A High Productivity Framework for Parallel Data Intensive Computing in MATLAB

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Rajkiran Panuganti, B.Tech, M.S.

* * * * *

The Ohio State University

2009

Dissertation Committee:
Dr. P. Sadayappan, Adviser
Dr. A. Rountev
Dr. D. K. Panda
Dr. S. Krishnamoorthy

Approved by

Adviser
Graduate Program in
Computer Science and
Engineering
© Copyright by

Rajkiran Panuganti

2009
ABSTRACT

Programmer’s productivity is recognized as one of the most significant challenges in the effective use of high-performance computing. Higher level languages like Matlab are being increasingly adopted. however, it has significant shortcomings when used for large-scale computationally intensive applications that require very high performance and/or significant amounts of memory. Developing efficient runtime frameworks to aid the high-level languages to make them scalable to larger problem sizes is an effective solution to this problem. Our solutions, mexMPI, GAMMA and LA enable parallel computing directly in Matlab for high-performance while retaining its productivity aspects. mexMPI provides message passing semantics to enable parallel computing within Matlab environment using high-performance networks. GAMMA presents a distributed shared memory programming model wherein the programmer develops his/her parallel algorithms using a ‘Get-Compute-Put model’ and ‘LA’ is a runtime framework that enable users to develop large scale applications directly in Matlab. We demonstrate the effectiveness of our frameworks using NAS benchmarks. The experimental evaluation of these frameworks indicate effectiveness of our approach.
To my mother, Dr.(late) P. Krishna Kumari, my father, Dr. P. Amareshwar and my brother Dr. P. Chakravarti
ACKNOWLEDGMENTS

The journey of obtaining a doctorate was undoubtedly the longest one in my life. It was a journey that taught me lessons that graduated me from an immature boy to a matured man. This long and precarious journey wouldn’t have been possible without the guidance, encouragement and support several people with whom I shared important sections of my space and time.

I would first like to thank and dedicate my thesis to my mother Dr. P. Krishna kumari, my father Dr. P. Amareshwar and my brother Dr. P. Chakravarti for providing me an immense inspiration to pursue this journey towards doctorate. Particularly, I would like to thank my mother whose constant pursuasion has made me choose this challenging line of career. She has left this world after ensuring that I would be starting my final goal in my academic career. I would also like to thank my father whose has been my primary teacher in my early part of the life. I also like to thank my brother for numerous immemorable time we have shared with each other and who has lived through numerous spanks of a typical younger brother.

In Dr. P. Sadayappan, I have found a perfect advisor one could ask for. I’m greatly indebted to him for guiding me through this journey. Providing just the right amount of guidance and ample space and time to overcome the ups and downs that one would go through in this strenuous journey. His astonishingly sharp mind, and a gentle personality has been an inspiration to many of his students.
I would also like to thank many individuals who have provided ample guidance in this course. Dr. Rountev particularly has been responsible in helping me to take my immature ideas to formalising them into mature ones. I would also like to thank Dr. Arun Chauhan, Dr. Ashok and Dr. Robert Schrieber for the guidance they have provided at appropriate junctions. I would also like to thank Dr. Subhasis Chaudhary who has introduced me to the field of research.

During this journey, I have gained immense memories that would live with me for the entire life. Gaurav Khanna, Sitaram Asur, Muthu Baskaran, Vijay Kumar, Amol Ghoting, Sirish Thatikonda, Swarup Sahoo, Abhinav Vishnu, Sriram Krishnamoorthy. Gaurav, in particular with whom I’ve shared over four years of social and academic life has been a valuable co-passenger in this journey. The moments I’ve played cricket with them have been my primary source of extra curricular activity. I would also thank Alok, Rudraksh, Saurabh, Rakesh, Harsh, Yogesh, Anant, Shashank, Archana, Tripti, Fehmida, Sabrina, Kanika, Kavita, and Shilpa with whom I’ve shared valuable time in Columbus. I would particularly like to thank Alok who has been a close confidante all through the years from my life at IIT to life at Columbus. I would also like to thank my numerous friends Rahul, Panda, Gnana Kiran, Harish, Avinash, Ruchin, Saurabh, Hemal and Amit. All of them have combinedly shaped both my academic and non-academic life for the past twelve years. I would also like to thank Tripti, Nithin, Vinay, Sudha, Sandeep, Anupama, Siddarth and Ravi with whom I have gained many immemorable moments during this journey. My life in a country away from home was riddled with many moments feeling nostalgic. My trips to my relatives, Vamshi, Surender, Sujatha and their lovely kids Sumona and Sumedh effectively helped me feel at home in this country.
I would like to specially thank the institution of IIT Bombay to which I would be indebted for my entire life. It is an institute that inspires millions of Indians to pursue technology as a career. I would like to thank everyone who is involved in building such a wonderful institution.

Lastly, I would like to thank OSU and OSC for providing me an opportunity to pursue my PhD and for providing ample financial support.
VITA

April, 1982 ................................. Born - Warangal, India

2002 ........................................B.Tech. Electrical Engineering
Indian Institute of Technology - Bombay,
mumbai, India

2008 ........................................M.Sc Computer Science Engineering
Ohio State University,
Columbus, OH

June 2006 - Sep. 2006 ...................... Research Intern,
Hewlett-Packard Research Labs.

June 2007 - Sep. 2007 ...................... Engineering Intern,
Google Inc.

2003-2004 .................................. Graduate Teaching Associate,
The Ohio State University.

2004-2009 .................................. Graduate Research Associate,
The Ohio State University.

PUBLICATIONS

Rajkiran Panuganti, Muthu Manikandan Baskaran, Ashok Krishnamurthy, Jarek Nieplocha, Atanas Rountev and P. Sadayappan. An Efficient Distributed Shared Memory Toolbox for MATLAB 12th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), held in conjunction with International Parallel and Distributed Processinig Symposium (IPDPS) 2007

Swarup Kumar Sahoo, Rajkiran Panuganti, Sriram Krishnamoorthy, P. Sadayappan
Cache Miss Characterization and Data Locality Optimization for Imperfectly Nested
Loops on Shared Memory Multiprocessors  19th IEEE International Parallel and
Distributed Processing Symposium (IPDPS), 2005

Swarup Kumar Sahoo, Sriram Krishnamoorthy, Rajkiran Panuganti, P. Sadayappan
Integrated Loop Optimizations for Data Locality Enhancement of Tensor Contraction
Expression  International Conference on Supercomputing (SC), 2005

FIELDS OF STUDY

Major Field:  Computer Science and Engineering

Studies in High Performance Computing:  P. Sadayappan
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
</tbody>
</table>

## Chapters:

1. Introduction ............................ 1
   1.1 Motivation ................................ 1
   1.2 Programmability .......................... 3
   1.3 Performance .............................. 3
   1.4 Memory Intensive Computing in Matlab .......... 5
   1.4.1 Overview of Our Approach ................. 6
   1.5 ParaM .................................... 7
   1.5.1 Matlab for High Performance Computing .... 7
   1.5.2 Matlab for Memory Intensive Computing .... 9
   1.6 Organization of the thesis .................. 10

2. Related Work .............................. 11
   2.1 High Performance Computing in Matlab .......... 11
   2.1.1 Compilation Approach ...................... 11
   2.1.2 Embarrassingly Parallel .................... 12
5.4.4 Compactness ........................................ 62
5.5 Experimental Evaluation ............................... 65

6. Large Arrays .............................................. 70
   6.1 Motivation ........................................ 70
   6.1.1 Overview .................................. 71
   6.2 Programming Model ............................... 72
   6.3 Execution Model ................................. 73
      6.3.1 Transparent framework .................. 73
      6.3.2 Efficient Execution ..................... 74
   6.4 Analyzer .......................................... 75
      6.4.1 Parametric Trace Generator ............... 75
      6.4.2 Library ................................ 77
   6.5 Prefetching .................................... 79
   6.6 Caching .......................................... 80
   6.7 Experimental Evaluation ....................... 80
      6.7.1 Experimental Setup ..................... 81
      6.7.2 Programmability ......................... 81
      6.7.3 Performance Analysis ................... 82
   6.8 Related Work .................................. 83

7. Conclusions ............................................ 86

Bibliography ............................................. 113
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Lines of source code for the NAS benchmarks</td>
<td>33</td>
</tr>
<tr>
<td>4.2 Speedup: NAS, Class A</td>
<td>35</td>
</tr>
<tr>
<td>4.3 Speedups using reference based semantics to access local portions of the global arrays for Jacobi iterative solver</td>
<td>37</td>
</tr>
<tr>
<td>4.4 Speedup: NAS, Class B</td>
<td>38</td>
</tr>
<tr>
<td>5.1 Trace Compression comparison of TraceAnalyzer with that of NLR and VPC (in bytes)</td>
<td>65</td>
</tr>
<tr>
<td>6.1 Lines of source code for the NAS benchmarks</td>
<td>82</td>
</tr>
<tr>
<td>6.2 Execution time of NAS Benchmarks with LA</td>
<td>82</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>NAS Benchmarks illustrating Matlab’s limitations</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>Comparison of mexMPI and MPI semantics</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>SLOC comparison between mexMPI and C/Fortran+MPI</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>mexMPI vs C/MPI</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>NAS IS with mexMPI</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Software architecture of GAMMA</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Global view of a Distributed MATLAB Array</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>Get-Compute-Put computation model</td>
<td>25</td>
</tr>
<tr>
<td>4.4</td>
<td>Parallel 2D Fast Fourier Transform in GAMMA</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>Illustration of ‘Distributed Arrays’ toolbox</td>
<td>31</td>
</tr>
<tr>
<td>6.1</td>
<td>Structure of a ‘Phase’</td>
<td>78</td>
</tr>
</tbody>
</table>
List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Infer</td>
<td>41</td>
</tr>
<tr>
<td>2</td>
<td>Progress</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>Find Transition</td>
<td>67</td>
</tr>
<tr>
<td>4</td>
<td>Backtrack</td>
<td>68</td>
</tr>
<tr>
<td>5</td>
<td>Infer for Imperfectly Nest Loop Structure</td>
<td>69</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 Motivation

The demand for computational power has risen dramatically in the past several years. Computational power of the hardware has also grown exponentially to meet the demands of much of the scientific and technological research. However, this increase in computational power has occurred at the cost of an increased complexity in both software and hardware. Programming of these large scale scientific applications for high-performance still remains a daunting task. The problem has become significant due to a rapid increase in the number of applications requiring high performance. This has been highlighted by DARPA High Productivity Computing Systems (HPCS) program [12] and it identifies this problem as one of the most significant challenges in the effective use of high-performance computing. In this context, languages like Matlab, Python etc. are emerging to improve the productivity of programmers. Matlab in particular, has established as the most preferred language for technical computing. However, these languages are not yet well equipped to cater to applications requiring very high performance. Hence, it is important to provide suitable support to these high productivity languages to be able to deliver high performance.
Design of any high productivity framework needs to address the three P’s of productivity namely,

- **Programmability**: Programmability denotes the ease of programming. Good programmability can be achieved by enabling users to program using higher level of abstractions, with a simple programming model. This is extremely important as the cost of a software is being increasingly dominated by the man-hours required to develop, maintain and upgrade the software.

- **Performance**: Many scientific applications often deal with huge amounts of data and require significant computation power. Performance is extremely important for wide-scale acceptance of any software especially among the scientific domain. A high productive framework must satisfy this as an essential feature.

- **Portability**: Portability is also another important feature of a high productive framework. This is important as rewriting software for numerous and ever changing hardware and software platforms involves huge development, maintenance and upgradation costs.

