposal was to create a programmer's aid that would identify those places in the converted program that were not parallel. Since it became obvious early on when NSF failed to provide the funding to match DARPA's that such an ambitious software project was not feasible, we have developed an alternative as part of our best effort. We have developed an abstraction called ZPL's WYSIWYG performance model [2], which enables programmers to have the information that an analyzer would normally have. In a sense this solution is superior to the "programmer's aid" because the information can be used both for creating ZPL from MATLAB as well as writing ZPL programs from scratch. The latter would have been impossible without a "ZPL side" solution.

So, the technical goals of the project -- support MATLAB's operations, support scientific libraries and handle the sequential nature of MATLAB program text -- have been achieved. In addition there has been a substantial addition to the capability of ZPL including,

- sparse regions
- Mscan
- problem space promotion
- advanced optimizations

These will be discussed below. Further, the project supported a dedicated cadre of users in applying ZPL to scientific problems, and received considerable feedback regarding practical applications. At the completion of this research, ZPL is a freely distributed parallel programming language capable of hosting MATLAB programs and running them in parallel for dramatic speed improvements. Interestingly, some MATLAB programmers have said that rather than converting, they'll take the opportunity to develop a new program directly in ZPL.

The remainder of this report gives technical substance to the topics raised in the Overview.

Sparse Regions and Arrays

During the 1990s the state-of-the-art in parallel algorithms improved dramatically, going from the naïve "dense" solutions so common previously to solutions involving much more sophisticated data structures, especially sparse arrays. Languages like Fortran 90/95 and High Performance Fortran require programmers to implement sparse structures manually. This is not only very difficult work for programmers, but the compiler is unable to determine what the program is actually doing, and so cannot perform sophisticated optimizations. MATLAB sought to help the programmer by constructing a "black box" sparse array that the programmer could declare but otherwise could not affect or be aware of how it was being used. ZPL through this award has created the first language level abstraction for sparse arrays, implemented it, shown how to compile it to run fast in parallel and demonstrated it on sparse benchmarks. This is a significant and fundamental accomplishment.

The key insight required to introduce sparse arrays into ZPL is to recognize that dense arrays are defined and transformed using *dense regions* [4]. Therefore, extending this notion to sparse arrays "only" requires the invention of *sparse regions*. Regions are index sets, and a powerful new idea in ZPL. For the dense index case, i.e. those common cases such as $n \times n$ arrays, regions are specified by giving their index range, as in

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

which specifies the n^2 set of indices from (1,1) through (n,n). Though this looks like an array definition in another language, it declares only the indices. The n x n arrays A, B and C could be declared from this region by

```
var A, B, C : [R] double;
```

which specifies that each array has n^2 elements and the elements are double precision floating point numbers data.

The sparse case is considerably more complicated [10]. First, there is the representation, which in the dense case requires only the lower and upper bounds, the stride and the starting position. In the sparse case a full data structure must be created to keep track of each represented item in the sparse structure (known commonly as a nonzero). Further, the structure must support all of the ZPL data traversals. This structure requires significant memory and so its aggressive optimization is essential or the program will suffer adverse cache affects. The other problem with sparse arrays is that the represented elements must be specified. This is sometimes static, as with tridiagonal matrices. Most commonly, the nonzeroes are known at the start of the computation and can either be computed at initialization time or read in from a file. The most dynamic case is when the configuration of the nonzeroes changes incrementally as the computation evolves. The current implementation handles the first two cases.

When measured on the NAS conjugate gradient benchmark, which has a programmer produced sparse data structure, the ZPL compiler is amazing [11]. It is able to match both the footprint, i.e. the memory usage, and the performance of a high quality parallel program. The source text for ZPL is trivial for the core sparse matrix-vector multiplication, whereas it runs to pages for the hand-coded version because of all of the communication.

The sparse array work is the core of Bradford L. Chamberlains dissertation research [10], and has recently appeared at an international conference [11]. It is too recent to see whether this will be incorporated into other programming languages, but it is sufficiently labor-saving from the programmer's point of view and sufficiently effective at enabling compiler optimizations that it is likely to be included in other future systems.

ZPL Release

Just six months after the start of the award the ZPL compiler was publicly released. Of course, most of the development was supported on previous awards, but the present award assisted in the distribution and user support, which was crucial to the feedback needed for the research. The free ZPL software, comprised of a compiler, libraries and documentation, was and remains the only high level parallel programming language that can claim performance, portability and convenience [8]. "Performance" in this claim means that the compiler produces from the high level source, object code that runs as fast as a program written by an experienced programmer in C with message passing, the present industry standard [9]. Recent comparisons reveal that even experienced programmers cannot write code that runs as fast as ZPL, even for a *sequential* computer. "Portability" means that ZPL runs well on *any* Unix/Linux platform, which includes contemporary parallel machines as well as all sequential machines. It is a fundamental fact of computer science (universality theorem) that any program can run on any computer, so the import of this remark is the "runs *well*" claim. Expect a well-written ZPL program to run well on every platform. [A serious effort was made to port ZPL to Microsoft's NT, but the effort eventually failed as the operating system is very difficult to work with where performance is concerned.] "Convenience" means that the programs are simple and clear. An example of one user's program required 2.5 pages to solve a multigrid combustion computation in ZPL and 12.5 pages in C with MPI message passing -- and the ZPL program ran more than twice as fast!