Matlab successfully addresses the programmability and portability aspects. However, it fails to address the performance aspect effectively. The problem is even more severe for applications with large problem sizes. Matlab also does not have good support for out of core programming. Design of runtime frameworks that can handle efficiently handle applications with large problem sizes without a compromise on either programmability or portability of Matlab would effectively solve the problem.
1.2 Programmability

Applications developed using C and Fortran can scale well to large scale problems, but their creation remains a difficult task as the programmer has to explicitly manage additional hardware resources (e.g., different threads of execution, movement from disk to memory etc.) and to orchestrate the interaction between various components. While the manual tuning of applications is the predominant practise, programmers are required to spend a large amount of time in tuning applications for different architectures. Further, as the architecture changes, the process of tuning has to start all over again. This problem has become particularly important with rapidly changing computer architecture to meet the rapidly increasing computational demands of the scientific applications. Due to the above mentioned reasons, there is an increasing recognition that high-level languages, and scripting languages in particular such as Matlab and Python, can provide enormous productivity advantages to programmers. An easy to use programming model with array-based semantics, powerful built-in visualization facilities, and an integrated development environment make Matlab a highly productive environment for the scientific programmer. Another key aspect for the success of Matlab has been the availability of a variety of domain-specific ‘toolboxes’ (library components) for various fields. These toolboxes have been specialized for various domains and caters all the most commonly used functions in that domain.

1.3 Performance

Matlab is used extensively in the scientific domains due to its productivity advantages. However, sequential Matlab currently cannot deliver to the computational demands of many compute-intensive scientific applications. In a scenario when more
and more of these applications are requiring performance beyond what a single computer can deliver, it is extremely important that these languages are extended to enable to be run efficiently on cluster environments. Figure 1.1 highlights this problem by comparing the execution times of the sequential NAS benchmarks implemented using Matlab and using traditional languages (C/Fortran) on a single node. It can be seen that Matlab is unable to deliver any comparable performance in relation with applications written in C/Fortran and in many cases, the execution time is higher by a factor of over 10X.

Figure 1.1: NAS Benchmarks illustrating Matlab’s limitations
1.4 Memory Intensive Computing in Matlab

Many scientific applications have extreme memory requirements which cannot be satisfied by the amount of memory available on a single computer. The problem is even more severe in the context of Matlab where the interpreter requires a significant chunk of memory. This makes numerous applications like weather climate modeling, satellite data processing, large scale matrix operations etc. unable to execute in Matlab unless programmers explicitly use other modes of storage. Figure 1.1 highlights Matlab’s inability to scale well to large problems. Due to this reason, its use is restricted only to prototyping these applications with smaller problem sizes. Programmers are hence required to explicitly use disk as an additional level of storage of data during the computation. A typical memory intensive computation in such context would involve rewriting the code in such a manner that the computation is performed on smaller chunks of data that can fit in memory. These chunks of input data are read from the disk and the output of the computation is written explicitly to the disk. Hence, the entire computation might involve numerous I/O accesses. However, frequent accesses to the disk would significantly slow down the computation. This is even more significant as the gap between processor and I/O speeds is ever increasing. Further, writing an out-of-core program imposes additional burden of programming like explicitly handling the movement of data between the disk and memory, index translation etc. This significantly affects the programmer’s productivity. Further, in the context of Matlab, productivity is also a significant problem.
1.4.1 Overview of Our Approach

Numerous approaches have been actively pursued in the literature to address the problem highlighted in the previous sections. Broadly these can be categorized into compilation based approaches and runtime approaches. Each of the projects that are based on these approaches have been discussed in detail in the Chapter 2. Our approach is to build specialized toolboxes to address each of the limitations of Matlab. While providing a completely transparent framework is ideal, it is extremely difficult to achieve good performance with such a framework. Our approach is based on semi-transparent approach wherein our toolboxes take away most of the regular grudging work from the programmer either by providing efficient higher level abstractions and/or by tuning the applications based on the inputs provided by programmer. The user is required to program using our toolboxes which provide a highly productive environment. Our framework ensures that the applications written using our toolboxes are executed efficiently. Further, the programmer is also able to use sequential Matlab’s toolboxes(libraries) wherever possible. The toolboxes have been built to provide a highly productive ‘Get-Compute-Put’ model.

ParaM consists of the mexMPI, GAMMA and LA toolboxes. mexMPI and GAMMA enable parallel computing in Matlab and ‘LA’ enables developing applications requiring large problem sizes. mexMPI and GAMMA provide higher level abstractions making parallel computing easier. Numerous optimizations have been incorporated into mexMPI and GAMMA to make an it efficient toolbox. ‘Large Arrays’ toolbox on the other hand, uses a semi-automatic method to tune the prototyped applications. The frameworks utilizes the profiling data collected during the prototype runs to build a generator which helps in optimizing the execution of the application with parameters
corresponding to large problem size. Applications developed using our toolboxes can achieve better performance over other runtime frameworks in this domain. Further, support for higher level abstractions and the ‘Get-Compute-Put’ programming model enables higher productivity to the programmer. Further, the applications developed using our toolboxes are portable across various platforms.

1.5 ParaM

ParaM (‘Parallel Matlab’) [33] project at the Ohio Supercomputer Center (OSC) is an effort to build a high productivity framework to enable Matlab programmers to build production applications directly in Matlab. The fundamental goal of the ParaM project is the development of an environment that will enable Matlab users to develop large-scale, high-performance applications. In this proposal, we describe all the components of the ParaM effort, namely, mexMPI, GAMMA and LA. These components combinedly address the two significant issues, namely ‘performance scalability’ and ‘memory scalability’ for successful adaptation by the scientific community beyond its prototyping use. ParaM’s approach has been broadly to provide runtime support to Matlab to enable it to efficiently run large scale scientific applications in parallel environment. Each component of ParaM provides an easy-to-use programming model to the programmer by automatically handling all the system related aspects. These components collect profiling data from the prototype runs to optimize for the actual run.

1.5.1 Matlab for High Performance Computing

Scientific applications often deal with huge amounts of data and require computational power beyond what a typical personal computer can deliver. Parallel execution
in a distributed environment over commodity clusters is the most common solution to this problem. Efficiency and scalability in parallel programming requires good utilization of the high performance network architectures like Myrinet, Infiniband, Quadrics etc. However, efficient parallel programming is again considered to be extremely difficult. Message passing programming model has been widely accepted, and MPI in particular has established itself as the de-facto standard for parallel programming. Various flavors of MPI have been able to cater to the performance requirements of the scientific community. Though programming with MPI enables portability, various low level aspects and specially the granularity of a message in MPI makes it extremely difficult to program. Message passing interface with support for higher level data structures would make it an appropriate programming model for use with Matlab. In Chapter 4, we describe our ‘message passing’ programming model, mexMPI which is an adaptation of the MPI programming model with additional support for higher level data structures like matrices.

GAMMA

Programming for distributed environment, and managing separate memory for each of the individual threads of execution makes programming on distributed memory machines particular difficult. In the world of parallel programming, alternatives to message passing have been proposed to improve programmer’s productivity. One successful example is the global shared-address space (GAS) models, which are generally easier to program than message-passing models. However, achieving scalable GAS performance has been a difficult challenge. A number of efforts have targeted the development of scalable shared-memory models for parallel computing [43]. One
of the notable successes with GAS models is the Global Arrays (GA) suite [31] developed at Pacific Northwest National Laboratory. In Chapter 4, we describe our programming model, ‘GAMMA’. GAMMA is an adaptation of the Global Arrays library suite for use with Matlab. GAMMA interoperates with mexMPI to provide additional flexibility wherever required. GAMMA exposes a non-uniform memory access (NUMA) model and the programmer is encouraged to program applications in a blocked manner.

1.5.2 Matlab for Memory Intensive Computing

Many scientific applications also deal with huge amounts of data and/or operate on large scale matrices. They are often required to read/write data from/to the disk. Matlab however, does not scale well for large scale problem sizes. It is significant to solve the memory scalability issues in order for Matlab to be used beyond prototyping. The predominant practical solutions for this problem so far has been either to add huge amounts of memory or to spend a large amount of time in manually developing out-of-core application in C/Fortran. We propose a high level framework, ‘LA” that enables efficient out-of-core programming directly in Matlab would extend Matlab’s use beyond prototyping. The LA framework automatically manages various system resources and developing mechanisms to hide/reduce the I/O latency. The framework automatically performs prefetching and caching thus significantly reducing the bottleneck of I/O latency in such applications. Chapter 6 describes the ‘LA’ framework and the programming model in detail and identifies important issues to
be solved for an acceptable adaptation by the scientific community. Our ‘LA’ framework presents an easy-to-use programming model automatically handling most of the grudging out-of-core computation aspects from the programmer.

1.6 Organization of the thesis

Chapter 4 describes mexMPI and GAMMA in detail and presents performance results to demonstrate their effectiveness. An overview of the programming model is presented and a discussion about various features that ensure development of efficient applications is presented. A brief description of the implementation is also discussed in the chapter. Chapter 6 describes ‘Large Arrays’ framework and Chapter 7 concludes the thesis and highlights some important issues that can be pursued in future. A discussion of the programming model and an overall approach to optimizations have been presented. A set of experimental results have been presented to highlight the effectiveness of the approach. A discussion of various efforts to address the performance and memory limitations of sequential MATLAB is presented in Chapter 2.
CHAPTER 2

RELATED WORK

The popularity of MATLAB has motivated various research projects to address its performance and memory limitations.

2.1 High Performance Computing in Matlab

There have been numerous promising projects to address performance limitations of Matlab. These projects vary widely in their approaches and functionalities. Broadly, these efforts can be classified into the following categories [10].

2.1.1 Compilation Approach

One of the key performance overheads of MATLAB is due to its interpreted environment. Many projects such as Otter [34], RTExpress [36], FALCON [35], CONLAB [14], MATCH [5], Menhir [9], and Telescoping Languages [8] use a compilation-based approach. Fortran and then using established compilers to compile into native code. MaJIC [3] provides an interactive MATLAB-like frontend and compiles/optimizes code employing just-in-time compilation. Some projects have also attempted to generate parallel versions of the transformed code using automatic parallelization compilers. However, compilation approaches for Matlab require efficient implementations,
in the target language, of the numerous and ever increasing Matlab's proprietary built-in functions. Furthermore, whenever newer constructs are added in MATLAB, the compilation approaches require fundamental changes in their implementation to support the newer features. (For example, different versions of MATLAB have used different syntax for function definitions and have introduced new features such as nested functions, etc.)

2.1.2 Embarrassingly Parallel

Research projects such as PLab [23] and Parmatlab [27] provide support for embarrassingly parallel applications in MATLAB. These approaches use master/slave model and the data is divided onto different slaves. Each slave works only on its local data and sends the result to the master process. However, this approach severely limits the type of applications that can be parallelized and are not applicable to wide range of applications.

2.1.3 Message Passing

Projects such as MultiMATLAB [42], MPITB [4], MatlabMPI [20], and Mathworks’ DCT 2.0 [39] add message passing primitives to MATLAB. With this approach, users have maximum flexibility to build their parallel applications using a basic set of communication primitives. Further, this approach also enables users to use sequential Matlab libraries for the computation part of the code. MatlabMPI uses a file-based mechanism for communication, while MPITB builds upon LAM/MPI communication library. However, development using message passing model requires significant developmental effort for efficiently implementing the parallel libraries and applications and defeats the productivity advantages of using MATLAB.
2.1.4 Global Address Space Frameworks

Projects such as DLAB [32], Star-P [11] provide completely transparent frameworks for parallel computing. This approach uses a client-server model where a MATLAB session acts as a front end (client) and the computation is done using a backend parallel computation engine (server). While it is the ideal programming model, these approaches require the implementors to develop parallel implementations for all the numerous Matlab’s functions and hence the user’s are limited by the library routines that have been overloaded. To address this issue, Star-P provides a software development kit (SDK) which requires library writers to program in one of the traditional languages (C++) using MPI and then link them to Matlab, which is obviously very labor intensive. These approaches have shown significant promise on shared memory architectures. However, their scalability suffers significantly on distributed memory architectures. All of these approaches provide a special Matlab class of distributed arrays and some overloaded (parallelized) functions for these distributed arrays. None of the above efforts support irregular distributions in multiple for distributed arrays due to the complexity involved in the development of efficient libraries for irregularly distributed arrays.

Projects like pMATLAB [41] and Mathworks’ DCT 3.0 [39] provide a global address space programming model. The advantages of this approach over the message passing approach is that they provide a global view of the individual arrays and transparently performs the transfer of data from the owner processes to the target processes. These approaches provide a mixed programming model wherein a user can also program using message passing semantics wherever library routines are not available. Fraguela et al [16] provide a new class called Hierarchically Tiled Arrays

13
(HTAs) in MATLAB. All of these approaches also provide a special Matlab class and
overloaded operators and few functions for this special class. pMATLAB uses Mat-
labMPI for the implementation of the overloaded functions. DCT also uses message
passing infrastructure to develop parallel libraries for distributed arrays.

2.2 Programming for Large Problem Sizes

Matlab’s another significant limitation has been its inability to scale to large
problem sizes. Matlab provides routines to write and read entire arrays to the disk.
Further, it also provides routines to read and write data to a file. Matlab users are thus
required to explicitly handle the I/O routines to develop out-of-core programs. There
have been numerous projects addressing the productivity aspects in developing large
scale applications. These projects can be broadly classified into virtual memory based
and frameworks supporting out of core programming. Virtual memory approaches
have been reasonable popular for general computing. However, it is not yet effective
for scientific applications requiring high performance. Out-of-core approaches have
been popular from early stages [15] and numerous projects have attempted to address
the productivity and performance issues in out-of-core programming.

pMATLAB-XVM [22] is an extension to pMatlab providing transparent support to
write parallel out of core applications. pMATLAB-XVM provides a highly productive
environment transparently handling distribution of data. The data on each node
is divided into in-core and out-of-core blocks. The programming model supports
execution at the core block level. These blocks are exchanged between the processes
using MatlabMPI’s file-based communication mechanism. Further, for operations on
the same node across the core blocks, operations are performed on one block after
another. However, the framework does not perform any of the common optimizations used for hiding the significant I/O latency due to its pure Matlab implementation hence suffers significant performance drawbacks for many applications.
CHAPTER 3

MESSAGE PASSING TOOLBOX

3.1 Motivation

Many scientific applications require computational power beyond what a single computer can deliver. Further, the computational power delivered by Matlab on a single computer is limited due to the overheads of interpretation. In such a scenario, parallel computing is the common solution to leverage higher computational power. It is important that programmer’s in Matlab environment are able to use high-performance networks with an easy-to-use programming model to perform computations in a distributed environment. In this chapter, we present our toolboxes, mexMPI and GAMMA which enable parallel computation directly in Matlab.

3.2 Message Passing

Message passing has been the most successful programming model for parallel computing. This is primarily due to the flexibility it provides in enabling programmers to achieve maximum scalability. However, programming using message passing involves great amount of development time. Most of the grudging work in programming using C/MPI is primarily due to a ‘distributed’ view of the data. The book-keeping
work necessary in this programming model consumes most of the development effort. We present our toolbox, mexMPI which is an adaptation from the MPI, to enable parallel computing in Matlab. mexMPI supports all the data structures in Matlab enabling message passing at a higher granularity and also handles most of the book-keeping work.

### 3.3 Related Work

mexMPI has been built as a Matlab toolbox using the MVAPICH [26] toolbox. Figure 3.1 presents a brief comparison of semantics between mexMPI and C/MPI. It can be clearly observed that mexMPI handles most of the book-keeping work, including the size, type of the message while retaining the flexibility provided by the message passing model. The semantics of mexMPI are derived from MPI based on the principle that any aspect of the semantics that can be automatically derived should be handled automatically. It is this principle that reduces the book-keeping code that the programmer had to maintain.
3.4 Performance Evaluation

3.4.1 Experimental Testbed

The experiments were conducted on the Ohio Supercomputer Center’s Intel Pentium 4 cluster constructed from commodity PC components running the Linux operating system. The hardware and software configuration of the cluster is as follows: two 2.4 GHz Intel P4 Xeon processors on each node; 4GB RAM on each node; InfiniBand interconnection network; Red Hat Linux with kernel 2.6.6; MATLAB Version 7.0.1.24704 (R14) Service Pack 3. All experiments were conducted such that no two processes were on the same node in the cluster, ensuring that the parallel processing environment is fully distributed.
Figure 3.3 demonstrates that the additional overhead of mexMPI is minimal in comparison to MPI. The semantics to receive a message does not require the user to allocate space for the data or to specify the size of the data being received. This information is exchanged in the initial transaction and the actual data is exchanged in the next message. However, for small messages, the data is also piggy-backed with the initial transaction. This ensures that the latency for small messages includes only the interpreter overhead, which is the minimal overhead that any function incurs in the Matlab environment. Figure 3.4.1 presents the scalability results of NAS benchmarks IS in comparison with the corresponding C/Fortran implementations. It can be clearly observed that the scalability results of mexMPI are comparable with that of C/MPI. Figure 3.2 presents a comparison of the SLOC (source lines of code) required to code the Matlab/mexMPI and C(or Fortran)/MPI versions of these benchmarks. It can
be clearly seen that in all of the cases, there are significant productivity advantages of using Matlab/mexMPI over C/MPI with only a minor compromise in performance.

![Figure 3.4: NAS IS with mexMPI](image)

**3.5 Conclusions**

Thus, mexMPI is a powerful toolbox that empowers Matlab user’s to write scalable distributed memory code directly in Matlab. mexMPI is highly productive in comparison to the corresponding C or Fortran versions. However, the inherent ‘distributed memory’ view presented by the message passing programming model raises further productivity issues. The ‘distributed memory’ model is difficult to program as the programmer is required to aware of the physical location of the data and requires to manually manage the transfers of data from data space to another. Though this
model removes complex synchronization issues, it also penalizes the programmer even in those sections of code where synchronization is not an issue. In the next section, 4.2, we describe our ‘GAMMA’ toolbox which presents a distributed shared memory toolbox. We also demonstrate its effectiveness and efficiency.
CHAPTER 4

GLOBAL ADDRESS SPACE TOOLBOX

4.1 Motivation

Many parallel computing toolboxes have been proposed in Matlab’s context addressing its performance issues. A detailed discussion of all the toolboxes have been presented in the Chapter 2. Our mexMPI toolbox was also an effort along similar lines. However, parallel computing with ‘message passing’ semantics is considered to be extremely difficult. The user has to explicitly manage the distribution of the data in addition to the distribution of the computation. Further, the user is also responsible for the data transfers between the processes involved in the parallel computation. Efficient transfer of data involves dereferencing the index functions, overlapping computation with communication. Scientific computations are often performed in blocks and these blocks can be physically distributed across different nodes. The user also has to manage the order of transfer of data between different nodes if required logical piece of the block is distributed across different nodes. Further, the entire user code needs to be changed if a different distribution is later detected to be more appropriate. All these operations increase the complexity of parallel programming. This is particularly inappropriate for Matlab environment which is recognized primarily for
its programmability. A completely transparent framework like HPF would require a very sophisticated compiler. Further, in the Matlab environment, which is an interpreted environment, it is extremely difficult to build such a sophisticated compiler. In this context, many global address space programming models have emerged to address the productivity issues in parallel computing while not compromising on the performance that they can deliver. In this chapter, we present our global address space toolbox, GAMMA which enable parallel computation directly in Matlab.

4.2 GAMMA

The GAMMA system has been built as a MATLAB toolbox and is a distributed share memory toolbox. It provides a global view of the matrices and automatically handles the transfer of data from one node to another. The design and implementation of GAMMA is inspired from the ‘Global Arrays’ library. The toolbox has been built using the Global Arrays (GA) [31] libraries and it uses Aggregate Remote Memory Copy Interface (ARMCI) [17] in its communication substrate. The software architecture of GAMMA is shown in Figure 4.1.

Figure 4.1: Software architecture of GAMMA

![Software architecture of GAMMA](image_url)
4.2.1 Programming Model

The features of the GAMMA programming model can be summarized as follows:

- **Global shared view of a physically distributed Array**: GAMMA provides a new distributed array datatype, for parallel programming in MATLAB, called “Global Arrays” that presents each user with a global shared view of the MATLAB arrays that are physically distributed across various processes. Figure 4.2 illustrates this model for an array that is distributed across processes \( P_0, \ldots, P_3 \); the required data that might span across multiple processes can be accessed by referencing it as a single logical block. The efficient implementation ensures that the performance overhead of using global indexing for distributed data is very small. The communication substrate is transparent to the user and hence the user need not be aware of the physical location of the data, in contrast to the message passing based programming model, where each of the process
that owns the section of the logical block posts a send request and the process that requires the data posts a receive request to each of the owner process.

- **Get-Compute-Put computation model:**

  The model inherently supports a Get-Compute-Put computation style, as illustrated in Figure 4.2.1. The data for the computation is fetched from the distributed array independently using a $GA_{Get}$ routine. A logical block, $A[192:704, 128:640]$, where $A$ is a handle to an array of size $1024 \times 1024$, can be obtained by a call to $GA_{Get}$ as $block = GA_{Get}(A, [192, 128], [704, 640])$. ([192, 128] represents the lower indices of the logical block and [704, 640] represents the higher indices of the block as shown in Figure 4.2). The computed data is then stored into the global array, again independently, using a $GA_{Put}$
routine. The computation model enables a GAMMA user to make full utilization of the extensive set of library functions provided by sequential MATLAB.

- **Pass-By-Reference Semantics:**

  The model provides a pass-by-reference semantics for distributed arrays with the belief that it might reduce redundant memory copies of large distributed arrays. By providing pass-by-reference feature, GAMMA compromises on being fully compatible with MATLAB’s value semantics based programming model. However, since GAMMA is primarily targeted for library builders, we believe that pass-by-reference semantics gives better flexibility in the model to exploit performance optimizations wherever possible. Further, MATLAB’s memory manager, due to its copy-on-write optimization can move the data in an array upon a write. Hence, a different memory manager (namely, Memory Allocator) than that of MATLAB is used in the GAMMA system for providing both pass-by-reference semantics and one-sided access.

- **Management of data locality:** The model provides support to control the data distribution and also to access the locality information and therefore gives explicit control to the user to exploit data locality. This encourages library writers to develop locality-aware code.

- **Synchronization:** Support for one-sided and asynchronous access to the global data requires user to handle synchronization mechanisms to ensure consistency. The user is provided with various explicit synchronization primitives to ensure the consistency of the distributed data. Examples of such primitives include
explicit barrier and fence. Though asynchronous mode of access increases programming complexity, it enables overlap of communication with computation without which many applications fail to show good scalability. A user can always use synchronous versions of the data access mechanisms to reduce programming complexity.

- **Data parallelism**: The model provides support for data parallel operations using collective operations that operate on the distributed data e.g., common matrix operations such as `transpose`, `sum`, `scale`, etc. These routines provide efficient implementations of numerous MATLAB operators.

- **Data replication**: GAMMA also provides support to replicate near-neighbor data, i.e., data residing in the boundary of the remote neighbor process.

- **Distributions**: The toolbox supports both regular and irregular block distributions of distributed arrays. Support for irregular block distributions gives better flexibility for the programmer to develop programs that are load balanced.

- **Interoperability with mexMPI**: GAMMA is interoperable with mexMPI. Hence, the programmer can make use of any libraries developed using mexMPI. Further, in the sections of the code where the programmer can exploit the flexibility of mexMPI to obtain better scalability.