ZPL's release has attracted a small, but dedicated set of users. These users have not only made the compiler more robust by testing out its facilities, but they have provided the raw material for both the language design and the performance studies.

Mscan

One of the most pioneering advancements in the present compiler is the creation of a high-level programming abstraction for pipelining [6,7]. As is well known pipelining is one of the most powerful and

widely used forms of parallelism. However, no high-level parallel language supported it directly despite the fact that certain classic scientific computations, like solvers, must use pipelining to achieve any performance at all.

One serious limitation with introducing pipelining into an array language, say for wavefront computations, is that it is contrary to array language semantics. To accumulate the rows of an array one might wish to write, in ZPL style,

```
A := A + A@north
```

which seems like it should take the rows of array A and replace each with itself and the row above it, leading to an accumulating sum. However, array language semantics require that the right-hand side be evaluated entirely before the assignment. To get the desired wavefront motion, ZPL introduces the prime operator, so the correct alternative to the previous statement is

```
A := A + A'@north
```

which produces an accumulating sum and pipelines the result on parallel computers.

Though the prime operator is an example of applying commonly understood metaphors to achieve new results, it doesn't quite solve the problem, because most scientific computation is more complicated than a single statement. For that reason, the **mscan** keyword was introduced to allow pipelining across a range of statements. (**mscan** takes its name from "mighty scan", the term used in Ton Ngo's thesis, where the idea was invented [12].) The fundamental research to incorporate pipelining into ZPL and other languages was the PhD dissertation of E Chris Lewis [13].

Advanced Optimizations

One of the fundamental rules that releasing the compiler to the public taught the ZPL team was that great parallel performance is useless unless great scalar performance is also achieved. That is, even if the processors are working well together -- and ZPL is outstanding at achieving that -- the efficiency of the computation on an individual processor is just as important if performance is to eclipse programmer-produced code. For that reason the team has worked intensively at both parallel optimizations and sequential optimizations. Most of this work is published, but an enumeration of it here is useful.

- **Communication optimizations** -- the dissertation topic of Sung-Eun Choi [15] shows how ZPL can optimize interprocessor communication to achieve better-than-message passing performance [14]. A key aspect of the approach is the Ironman communication abstraction. The bottom line result of this dissertation is that well designed compilers are more effective that humans at inserting interprocessor communication, raising the question "Why is message passing so popular?"

- **Fusion and Contraction** -- scalar language compilers create temporaries to hold intermediate results, but when an array language does it there is a significant impact on storage. Removing this problem was an important goal of the project because the temporaries ruin cache performance, a key advantage of parallel machines that should not be lost. The net result is that an aggressive compiler (ZPL) can remove not only the temporaries introduced by the compiler but also those introduced by the programmer [3, 13].

- **Collective Communication Optimizations** -- Parallel computations require such things as global sums, known as "collective" operations. The communication patterns for these are quite different than those for other operations, so it makes since to try optimizing them. This was the task of Derrick Wethersby's dissertation [16], which showed that combining and pipelining were powerful techniques to reduce the wait times and overheads for communication in collective operations.

Other less grandiose optimizations have been incorporated in the compiler, though they have not lead to dissertation research.

Problem Space Promotion

Part of the challenge in parallel language and compiler design is to determine how a problem should be solved in parallel in the first place. Once this is known then the concepts can be incorporated into the language and the compiler can be designed to produce the code. One technique is Problem Space Promotion. The idea is that computations are usually solved in the "dimensionality" in which they are represented, e.g. matrix multiplication is solved in 2-dimensions because matrices are 2-dimensional. But, it is often possible to specify *what* is to be computed by raising the dimensionality of the solution and thereby avoid over-specifying *how* it is to be computed. Without over-specifying the compiler has more latitude to create an efficient solution. So, matrix multiplication can be solved in 3-dimensions by thinking of each operand array as being replicated n times (n is the common dimension), the corresponding elements multiplied elementwise and then the dimensionality reduced by summing along the common dimension. The result is a specification of matrix product with only computationally required dependencies given. PSP opportunities arise repeatedly [5].

The idea of PSP computations seems clear, but with so much latitude, it is complicated for the compiler to figure out how best to solve the promoted problem. The team took on as the goal to do as well as an expert programmer, which amounts to avoiding the creation of higher dimensional intermediate arrays If this happened it could often overflow memory, since for example, multiplying 1000x1000 arrays of doubles, requires an intermediate array of 8GB. But, even when it doesn't overflow the memory, it will surely overflow the cache, an equally bad outcome. The ZPL compiler generates code for "problem space promoted" computations that achieves both efficient intermediate memory usage as well as high performance [3].