- **Processor groups**: The toolbox provides a facility to divide the parallel computing domain into subsets of processors that can act independently of other subsets. This functionality allows improved load balance in the application.
[ rank  nproc ] = GA_Begin();

% define column distribution
dims = [ N  N ]; distr = [ N  N/nproc ];
A = GA_Create(dims, distr);
[ loA  hiA ] = GA_Distribution(A, rank);
GA_Fill(A, 1);

% perform fft on each column of the initial array
tmp = GA_Get(A, loA, hiA);
tmp = fft(tmp);
GA_Put(A, loA, hiA, tmp);
GA_Sync();

% GA_Transpose requires the resultant array to be different than source
ATmp = GA_Create(dims, distr);
GA_Transpose(A, ATmp);
GA_Sync();

% perform fft on each column of the transposed array
[ loATmp  hiATmp ] = GA_Distribution(ATmp, rank);
tmp = GA_Get(ATmp, loATmp, hiATmp);
tmp = fft(tmp);
GA_Put(ATmp, loATmp, hiATmp, tmp);
GA_Sync();
GA_Transpose(ATmp, A);
GA_Sync();
GA_End();

Figure 4.4: Parallel 2D Fast Fourier Transform in GAMMA
4.2.2 Implementation

The implementation of the toolbox presents several challenges. First, MATLAB is an *untyped language*. Hence, the toolbox dynamically tracks the type of the data on which the operations are being performed in order to make the appropriate calls to underlying layers. Further, in MATLAB, a user is not exposed to any explicit memory management routines. Therefore, the memory in the user space (i.e., MATLAB space) is managed automatically by the toolbox. Secondly, in MATLAB, a variable name is not bound to a particular memory location as in languages like C and Fortran. Due to this property, upon providing access to the local portion of the global array, the data that is being written to the array subsequently need not be written in the same location. This might happen due to the copy-on-write optimization present in MATLAB which might move the data to a different location when a memory location is being shared by two or more variables. This makes in-place write impossible if both Global Arrays and MATLAB use different memory managers. Thirdly, MATLAB is based on value based semantics, while we provide reference based semantics to our datatype, “Global Arrays”. This is because GAMMA is intended for building parallel programming libraries and providing value based semantics for the ‘distributed’ arrays might result in copies of large arrays. GAMMA provides Get and Put functions to transfer data between the two different operating spaces and, all the regular MATLAB functions can be used to operate on the data that is brought into the MATLAB’s space. This however, might lead to scalability issues in cases where frequent updates of the distributed arrays are required to be made (the case when different processes interact frequently). To address this issue, we provide a special class of objects, called, “local”. An object of type “local” provides a handle to the local part of
the distributed global array. We provide in-place operations (for e.g., inplacePlus) for MATLAB operators (for e.g., plus), which accepts the two arrays on which the operation is to be performed and the array where the result is to be stored. These operations involve no copy and provide reference based semantics which a programmer can utilize to achieve superior scalability. Programs written in MATLAB are built on the basic set of operators and few in-built functions. Since, all the operators have been overloaded for objects of the type “local”, existing programs written in MATLAB language require no change to execute in the system with the “local” objects.

GAMMA also handles transfer of data between the MATLAB workspace and the GA workspace. In addition, during the data transfer from the GA workspace to the MATLAB workspace, the toolbox dynamically creates a data block in the MATLAB workspace inferring the type, size, and dimensions of the block from the Get request. Furthermore, the toolbox handles the data layout incompatibility issues between MATLAB and GA and preserves the MATLAB semantics for the user. The toolbox also supports out-of-order (arbitrary) array indexing and thereby preserves an important feature of MATLAB. For example, consider a vector \( A[1:100] \). A user can index the vector in an arbitrary fashion as \( A([54\ 87\ 15]) \).

### 4.2.3 Illustration

Figure 4.4 shows the code for parallel 2D Fast Fourier Transform (FFT) using GAMMA. The code is a straightforward implementation of a standard parallel 2D FFT algorithm. The call to \( GA\_Begin \) initializes the underlying layers (MPI, ARMCI, and GA) and returns the rank of the process and the total number of processes. The use of \( distr \) in the call to \( GA\_Create \) defines the data distribution: each block is of
\[ r = x; \]
\[ \rho = r^T \times r; \]
\[ p = r; \]

Do 25 iterations
\[ q = A \ast p; \]
\[ \alpha = \rho / (p^T \ast q); \]
\[ z = z + \alpha \ast p; \]
\[ \rho_0 = \rho; \]
\[ r = r - \alpha \ast q; \]
\[ \rho = r^T \times r; \]
\[ \beta = \rho / \rho_0; \]
\[ p = r + \beta \ast p; \]
End Do
\[ \|r\| = \|x - A z\|; \]

\[ r = x; \]
\[ \rho = r \cdot r; \]
\[ p = r; \]

for \( i = 1:25 \)
\[ q = A \ast p; \]
\[ \alpha = \rho / (p \cdot q); \]
\[ z = z + \alpha \cdot p; \]
\[ \rho_0 = \rho; \]
\[ r = r - \alpha \cdot q; \]
\[ \rho = r \cdot r; \]
\[ \beta = \rho / \rho_0; \]
\[ p = r + \beta \cdot p; \]
end
\[ r = \text{norm}(x - A z); \]

Figure 4.5: Illustration of ‘Distributed Arrays’ toolbox

size \( N \times (N/nproc) \), and process \( P_i \) is assigned the block with logical indices for the upper left corner \((1, 1 + i \times N/nproc)\) and lower right corner \((N, (i + 1) \times N/nproc)\).

For better illustration, the example assumes that the global array \( A \) is initialized with values of 1, using \textit{GA\_Fill}. Each process gets the block of data to operate on locally through a one-sided \textit{GA\_Get} routine (the values of \textit{loA} and \textit{hiA} are 2-element vectors). Since the user has a global shared view of the distributed data, the code has no complex communication involving data location information. Every process then computes their local result using the sequential built-in \textit{fft} function in MATLAB, and puts back the computed data into the distributed array using a one-sided \textit{GA\_Put} call. The example also demonstrates how the programming model allows programmer to utilize the functions provided by sequential MATLAB. Furthermore, the collective operation \textit{GA\_Transpose} does not involve unnecessary data movement between the MATLAB and GA workspaces.
4.3 Distributed Arrays

‘Distributed Arrays’ toolbox is built over GAMMA adding syntactic sugar to improve productivity. It overloads all the operators and few of the commonly used functions. The implementations of these operators and functions are done using GAMMA and are executed as collective operations in our environment. Library writers can develop their applications in GAMMA and provide functionalities for these ‘Distributed Arrays’. Availability of all the necessary libraries makes the programming in this toolbox similar to programming in sequential Matlab. However, this is primarily only a syntactic sugar over GAMMA. It does not handle automatic parallelization issues and leaves it to the GAMMA programmer to develop efficient parallel libraries required for this toolbox.

4.3.1 Illustration

Figure 4.5 presents a comparison of the programming of the CG benchmark using sequential Matlab and the Distributed Arrays toolbox. It can be clearly seen that in a case where all of the operators and functions are effectively overloaded using GAMMA, the ‘Distributed Arrays’ framework provides a transparent framework. However, this approach is severely limited by the requirement of efficient overloading of the numerous functions that are available in Matlab.

4.4 Experimental Results

In this section, we present a detailed assessment of GAMMA along the dimensions of programmability, and performance scalability. Using GAMMA, we implemented the NAS parallel benchmarks: FT (Fourier Transform), CG (Conjugate Gradient),
Table 4.1: Lines of source code for the NAS benchmarks

<table>
<thead>
<tr>
<th>Application</th>
<th>Serial C/Fortran</th>
<th>C/MPI or Fortran/MPI</th>
<th>MATLAB</th>
<th>GAMMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT</td>
<td>665</td>
<td>1205</td>
<td>189</td>
<td>209</td>
</tr>
<tr>
<td>CG</td>
<td>506</td>
<td>1036</td>
<td>59</td>
<td>98</td>
</tr>
<tr>
<td>IS</td>
<td>422</td>
<td>665</td>
<td>128</td>
<td>197</td>
</tr>
<tr>
<td>EP</td>
<td>130</td>
<td>177</td>
<td>35</td>
<td>39</td>
</tr>
</tbody>
</table>

IS (Integer Sort), and EP (Embarrassingly Parallel). These implementations were evaluated against the standard MPI-based Fortran (or C) implementations. All the above benchmarks, namely, NAS EP, FT, CG, and IS incurred no redundant copy from the Global Array space to the MATLAB space. To illustrate the benefit of the in-place array access, we have implemented the Jacobi iterative solver in which accessing the local portions of the global array using reference based semantics can provide significant performance gains by avoiding redundant data copy from the global address space to the MATLAB space. The results are compared with the naive version wherein data is required to be explicitly transferred from Global Arrays address space to MATLAB and vice-versa.

4.4.1 Experimental Setup

The experiments were conducted on the Ohio Supercomputer Center’s Intel Pentium 4 cluster constructed from commodity PC components running the Linux operating system. The hardware and software configuration of the cluster is as follows: two 2.4 GHz Intel P4 Xeon processors on each node; 4GB RAM on each node; InfiniBand interconnection network; Red Hat Linux with kernel 2.6.6; MATLAB Version 7.0.1.24704 (R14) Service Pack 3. All experiments were conducted such that no two
processes were on the same node in the cluster, ensuring that the parallel processing environment is fully distributed.

4.4.2 Programmability

GAMMA retains the programmability features of MATLAB which makes it an attractive system for achieving both high productivity and high performance for computationally intensive applications. Even though there does not exist an ideal metric for evaluating programmability of a parallel language/toolbox, the number of source lines of code (SLOC) is typically being used as a metric to measure programmability (ease of use) [7]. Table 4.1 compares the SLOC required to implement the NAS parallel benchmarks using GAMMA with those required to implement the sequential versions in MATLAB and the parallel and sequential versions of the benchmarks in Fortran (or C). The measurements in Table 4.1 clearly show that the GAMMA-based implementations require only a modest increase in the code size, compared to sequential MATLAB. Furthermore, compared to the standard MPI-based Fortran (or C) implementations of the NAS benchmarks, the number of SLOC is reduced significantly, and sometimes even dramatically (e.g., by a factor of 10 for the CG benchmark). The results clearly indicate that programming in GAMMA could potentially achieve substantial productivity benefits, compared to a message-passing-based programming model. These benefits can be attributed to GAMMA’s feature-rich programming model, array-based semantics and the utilization of MATLAB’s extensive set of sequential libraries.
4.4.3 Performance Analysis

GAMMA not only provides good programmability, but also achieves scalable performance. This section presents the execution time and speedup results of the NAS benchmarks FT, CG, IS and EP, written using GAMMA. We also present these results for the standard MPI-based Fortran (or C) NAS implementation (version 3.2).

The sequential MATLAB versions of the benchmarks are executed with MATLAB 7.0.1, with the just-in-time compilation feature enabled. To demonstrate the benefit of in-place array access, we present the speedup results of Jacobi iterative solver implemented using the naive version of GAMMA (involving copy) and the optimized GAMMA version providing reference-based semantics.

- **NAS FT**: NAS FT solves a 3D partial differential equation using FFTs. This benchmark captures the essence of many spectral codes. From Table 4.2 and Table 4.4 that present the speedup of the GAMMA-based implementation and that of the standard hand-coded Fortran/MPI (or C/MPI) implementation for Class A and Class B that presents the execution times of sequential MATLAB implementation and sequential Fortran/C implementation for Class A and Class B.
B, it can be inferred that the execution times of the GAMMA implementation become comparable to that of the Fortran implementation with increasing number of processors. Our measurements also show that the speedups achieved by GAMMA implementation are slightly better than the speedups achieved by standard Fortran implementation. This is because the communication efficiency of GAMMA is comparable to that achieved using MPI in traditional languages. The computation efficiency of MATLAB also increases superlinearly as the problem size handled by each processor decreases (with the increase in number of processors) in NAS FT.

- **NAS CG**: The NAS Conjugate Gradient (CG) benchmark is a scientific kernel that uses an inverse power method to find the largest eigenvalue of a symmetric definite random sparse matrix. As in the case of NAS FT, from Table 4.2 and Table 4.4, it can be observed that the execution times of the GAMMA implementation gradually become comparable to that of the Fortran implementation. This is possible because of the utilization of the “processor group” feature available in the GAMMA system. It can also be observed that the speedups achieved by GAMMA implementations are either comparable or slightly better than the speedups achieved by Fortran implementations due to the reasons described earlier.

- **NAS IS**: The NAS IS benchmark is a parallel integer sort kernel. It performs a sorting operation that is important in particle method codes; the benchmark tests both integer computation speed and communication performance. As in
Table 4.3: Speedups using reference based semantics to access local portions of the global arrays for Jacobi iterative solver

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive (N=512)</td>
<td>0.882</td>
<td>2.565</td>
<td>4.661</td>
<td>8.940</td>
<td>10.640</td>
</tr>
<tr>
<td>No Copy (N=512)</td>
<td>0.965</td>
<td>2.843</td>
<td>5.023</td>
<td>9.634</td>
<td>14.212</td>
</tr>
<tr>
<td>Naive (N=1024)</td>
<td>0.847</td>
<td>2.880</td>
<td>4.768</td>
<td>9.304</td>
<td>11.395</td>
</tr>
<tr>
<td>No Copy (N=1024)</td>
<td>0.869</td>
<td>3.213</td>
<td>5.708</td>
<td>10.07</td>
<td>14.502</td>
</tr>
</tbody>
</table>

the cases of NAS FT and CG, the execution times of the GAMMA implementation eventually get closer to that of the C implementation. Superior speedups can be observed even in this case.

- **NAS EP**: The NAS EP benchmark is an embarrassingly parallel kernel. It provides an estimate of the upper achievable limits for floating point performance - that is, performance without significant interprocess communication. As with the other benchmarks, it can also be observed here that the performance gap between the two implementations decrease with increasing number of processors.

- **Jacobi iterative solver**: The Jacobi iterative solver uses the Jacobi method to solve a linear system of equations arising in the solution of a discretized partial differential equation. All data were distributed using a two-dimensional block distribution. Table 4.3 demonstrates the benefit of providing reference-based semantics for accessing the local portions of the global array. The sequential execution times are 16.218 sec and 72.209 sec for the problem sizes with N=512 and N=1024 respectively. This benchmark also utilizes the in-place semantics provided by GAMMA to achieve superior speedups.
Overcoming MATLAB’s Memory Limitations

The use of MATLAB for large-scale computationally intensive applications is also limited because of memory constraints. For example, the sequential version of MATLAB runs out of memory and is unable to execute the NAS benchmarks FT and EP for the Class B problem size. One of the direct advantages of parallel computing is the ability to run problems of larger scale due to the availability of memory from multiple nodes. By enabling parallel computing, GAMMA allows MATLAB users to run large-scale problems with data distributed across multiple processors. This is illustrated with the results for FT and EP for Class B problem size. With GAMMA, NAS EP Class B can be run on four or more processors and NAS FT Class B can be run on eight or more processors. The speedup is calculated with respect to the execution time using the minimum number of processors that could successfully run the benchmark in GAMMA. Table 4.4 shows the speedup for GAMMA and Fortran implementations of NAS EP and FT Class B, with the execution time on four processors as the reference for EP and that on eight processors as the reference for FT.

These experimental results clearly indicate that (1) the GAMMA implementations of the benchmarks achieve good scalability, (2) the performance gap between

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT (GAMMA)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>2.29</td>
</tr>
<tr>
<td>FT (C/F+MPI)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1.98</td>
</tr>
<tr>
<td>CG (GAMMA)</td>
<td>0.99</td>
<td>2.01</td>
<td>8.97</td>
<td>19.87</td>
<td>39.85</td>
<td>68.50</td>
</tr>
<tr>
<td>CG (C/F+MPI)</td>
<td>0.99</td>
<td>4.35</td>
<td>8.35</td>
<td>22.53</td>
<td>37.45</td>
<td>68.06</td>
</tr>
<tr>
<td>IS (GAMMA)</td>
<td>0.99</td>
<td>1.94</td>
<td>3.52</td>
<td>6.28</td>
<td>10.96</td>
<td>17.38</td>
</tr>
<tr>
<td>IS (C/F+MPI)</td>
<td>0.94</td>
<td>1.71</td>
<td>3.32</td>
<td>5.95</td>
<td>9.72</td>
<td>16.35</td>
</tr>
<tr>
<td>EP (GAMMA)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1.90</td>
</tr>
<tr>
<td>EP (C/F+MPI)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1.99</td>
</tr>
</tbody>
</table>

Table 4.4: Speedup: NAS, Class B

4.4.4 Overcoming MATLAB’s Memory Limitations

The use of MATLAB for large-scale computationally intensive applications is also limited because of memory constraints. For example, the sequential version of MATLAB runs out of memory and is unable to execute the NAS benchmarks FT and EP for the Class B problem size. One of the direct advantages of parallel computing is the ability to run problems of larger scale due to the availability of memory from multiple nodes. By enabling parallel computing, GAMMA allows MATLAB users to run large-scale problems with data distributed across multiple processors. This is illustrated with the results for FT and EP for Class B problem size. With GAMMA, NAS EP Class B can be run on four or more processors and NAS FT Class B can be run on eight or more processors. The speedup is calculated with respect to the execution time using the minimum number of processors that could successfully run the benchmark in GAMMA. Table 4.4 shows the speedup for GAMMA and Fortran implementations of NAS EP and FT Class B, with the execution time on four processors as the reference for EP and that on eight processors as the reference for FT.

These experimental results clearly indicate that (1) the GAMMA implementations of the benchmarks achieve good scalability, (2) the performance gap between
the Fortran (or C) implementation of the NAS benchmarks and the corresponding GAMMA implementation decreases as more processors are used, (3) the GAMMA system provides significant productivity benefits for library writers to build their custom parallel libraries using distributed arrays, and (4) MATLAB programmers can run computations of larger problem sizes that are impossible to be run using the standard sequential MATLAB.
5.1 Introduction

Traces of programs find a wide range of applications such as understanding program behavior, finding opportunities for performance optimizations, and debugging. However, data access traces can be extremely large, which makes their storage and analysis expensive and difficult. Many trace compression techniques have been proposed in earlier work [21, 18, 13, 38] to address this challenge. Some of these earlier approaches have achieved significant compression ratios. However, there are numerous compression opportunities that these techniques fail to exploit. In particular, repetitive behavior exhibited by the underlying software manifests itself in trace patterns that often have great potential for compression. General handling of such patterns is a challenging problem. Simple cases such as repeated accesses to the same memory locations are relatively easy to handle (e.g., “access addresses a, b, a, b, a, b, . . .”). The more interesting case is when the same simple memory access pattern (e.g., ab from above) is not repeated, but rather a higher-level patterns exists. As a very simple example, consider the initialization of a data array with some constant values. The memory access pattern observed at run time is of the
Algorithm 1 Infer

Globals: T[d][2], X[d][2]
Constants: d – max loop nest depth
Starting call: infer([0], 0, 0)
Output: Loop structure inferred in values of T

infer(I’, l, tp) -> bool::

I = I’
if l==d-1 then
    {no transition points for outermost loop – just consume till end of trace}
    while tp != END_OF_TRACE do
        (tp, succ) = progress(I, l, tp, I[l]+1, +)
        if !succ then
            return false
        end if
    end while
end if

(tp, succ) = find_transition(I, l, 0, tp)
if !succ then
    return false
end if

{update array ref coeff w.r.t l+1}
(tp,succ) = find_transition(I, l, 1, tp)
if !succ then
    return false
end if

{update upper bound coeff w.r.t to l+1}
while !infer(I, l+1,tp) do
    (tp, succ) = regress(I,l)
    if !succ then
        return false;
    end if
end while
end while
return true
Algorithm 2 Progress

\[\text{progress}(I, l, tp, I_\perp, \text{dir}) \rightarrow (\text{int}, \text{bool})::\]

\[\text{suc}\text{c} = \text{true}\]
\[I' = I\]
\[\text{if } \text{dir} \equiv \text{+} \text{then}\]
\[lb = I[l]; \text{ub} = I_\perp; s\text{t}=1\]
\[\text{else}\]
\[lb = I_\perp; \text{ub}=I[l]; s\text{t}=-1\]
\[\text{end if}\]
\[\text{for } i \text{ in } lb \text{ to } \text{ub step } st \text{ do}\]
\[\text{if } l > 0 \text{ then}\]
\[\text{(tp,suc}\text{c}) = \text{progress}(I', T, I_\perp-1, \text{tp}, \text{ub}(I,l-1), \text{dir})\]
\[\text{end if}\]
\[\text{if } (!\text{suc}\text{c}) \text{ or } (\text{tp}==\text{END}\_\text{OF}\_\text{TARCE}) \text{ or } (\text{trace}[\text{tp}] \neq \text{predicted_val}(I', T)) \text{ then}\]
\[\text{return } (\text{tp}, \text{false})\]
\[\text{end if}\]
\[\text{tp} += 1\]
\[I'[l] += 1\]
\[\text{end for}\]
\[\text{return } (\text{tp}, \text{true})\]

form \text{base}, \text{base} + \text{off}, \text{base} + 2 \times \text{off}, \text{base} + 3 \times \text{off}, \ldots \] where \text{base} is the address of the first array element and \text{off} is the offset. Such a pattern has a very compact representation and its detection could lead to excellent compression ratios.

We propose a trace compression approach that attempts to infer the code structure that might have generated the trace. Specifically, we aim to discover subtracts that could have been generated by some loop nest. The nests of interest have the following properties: (1) loop bounds are affine functions (i.e., \(a_1i_1 + \ldots + a_ni_n + b\); \(a_i\) and \(b\) are constants) of loop iteration variables for the surrounding loops; (2) the memory addresses referenced by any statement in the loop nest can be expressed as affine functions of the iteration variables of the surrounding loops. We are interested in
the general case of a multi-dimensional loop nest, with imperfectly nested and non-rectangular loops. This approach not only helps to achieve significant compression ratios, but also produces trace representations that are easy to understand by a human (e.g., a programmer working on performance tuning or debugging).

The proposed trace compression technique involves considering various candidate loop nest structures that can generate the given trace sequence. The algorithm takes as input the ordered sequence of trace elements and incrementally builds various possible candidate structures that could have generated the trace sequence up to this point. We define a notion of loop nest compactness based on the number of statements and loops in the candidate structures. The algorithm tries to select the most compact candidate structure. In other words, the proposed approach explores the complete space of possible candidate structures (up to a pre-defined maximum loop depth) and determines the most compact one that faithfully captures the given trace.

The exploration of the space of possible loop nest structures is performed greedily. While processing the sequence of memory reference, at any moment of time the algorithm attempts to construct the most compact structure that explains the trace prefix seen up to this point. In general, it is impossible to guarantee that at this moment the correct decision is made — the structure that is chosen may later be determined to be incorrect. If some subsequent memory reference does not “fit” the currently-chosen structure, this structure is discarded and the algorithm backtracks in the trace, aiming to discover the next compact candidate structure. Such backtracking is unavoidable in we want to ensure that eventually the most compact correct structure is constructed for the entire trace. Our experiments show that the extent of backtracking (measured in terms of the total trace elements processed) for various
benchmarks is within a factor of 1.6 times the total length of the trace. Thus, the best compression does require backtracking, but this comes without an excessive cost. An evaluation using the SPEC CPU 2000 benchmarks demonstrates the effectiveness of this approach.

The paper is organized as follows. Section 5.3 outlines the basic ideas of the proposed technique using a simple example. Section 5.4 describes the trace analysis algorithm. Chapter 2 discusses previous approaches on trace compression and compares their effectiveness with our algorithm. Section 5.5 presents an experimental evaluation on traces generated by the SPEC benchmarks.

5.2 Related Work

Program traces including data memory accesses are widely used for various applications. Due to the increasing size of these traces, various trace compression techniques have been widely explored. Since the trace sizes are usually too huge, most of the recent compression techniques have been based on an incremental approach.

Historically, various general purpose compression techniques and tools (like bzip2, gzip etc.) have been used for trace compression. However, such general purpose compression techniques fail to achieve acceptable compression ratios. Numerous specialized techniques for trace compression have been evolved. These techniques try to extract employ a suitable combination of general purpose techniques to achieve higher compression ratios. The initial approaches in this direction have tried to leverage spatial and temporal locality in the program execution. Since most of the programs that are written for efficient execution have good temporal and spatial locality, this approach achieves good compression ratio. MACHE [37] and SEQUITUR [40] are
examples of such an approach. Tools like MACHE encode the stride instead of the full memory address. If encodes the traces for various instructions (reads, writes and fetches) separately into different streams and uses strides instead of the memory addresses. SEQUITUR builds a context free grammar from a sequence of incoming memory addresses. The PDATS [19] algorithm employs an additional strategies like run-length encoding over the above technique. This would result in a higher level of compression over techniques like MACHE.

The techniques of subsequent generation [28, 29] involved extracting higher level structures from the traces. Since, the general purpose techniques are complementary to these techniques, various general purpose compression tools are usually used for a further level of compression of the resultant structures. Systems like WPP (Whole Program Paths) [25] uses SEQUITUR sequence analysis algorithm to extract higher level data structures to represent the trace. WPP is an incremental algorithm and builds a context-free grammar from the program trace. The grammar is build based on the rules that no two symbols can appear twice and every symbol can be used more than once. The context free grammar is built using these set of rules results in an output that is hierarchical in structure. WPP is very effective for the cases where regular repetitive pattern exists in the trace. WPP also provides a bound on the size of the output structure to be the logarithm of the trace size. WPP also improves upon its compression ratio by removing few unpredictable values as noise. Noise is encoded in a separate structure to ensure the main structure would be as compact as possible. Programs with good spatial and temporal locality would usually achieve very good compression ratio with WPP. It can effectively used for compressing control
flow information in the traces. Kobayashi et al also try to construct a control flow model of the sequence of trace events.