What You See Is What You Get

The development of the WYSIWYG model of parallelism turned out to be critical to enabling MATLAB programmers to know when their corresponding ZPL programs would have limited parallelism. But, the original purpose of WYSIWYG [2], was to write good parallel programs from scratch. This is a pioneering idea, and it works like this.

When programmers write in C or Fortran they believe they "know" what the generated code will look like. In actuality, they are often surprised because aggressive compilers often transform source code tremendously, but that's not the issue. The point is that programmers "know" because there is a standard model of sequential computers (von Neumann) and the model tells them how efficiently their program will run. (The compiler's transformations are improving on this, so their understanding is the worst-case performance.) In the parallel world only ZPL has adopted a standard model, the CTA model. In the same way that the von Neumann model tells Fortran programmers how their code will run, unless the compiler will do better, the CTA tells ZPL programmers how well their program will run, unless the compiler can do better. The CTA concentrates on those features like interprocessor communication and latency that are peculiar to parallel computers, leaving the details of the scalar processor to the von Neumann model.

Though this appears to be an amazingly obvious requirement for parallel programming success, it is not a property of any other parallel programming language. Further, it cannot be a property of any programming approach based on message passing. To note how well it works, the project members took two standard matrix multiplication problems and wrote them in ZPL. The programs were quite different, of course, but using the WYSIWYG model, it was possible to do a back-of-the-envelop analysis of which program would run faster. A MATLAB programmer would do this. Once completed, a series of experiments across a series of parallel programs showed that the WYSIWYG performance prediction was, indeed, true [2]. As always, *the ability to correctly predict an outcome is the hallmark of quality science.*

Summary

As a result of this award it is now possible to migrate programs from MATLAB to ZPL. If the programs use the sparse features of MATLAB, then the sparse features of ZPL will be used. In addition to the basic

goals of the project, a large body of associated and related scientific research were also created. Four graduate students wrote doctoral dissertations under its auspices. All of the features of this report are implemented and are available free of charge to the community.[1] A small cadre of programmers uses ZPL routinely.

References

[1] G. Alverson, W. Griswold, C. Lin and D. Notkin, Lawrence Snyder, "Abstractions for Portable, Scalable Parallel Programming," IEEE *Transactions on Parallel and Distributed Systems*. 9(1):71-86, 1998.

[2] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby, "ZPL's WYSIWYG Performance Model," Proceedings of the IEEE Workshop on *High-Level Parallel Programming Models and Supportive Environment*, August 1998.

[3] E Christopher Lewis, Calvin Lin and Lawrence Snyder, "The Implementation and Evaluation of Fusion and Contraction in Array Languages," In Proceedings of the ACM SIGPLAN '98 Conference on *Programming Language Design and Implementation* (PLDI), August 1998

[4] Bradford L. Chamberlain, E Christopher Lewis, Calvin Lin and Lawrence Snyder, "Regions: An Abstraction for Expressing Array Computation," Proceedings of the 1999 SIGPLAN/SIGAPL International Conference on Array Programming Languages, pp. 41-49, August 1999.

[5] Bradford L. Chamberlain, E Christopher Lewis and Lawrence Snyder, "Problem Space Promotion and Its Evaluation as a Technique for Efficient Parallel Computation," Proc. 13th International Conference on Supercomputing, pp. 311-318, June 1999

[6] Bradford L. Chamberlain, E Christopher Lewis and Lawrence Snyder, "Array Language Support for Wavefront and Pipelined Computations," Proc. Workshop on Languages and Compilers for Parallel Computing, August 1999.

[7] E Christopher Lewis and Lawrence Snyder, "Pipelining Wavefront Computations: Experience and Performance," *Proc. 5th IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments*, May 2000.

[8] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder and W. Derrick Weathersby, "ZPL: A Machine Independent Language for Parallel Computers," *IEEE Transactions on Software Engineering*, March 2000.

[9] Bradford L. Chamberlain, Steven J. Deitz, and Lawrence Snyder. "A Comparative Study of the NAS MG Benchmark across Parallel Languages and Architectures." Proceedings of the 2000 *ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC2000)*, November 2000.

[10] Bradford L. Chamberlain, *The Design and Implementation of a Parallel Programming Language*, PhD Dissertation, University of Washington, (to appear) 2001.

[11] Bradford L. Chamberlain and Lawrence Snyder. "Array Language Support for Parallel Sparse Computation. To appear in *Proc. 15th ACM International Conference on Supercomputing (ICS'01)*, June 2001.

[1] The software is huge and is not included with this report. See the ZPL Web site for a copy all project materials: http://www.cs.washington.edu/research/zpl/

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

[13] E. Chris Lewis, Achieving Robust Performance in Parallel Programming Languages, PhD Dissertation, Universrity of Washington, 2001

[14] Sung-Eun Choi and Lawrence Snyder, Quantifying the Effects of Communicaton Optimizations, International Conference on Parallel Processing, pp. 218-222, August 1997.

[15] Sung-Eun Choi, Machine Independent Communication Optimizations, PhD Dissertation, University of Washington, 1999

[16] W. Derrick Wethersby, Machine Independent Compiler Optimizations for Collective Communication, PhD Dissertation, University of Washington, 1999

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.