The recent works of VPC [18] and NLR (Nested Loop Recognition) [21] extract a set of predictors that can predict the subsequent entries in the trace. These set of predictors are suitably adapted to adapt to the changing pattern of the input trace events. Once these techniques encode the trace into a hierarchical structure, it is passed through general purpose compression tools to perform a further level of compression. VPC uses value predictors and is based on the observation of all previous values. It is an online method and when a value is read, the predictor that predicts the input correctly would be the output. The approaches removes the unpredictable values as noise and is directed to a separate stream. The noise direction to a separate stream ensures that the predictors extend to a larger range of the trace and ensures a higher level of compression. NLR derives affine predictors from the trace sequences and combines these predictors into a nested loop structures to derive a code structure that when executed regenerates the entire trace. NLR reads an input traces and builds a sequence of loop nests. The user inputs several parameters like the maximal breadth of the loop, the models of functions to represent the upper bounds for inner loops and functions that compute values. Since each loop nest can potentially encode a large number of trace events, NLR achieves the best known compression ratios currently. The NLR output is very similar to the output that our algorithm generates. NLR, similar to our approach uses recognizing loop nests as an effective trace compression strategy. NLR first builds various affine predictors and combines them to achieve higher level structures. The significant difference of NLR’s approach from ours become evident for the cases that involve backtracking. Further,
our approach performs an exhaustive search of the candidate structures to arrive at the most compact structure.

5.3 Overview

The goal of the algorithm is to generate a nested loop structure which, if executed, would reproduce precisely the given trace sequence. Thus, this loop structure is a lossless compression of the given trace. Both loop bounds and memory reference expressions are affine functions of surrounding loop iterators. This section provides a high-level overview of the approach and illustrates it with some simple examples. The detailed description of the algorithm is presented later in the paper.

Input. The algorithm takes as input a sequence of trace events (memory references) as input. An additional input is an upper bound on the depth of loop nesting in the generated structures. Regardless of the value of this bound, the algorithm will always output a structure that correctly captures the trace; the bound affects only the trade-off between compression ratio and cost of the analysis.

Loop structures. The loop structure that the algorithm builds is represented by an abstract syntax tree (AST), based on the following simple grammar:

\[
\text{Root} ::= ((\text{Loop}) \mid \text{statement})^+ \quad \text{Loop} ::= ((\text{Loop}) \mid \text{statement})^+
\]

The root node has an ordered sequence of children representing loops or individual statements. Each loop node also has an ordered sequence of children representing the loop body. In addition to these children, the loop body has a number of attributes: (1) a loop iteration variable, whose name is unique in the entire AST, and (2) a sequence of coefficients \(a_1, \ldots, a_n, a_{n+1}\) where \(n\) is the number of surrounding loops. If the iteration variables for these surrounding loops are \(i_1, i_2, \ldots, i_n\), the iteration variable
for this loop has values \(0 \leq i \leq a_1i_1 + a_2i_2 + \ldots + a_ni_n + a_{n+1}\) for any combination of values for \(i_1, i_2, \ldots, i_n\). Thus, the coefficients \(a_1, \ldots, a_n, a_{n+1}\) define an affine function that captures the upper bound of the loop in terms of the values of the iterators for outer loops. Each loop is normalized, in the sense that it starts with the loop iterator being 0. Furthermore, the algorithm ensures that the upper bound is always non-negative, for all values of the iterators for outer loops.

The leaf nodes corresponding to terminal statement represent statements that make memory references. For simplicity, each node is associated with exactly one memory reference expression. The form of this expression is \(\text{out}(b_1i_1 + b_2i_2 + \ldots + b_mi_m + b_{m+1})\), where \(m\) is the number of loops surrounding the statement. Here \text{out} is a placeholder for a program expression the “emits” a single memory reference. Coefficients \(b_1, \ldots, b_{m+1}\) express this reference as a function of the iterator variables \(i_1, \ldots, i_m\) of the surrounding loops.

**Output.** The algorithm generates an AST as discussed above, with all coefficients \(a_i\) and \(b_i\) having some particular values. If this AST is considered as a “generator” (i.e., executable code), the sequence of memory references produced by \text{out} statements is exactly the same as the sequence of references in the input trace.

**Compactness and size.** We will say that an AST \(A_1\) is compact in comparison to an AST \(A_2\), if (1) \(A_1\) has fewer statement nodes or (2) \(A_1\) and \(A_2\) have the same number of statement nodes and \(A_1\) has fewer loop nodes. An AST with a maximum of \(n\) statements with a maximum loop depth of \(d\) has an upper bound of \(n(d + 1)\) nodes. Also, the maximum number of coefficients associated with every node is at most \(d + 1\). Hence, the maximum size of such a structure is in the order of \(n(d + 1)^2\). Since \(d\) is usually a small constant, the overall size of the structure is quite small.
5.3.1 Detecting Affine Patterns

Consider the following code:  
\[
\text{for } i = 1:6, \text{ out}(5*i); \text{ end }
\]
The code generates the trace 5, 10, 15, 20, 25, 30. Note that this is an affine pattern with a single variable varying — the difference between the successive trace events is constant. When the first event (i.e., 5) is processed by the algorithm, it creates a single statement node. Since there are no surrounding loops, this statement node has a single coefficient with value 5. After the second trace event 10 is processed, the algorithm modifies the structure to be  
\[
\text{for } i = 0:1, \text{ out}(5*i+5); \text{ end }
\]
The coefficients for the statement node are derived from the two instances of the iterator: since the statement is of the form \(\text{out}(a_1 i_1 + a_0)\) and we know that the output is 5 when \(i_1 = 0\) and 10 when \(i_1 = 1\), we can conclude that \(a_1 = 5\) and \(a_0 = 5\). Upon processing the rest of the events, the algorithm increments the upper bound of the loop iterator until the end of the pattern. The final result is  
\[
\text{for } i_1 = 0:5, \text{ out}(5*i_1+5); \text{ end }
\]
Note that the generated code is a normalized equivalent of the original code — the loop iteration starts from 0.

5.3.2 Detecting Loop Nests

Now, consider a code structure with a single statement and a rectangular iteration space:  
\[
\text{for } i = 1:4, \text{ for } j = 1:3, \text{ out}(i+j); \text{ end end }
\]
The trace generated by this code is 2, 3, 4, 3, 4, 5, 4, 5, 6, 5, 6, 7. The algorithm, following the procedure outlined in the previous example, generates the following structure for the first 3 trace events:  
\[
\text{for } i_1 = 0:2, \text{ out}(i_1+2); \text{ end }
\]
With the fourth trace event, the approach observes a variation in the affine pattern and expands the current structure as follows:  
\[
\text{for } i_2=0:1, \text{ for } i_1 = 0:ub(i_2), \text{ out}(i_1+i_2+2); \text{ end end }
\]
Here \(ub(i_2)\) denotes
the affine function for the upper bound of the inner loop. At this moment, this function is not fully known — the only information is that the current iteration vector is \( \langle i_2, i_1 \rangle = \langle 1, 0 \rangle \) and \( \text{ub}(i_2 = 0) \) is 2. Later, the analysis observes that the variation in the affine pattern occurs with the seventh trace event. Hence, the resulting code structure (up to six trace events) is

\[
\text{for } i_2 = 0:1, \text{ for } i_1 = 0: \text{ub}(i_2), \text{ out}(i_1 + i_2 + 2); \text{ end end}
\]

and furthermore \( \text{ub}(i_2 = 1) \) is 2. Having the value of \( \text{ub}(i_2) \) for two distinct values of \( i_2 \) is enough to determine that it is a constant function (i.e., the coefficient for \( i_2 \) is zero). The approach continues this processing until the last trace event; the final code structure is

\[
\text{for } i_2 = 0:3, \text{ for } i_1 = 0:2, \text{ out}(i_1 + i_2 + 2); \text{ end end}
\]

5.3.3 Detecting Triangular Loop Nests

Consider the case where the original code structure is similar to the previous example but with a triangular iteration space:

\[
\text{for } i = 1:4, \text{ for } j = 1:i, \text{ out}(i+j); \text{ end end}
\]

The generated trace is 2, 3, 4, 4, 5, 6, 5, 6, 7, 8. The analysis creates one statement \( \text{out}(2) \) for the first trace event. After the first three events are processed, the following structure is built:

\[
\text{for } i_1 = 0:2, \text{ out}(i_1+2); \text{ end}
\]

Continuing as described in the previous example, with the first six trace events, the constructed structure is

\[
\text{for } i_2 = 0:1, \text{ for } i_1 = 0:2, \text{ out}(i_1+2*i_2+2); \text{ end end}
\]

However, with the seventh trace event, the current structure fails: the only possibility is to continue with iteration vector \( \langle i_2, i_1 \rangle = \langle 2, 0 \rangle \), but there is a mismatch with the seventh event. At this point, the algorithm backtracks and checks whether there are any other compact structures that can match the trace. So, at the second trace event, it considers the possibility of variation in the outer loop instead of the inner loop:

\[
\text{for } i_2 = 0:1, \text{ for }
i_1=0:ub(i_2), out(i_1+i_2+2); end end. Note that the iteration vector after the first two events is (1,0) and ub(i_2 = 0) is 0, but the full form of ub(i_2) is not fully known yet. The AST (i.e., the generator code) is extended until the affine pattern breaks at the fourth event. The generator structure until this point is for i_2=0:1, for i_1=0:i_2, out(i_1+i_2+2); end end. The analysis continues to check if the subsequent events match this structure. Finally, after the tenth event, the algorithm finalizes the generator to be for i_2=0:3, for i_1=0:i_2, out(i_1+i_2+2); end end.

5.3.4 Detecting Multiple Statements

Consider the trace generated by the following multi-statement structure: for i_1=1:4, out(i_1), out(2*i_1+1); end. The trace generated by this code is 1, 3, 2, 5, 3, 7, 4, 9. Suppose that the maximum loop depth is three. The first two events result in the structure for i_1=0:1, out(2*i_1+1); end. However, no structure with a single loop and a single statement can produce the entire sequence. The algorithm discards this current structure and builds a new structure with a single statement and two loops. It checks if a structure with a rectangular iteration space can explain the trace. It also checks for a structure with a triangular iteration space. However, no such structures can produce the entire trace. In general, having two loops and one statement is not sufficient for this trace. Next, the analysis starts considering structures with three loops and one statement. Again, for this case, no such structures can generate the entire trace. Given that the maximum loop depth is three, the algorithm has exhausted all possibilities with only a single statement, and it tries to find a structure with two statements. So, for the first two trace events, the generator would be out(1); out(3); — there would be no loops. After the next two
trace events, the generator is modified to be \texttt{for } i_1=0:1, \texttt{out}(i_1+1), \texttt{out}(2*i_1+3), \texttt{end}. After all events are processed, the final generator is \texttt{for } i_1=0:3, \texttt{out}(i_1+1), \texttt{out}(2*i_1+3), \texttt{end}.

5.3.5 Normalized Form

The AST that is generated is always represented in a normalized form: loop iterators always start from zero, they are incremented by one, and the upper bounds of loops are always non-negative. Hypothetically, suppose that the given trace was generated by some code that is an imperfectly-nested multi-dimensional loop in which (1) loop bounds and memory reference expressions are affine functions of surrounding loop iterators, (2) loop iterators are incremented by one, and (3) each loop iterates at least once. It is easy to see that this code can be represented in an equivalent normalized form. If a top-level loop in this code were of the form \texttt{for } i = a:b \ldots \texttt{end}, where \(a\) and \(b\) are some constant values, the loop can be rewritten as \texttt{for } i' = 0:b-a \ldots \texttt{end} and all occurrences of \(i\) in the loop body can be replaced by \(i'+a\). Such a substitution will result in new affine functions for loop bounds and memory references in the body. At the next level of loop nesting, \texttt{for } j = a_1*i'+a_2:b_1*i'+b_2 \ldots \texttt{end} can be transformed to \texttt{for } j' = 0:(b_1-a_1)*i'+b_2-a_2 \ldots \texttt{end} and all occurrences of \(j\) in the loop body can be replaced with \(j'+a_1*i'+a_2\). This again will result in the new affine functions inside this second-level loop. The substitutions can continue from outer to inner loops, resulting in an equivalent structure in normalized form.

5.4 Algorithm for Trace Analysis

The overall goal of the algorithm is to detect nested loop structures that can regenerate the trace. The basic idea is to detect affine patterns in the sequence of
trace events and fit these different affine patterns in a loop structure. The algorithm builds a set of candidate structures; these structures can regenerate the trace prefix that has been processed up to this point. Upon scanning further trace events, the algorithm checks for possible modifications to “fit” these trace events. Upon a “fitting failure” of the current candidate structure, the algorithm discards it and backtracks to the next compact structure that might be able to generate the trace. Ultimately, this approach performs an exhaustive search of a set of candidate structures that can generate the trace, always preferring more compact structures to less compact ones. This process continues until the entire input trace is completely scanned.

5.4.1 Phases of the Algorithm

The algorithm consists of seven phases: initialization, compliance check, progression, expansion, backtracking, coefficient derivation, and termination.

Initialization

The algorithm always starts with a single candidate structure for the AST. This structure contains only a single leaf node representing the a statement node. The single coefficient corresponding to the affine function of this node is initialized to the value of the first trace event. After initialization, every subsequent event from the trace is subjected to a compliance test.

Compliance Check

At every additional scan of a trace event, the analyzer first checks whether the current candidate structure can predict the next trace event correctly. There are several possibilities for the next trace event given a candidate structure:
• If the current statement node has another statement node as its right sibling, the next trace event is generated by this right sibling. The generation uses the same values of the iterators of all surrounding loops (i.e., the same iteration vector) as used for the previous event. If the the predicted event matches the scanned event, the algorithm continues this process until no further right sibling is found. However, if in the process, the predicted value fails to match the next scanned event, the structure is said to be non-compliant and the algorithm discards this candidate structure and backtracks to the next compact structure. The backtracking procedure is described in Section 5.4.1.

• If no right sibling exists, and the algorithm checks if the current loop iterator’s upper bound is fully determined. If the bound is fully determined and if the value of the current loop iterator is less than its upper bound, then the next trace event is predicted by incrementing the loop iterator and producing the output of the first (leftmost) statement node. In this case, if the predicted value fails to match the input value, the code is said again said to be non-compliant and the algorithm discards this candidate structure and backtracks to the next compact structure. However, if the current loop iterators equal their respective upper bounds, the algorithm considers possible expansion as described in 5.4.1.

• If the current loop iterator’s upper bound is undetermined, the algorithm proceeds in progression phase. In other words, if the next iteration vector instance is not uniquely predictable, the algorithm proceeds by considering various possibilities of the next iteration vector instance.
Thus, the algorithm proceeds either in one of the progression, expansion or backtracking mode based on the outcome of the compliance test.

**Progression**

The algorithm proceeds in the progression phase if the value of the next iteration vector is not unique as described earlier. It increments the current iteration vector instance. This can be done in multiple ways.

- Incrementing the innermost loop iterator
- Incrementing any of the outer loop iterator and resetting all the inner loop iterators to zero. Note that it also marks an upper bound instance for all the inner loop iterators.

If the predicted event obtained by incrementing the innermost loop iterator matches that of the next event scanned from the trace, the algorithm proceeds continues the progression. However, if it fails to ‘fit’ the scanned event, other possible progressions are considered starting from the case of incrementing the innermost loop iterator to incrementing the outermost loop iterator. The algorithm continues the progression as long as the scanned events ‘fit’ the predicted values. However, if none of the predicted values matches the scanned event, the current candidate structure is discarded and follows the backtracking mode.

**Possible Expansion**

In the expansion phase, the algorithm generates different candidate structures expanding from the current candidate structure. The expansion of the candidate structure occurs when the current affine pattern breaks and none of the iterators
whose upper bounds have been determined are incomplete. In other words, the current candidate structure is a valid compact structure for this subset of the trace sequence. The expansion phase adds new nodes to the AST of the current candidate structure to ‘fit’ subsequent trace events. The expansion can result in multiple candidate structures. Specifically, the algorithm generates the following candidate structures:

- The upper bound instance of the current loop iterator is marked, reset to zero and an outer loop iterator is added to the candidate structure. There are multiple ways in which a new outer loop iterator is added. The new iterator node can be added either to immediately encompassing the loop nest of the current statement node or at any point in the path from this point to the root. This new loop iterator node however must ensure that the length of the iteration vector of any statement node is at most ‘d’. The algorithm continues from the beginning of the second iteration instance of this new loop iterator node.

- The current candidate structure can also be expanded by adding a new statement as the right most child to the root node of the AST.

In the case where at least one new loop node has been added to the candidate structure, the affine functions (both upper bound functions and the array reference functions) are updated to contain an additional variable. It is important to observe that due to the affine nature of the functions, addition of a new variable to the affine function does not alter any of the coefficients that are previously determined for any other variable. This is because the coefficient ‘c’ of a loop iterator variable ‘p’ is dependent only on the variation in ‘p’. However, the upper bound functions that have been
flagged as determined would be flagged as undetermined. The algorithm progresses by choosing the most compact candidate structure among all the candidates.

**Coefficient Derivation**

The earlier modes of initialization, progression and expansion have described the possible shapes (AST nodes) of the candidate structures. The AST is not completely defined as long as the coefficients corresponding to each node (both loop nodes and statement nodes) are undetermined. The coefficients of a particular loop variable is determined in an affine function (both upper bound functions and array reference functions) when any two instances of the function are available for corresponding instances of the loop iterator. Specifically, when the second iteration of a loop variable is executed completely, all the coefficients corresponding to that variable in all the functions would have been determined. In an affine function, ‘f’, the derivation of affine coefficients ‘c’ of a variable ‘p’ follows the principle that 

\[ c = \frac{f(p_1) - f(p_2)}{p_1 - p_2}. \]

Hence, ‘c’ is derived when \( f(p_1) \) and \( f(p_2) \) are determined for any \( (p_1, p_2) \) such that \( p_1 \neq p_2 \).

**Backtracking**

The algorithm enters backtracking phase if and only if the current candidate structure fails to ‘fit’ the next trace event correctly. The failure to ‘fit’ the next trace event occurs if and only if the predicted value of the upper bound function of the current loop iterator was wrong. In other words, the current iterator is unable to complete its set of iterations. The algorithm in this phase discards the current candidate structure and backtracks to the trace event where the latest decision on an upper bound function was made. The algorithm resets the upper bound function
to a trace event prior to this event and follows the progression phase. However, in
the case where there can be no alternate value be set to the upper bound function,
the algorithm discards the current candidate structure and proceeds to the expansion
phase. Note that this backtracking procedure would result in an exhaustive search of
the candidate structures.

**Termination**

The termination condition for the algorithm is satisfied if and only if when all the
trace events are scanned and the current candidate structure is not in an incomplete
loop iteration. In other words, the last trace event scanned must correspond to a trace
event generated by the rightmost leaf node of the AST. Further, the corresponding
iteration vector must equal the predicted upper bound values of all the iterators.
However, if the iteration is incomplete and the algorithm has scanned all the trace
events in the sequence, the candidate structure is said to be non-compliant and it
backtracks in the way described earlier.

### 5.4.2 Algorithm

The routines infer 1, progress 2, transition 3 and regress 4 combined present
our algorithm where the possible structures can have only one statement node. Our
algorithm where the for this perfectly nested case consists of four main routines,
namely, infer, progress, regress and transition. An overview of the routines used in
the algorithm is presented in this section. An extension of these routines to extend
to the case where the candidate structures can be arbitrary imperfectly nested loop
structures is presented later in this section.
\textbullet \ infer(I,l,tp) \rightarrow \text{bool} : \text{ Infer finds transition points for loops (l..d-1). Transition points are the events corresponding to the upper bound instances of any of the loop iterator in the candidate structure. It returns true if these can be found to match the remainder of the trace starting at index ‘tp’ and is false otherwise. The basic idea is that we determine the the transition points by observing the affine pattern. These transition points are our first guess of the instances of upper bound functions corresponding to a loop iterator. These transition points are altered during ‘regress’.}

\textbullet \ progress(I,l,tp,I,dir) \rightarrow (\text{int, bool}) : \text{ This procedure progresses through the trace events starting with iteration index I of the loop iterator ‘l’, until I[l] reaches I[\ell]. The direction of progress (forward or backward) is given by ‘dir’. ‘tp’ is the starting trace event index. On success, progress (with no intervening mispredictions), returns pair of next-trace-index-to-be-processed and true. On failure, it returns (-1,false). The basic idea of this routine is that using the guess of transition points provided by the ‘infer’ routine, we check if these transition points are correct. The check fails if in the journey of the loop iterator from I to I[\ell], the current candidate structure fails to ‘fit’ the trace sequence starting from ‘tp’. Failure in ‘progress’ would result in backtracking and is handled by ‘regress’. Note that ‘progress’ is again executed when the guess of transition points is altered.}

\textbullet \ regress(I',l) \rightarrow (\text{int, bool}) : \text{ The regress procedure changes the transition points chosen for loop l, (since they have failed) given the iteration index I’. If the procedure completes a successful regression, the transition points are
updated and the return is a pair next-trace-index-to-be-processed and true else, it returns (-1, false). When the progress routine fails (i.e. the guess of transition points is wrong), the algorithm backtracks and the ‘regress’ routine alters the guess of the transition points to chose from within a set of possible pairs of transition points.

- $\text{find\_transition}(I, l, s, tp) \rightarrow (\text{int}, \text{bool})$: The ‘find\_transition’ procedure finds the s-th transition point (0 or 1) for loop l, starting at trace index ‘tp’. The current iteration index is given by I. If the procedure finds a transition, it returns the pair (next trace index to be processed, true) else, it returns (-1, false). This routine finds the trace events that breaks the affine pattern. In other words, it finds the latest possible values for the transition points.

5.4.3 Imperfectly Nested Loops

The algorithm when the candidate structures can include any arbitrary imperfectly nested loop structure is an extension of the routines described earlier. An imperfectly nested structure is represented as a binary tree and is generated by the following rules:

\[
\langle P \rangle ::= \langle S \rangle
\]

\[
\langle S \rangle ::= \langle SS \rangle
\]

\[
\langle S \rangle ::= \langle LS \rangle
\]

\[
\langle S \rangle ::= \langle \text{assignment} \rangle
\]

Any candidate structure is represented in the form of a tree where each node corresponds to the LHS of the above rules and its children correspond to the RHS
of the rules. The choice of rule is denoted as "S S" or "L S", corresponding to concatenation of two statement nodes and loop nesting of a statement respectively.

The above routines described for the perfectly nested case with a single statement is extended to allow arbitrary imperfectly nested loop structures. The ‘infer’ routine for the earlier case finds the latest transition points as the trace events corresponding to which the affine pattern breaks. An increment in the loop iterator surrounding the current statement node would correspond to the next event in the trace sequence. However, for the arbitrary imperfectly nested loop structure, an increment in the loop encompassing the current statement node would correspond to the event in the trace sequence that is encountered after a complete loop iteration of the sibling nodes in the tree. The ‘progress’ routine is also extended similarly to consider the next predicted event as the event corresponding to the sibling of the current node. The sibling can either be a statement node with the same set of loop iterators or a statement node with different loop iterators. The next event in the former case can be predicted using the current value of the iteration vector and in the latter case, the next event corresponds to the first event of the next statement node. The ‘regress’ routine as in the perfectly nested single statement case would consider the transition point by decrementing the event that corresponds latest transition point. When the ‘progress’ routine fails to ‘fit’ the next trace event, the algorithm expands the current candidate structure either by adding a loop around the statement node (or a collection of the statement nodes) or expand by concatenating a new statement node. Note that addition of a loop introduces a new variable in all the inner nodes of the tree and concatenation of a statement node would add an additional child to the root of the tree. The traversal of the ‘progress’ routine in this case is a bottom-up traversal from the first innermost
statement node to the latest statement node in the tree. Each loop node in the tree is associated with two transition points (instances of upper bound function of the loop iterators). The modified ‘infer’ routine is presented in the algorithm 5.

### 5.4.4 Compactness

We present a proof for the compactness of the generated structure.

In this section, we prove that there cannot be a generator with a lesser number of statement nodes than the statement nodes in the generator constructed with our algorithm, given a maximum depth, d.

In order to prove the above, we prove that:

- At any point in the algorithm, there does not exist any other structure in the space of candidate structures which has lesser number of statement nodes than the candidate structure under consideration.

- The space of candidate structures consists of all the possible candidate structures with a fewer statement nodes.

Note that if the these two claims are proved, we can claim that the size of the structure generated by the algorithm out of a trace generated by a program is bound by 

\[(\#\text{statement nodes in the original program} \times d)\]

**Proof of Claim 1:**

At any point in the algorithm, there does not exist any other structure in the space of candidate structures which has lesser number of statement nodes than the candidate structure under consideration.
The proof of the claim is by construction. The algorithm starts with the simplest structure i.e one statement. If the claim is false, there exists some candidate structure has less than ‘k’ statements. This is not possible since the algorithm performs a search of the most compact structure in the space of candidate structures whenever a new statement is added to the structure or whenever the current candidate structure is discarded. The algorithm in such case starts evaluating with this structure that has less than ‘k’ statements. Whenever, there is no expansion of the candidate structure, the algorithm pursues with the same candidate structure without any addition of new candidates to the space of candidate structures. Hence, by construction, it is always true that at any point in the algorithm, there cannot exist any other candidate structure with less than ‘k’ statements.

Proof of Claim 2:

The space of candidate structures consists of all the possible candidate structures with a fewer statement nodes.

The proof of the claim is again by induction on construction process. The grammar described in the earlier section ensures that all possible ‘shapes’ are considered in the algorithm. A ‘shape’ is the AST structure with the upper bound functions undefined. Notice that using the grammar, any node can have any arbitrary number of loop nodes and statement nodes in a recursive fashion (until the maximum depth is d). In other words, the grammar the algorithm uses constructs all possible ‘shapes’. Now, it is sufficient to prove that all possible values for the upper bound function are considered.

Trip: We define a trip for a loop iterator as the transition from the lower bound (0) of the loop iterator to the upper bound of the loop iterator. Note that if the loop
iterator is not the inner most, a trip of any outer loop iterator means trips for all its inner loops as defined by their respective upper bounds.

An iteration of a loop iterator is said to be complete when all the inner loop iterators of that iterator have completed their respective trips corresponding to their respective upper bounds. Note that when a loop iterator completes two iterations, all the affine coefficients corresponding to that loop iterator in all the functions (upper bound functions and array reference functions) can be evaluated.

**Transition:** We define transition as when the inner most loop iterator has completed its trip.

Now, consider any arbitrary ‘shape’ that is under consideration. The algorithm progresses only if the predicted trace events match the events in the trace. The progress routine fails only if the latest decision about the transition point was wrong. The regress routine resets the transition point to the corresponding decremented iterator before restarting the progress routine. The decremented iterator corresponds to changing the decision on the transition point for the outermost loop. If the outermost loop cannot be decremented further, the next inner loop is decremented. This process continues for the entire iteration space. In the case where no decrement is possible, the candidate structure is expanded by adding a new statement. Thus, the algorithm explores all the possible transition points for each of the iterator. Thus, the algorithm considers all possible values of the upper bounds for a given shape.

From the above two claims, we can conclude that the algorithm explores all possible candidate structures with less than ‘k’ nodes before the candidate structure with ‘k+1’ nodes are considered.
5.5 Experimental Evaluation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>TraceAnalyzer</th>
<th>NLR [21]</th>
<th>VPC [18]</th>
<th>TraceAnalyzer</th>
<th>NLR</th>
<th>VPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>applu</td>
<td>8540</td>
<td>5610</td>
<td>2441460</td>
<td>13238.32</td>
<td>20153.28</td>
<td>46</td>
</tr>
<tr>
<td>art</td>
<td>672</td>
<td>838</td>
<td>1254</td>
<td>5115.50</td>
<td>4102.17</td>
<td>2741</td>
</tr>
<tr>
<td>ammp</td>
<td>154548</td>
<td>852435</td>
<td>1817221</td>
<td>302.75</td>
<td>54.89</td>
<td>25</td>
</tr>
<tr>
<td>apsi</td>
<td>2232</td>
<td>7392</td>
<td>32397</td>
<td>8130.07</td>
<td>2454.86</td>
<td>560</td>
</tr>
<tr>
<td>equake</td>
<td>1832</td>
<td>1988</td>
<td>36088</td>
<td>165.70</td>
<td>152.70</td>
<td>8.12</td>
</tr>
<tr>
<td>mgrid</td>
<td>3852</td>
<td>5417</td>
<td>12405</td>
<td>23346.46</td>
<td>16601.55</td>
<td>7249</td>
</tr>
<tr>
<td>mesa</td>
<td>505824</td>
<td>4782425</td>
<td>4038886</td>
<td>58.33</td>
<td>6.17</td>
<td>7</td>
</tr>
<tr>
<td>lucas</td>
<td>2268</td>
<td>5300</td>
<td>1755</td>
<td>18532.47</td>
<td>7930.50</td>
<td>23949</td>
</tr>
<tr>
<td>fma3d</td>
<td>1548</td>
<td>4167</td>
<td>7003</td>
<td>950.68</td>
<td>353.17</td>
<td>210</td>
</tr>
<tr>
<td>swim</td>
<td>856</td>
<td>1367</td>
<td>1402</td>
<td>99304.93</td>
<td>62183.63</td>
<td>60631</td>
</tr>
</tbody>
</table>

Table 5.1: Trace Compression comparison of TraceAnalyzer with that of NLR and VPC (in bytes)

An experimental evaluation of the algorithm was done using the memory traces of SPEC 2000 benchmarks. Each trace event is a memory reference and the entire trace is a sequence of these memory references. We have compared the compression that we achieve with NLR and VPC. Table 5.5 gives compressed trace sizes (in bytes) for compression on memory reference traces. The table includes columns for comparing performance of the TraceAnalyzer (our approach) with that of NLR and VPC. We note that our approach outperforms the other approaches in all but one of the cases. Further, the improvement with our scheme is very significant for several of the benchmarks where NLR and VPC were unable to achieve good compressions (e.g. ammp and mesa).

The superior compression ratio compared to NLR is likely because of the exhaustive search of the candidate structures. The NLR approach primarily uses a greedy
approach in finding nested loop structures. In many cases, it could result in constructing a structure that is not as compact. Specifically, our approach achieves significant compression where there was a necessity to backtrack to find a more suitable candidate structure. The degree of backtracking (measured in terms of percentage of additional trace events scanned over the total number of events in the trace) in most of the cases is also small. In most of the cases, extensive backtracking occurs only in the initial stages where the algorithm searches for the suitable candidate structures. Once a suitable candidate structure is found, the huge trace sizes that are processed amortize the overhead of backtracking.
Algorithm 3 Find Transition

\begin{algorithm}
\textbf{find\_transition} (I, l, s, tp) \rightarrow (\text{int, bool}):\\
\textbf{if} I[l] < 2 \textbf{then}\\
\hspace{1em} (tp, succ) = \text{progress}(I, T, l, tp, I[l]+1, +)\\
\hspace{1em} \textbf{if} !\text{succ} \textbf{then}\\
\hspace{2em} \text{return} (-1, false)\\
\hspace{1em} \textbf{end if}\\
\hspace{1em} I[l] += 1\\
\hspace{1em} \textbf{end if}\\
\textbf{if} I[l] < 2 \textbf{then}\\
\hspace{1em} (tp, succ) = \text{progress}(I, T, l, tp, I[l]+1, +)\\
\hspace{1em} \textbf{if} !\text{succ} \textbf{then}\\
\hspace{2em} \text{return} (-1, false)\\
\hspace{1em} \textbf{end if}\\
\hspace{1em} I[l] += 1\\
\hspace{1em} \textbf{end if}\\
\{\text{have two values – update array ref coe}ffs\}\\
\text{succ} = \text{true}\\
\textbf{while} succ \textbf{do}\\
\hspace{1em} (tp', succ) = \text{progress}(I, T, l, tp, I[l]+1, +)\\
\hspace{1em} \textbf{if} suc\text{c} \textbf{then}\\
\hspace{2em} I[l] += 1\\
\hspace{2em} tp = tp'\\
\hspace{1em} \textbf{end if}\\
\hspace{1em} \textbf{end while}\\
\{\text{made maximum progress. mark latest possible transition point}\}\\
T[l][s] = I[l]\\
X[l][s] = tp\\
I[l] = 0\\
I[l+1] += 1\\
\textbf{return} (tp, true)\\
\end{algorithm}
Algorithm 4 Backtrack

1: revert(I,l) -> (int, bool):
2: I=I'
3: assert l != d;
4: {never regress outermost loop}
5: I[l+1] = 1
6: if T[l][1] > T[l][0]+1 then
7:   I[l] = T[l][1]-1
8:   {go back by 1}
9: (tp,succ) = progress(I, l, X[l][1]-1, T[l][1]-2, -)
10: assert succ; {you can always progress in - dir}
11: T[l][1] = T[l][1] - 1
12: X[l][1] = tp
13: {update upper bound coeffs}
14: else
15:   succ = false
16:   while (!succ) and (T[l][0] > 0) do
17:     I[l] = T[l][0]-1
18:     {go back by 1}
19:     (tp,succ) = progress(I, l, X[l][0]-1, T[l][0]-2, -)
20:     assert succ; {you can always progress in - dir}
21:     T[l][0] = T[l][0] - 1
22:     X[l][0] = tp
23:     {update array ref coeffs and discard upper bound coeffs}
24:     (tp, succ) = find_transition(I,l,1,1)
25: end while
26: end if
27: if T[l][0]==0 then
28:   return (-1,false)
29: end if
30: return (tp, succ)
Algorithm 5 Infer for Imperfectly Nest Loop Structure

1: infer(S,tp)::
2: if S.body == terminal then
3:   return (tp+1,succ)
4: end if
5: if S.body == "L S" then
6:   (tp,succ) = infer(S.body.child[1],tp)
7:   assert succ
8:   /*first one should have some soln.*/
9:   (tp,succ) = infer(S.body.child[0],tp)
10: while !succ do
11:     (tp,succ) = regress(S.body.child[0], tp)
12:     (tp,succ) = regress(S.body.child[1], tp)
13:     if !succ then
14:        return (-1,false)
15:     end if
16:     (tp,succ) = infer(S.body.child[1],tp)
17:     (tp,succ) = infer(S.body.child[0],tp)
18: end while
19: else
20: if S.body == "S S" then
21:   (tp,succ) = infer(S.body.child[0], tp)
22:   assert succ
23:   /*first one should have some soln.*/
24:   (tp, succ) = infer(S.body.child[1], tp)
25: while !succ do
26:     (tp,succ) = regress(S.body.child[1], tp)
27:     (tp,succ) = regress(S.body.child[0], tp)
28:     if !succ then
29:        return (tp,false)
30:     end if
31:     (tp,succ) = infer(S.body.child[0],tp)
32:     (tp, succ) = infer(S.body.child[1], tp)
33: end while
34: return (tp, succ)
35: end if
36: end if
CHAPTER 6

LARGE ARRAYS

6.1 Motivation

Many scientific applications have extreme memory requirements which cannot be satisfied by the memory on a single computer. The ever-increasing gap between the processor and I/O speeds of computer systems has become a problem for many of these I/O intensive applications like scientific simulations. The common solutions to address this problem have been either providing a transparent framework like virtual memory or by frameworks supporting out of core programming. While virtual memory is an ideal solution from the productivity point of view, achieving high performance is difficult. Frameworks supporting out of core programming has been a good and effective way to address this limitation. However, efficient out-of-core programming is considered to be extremely difficult and is inappropriate for a high productivity frameworks like Matlab. This is because efficient out-of-core programming requires efficient overlapping of I/O and computation; efficient prefetching and caching policies and to hide the significant latency. The framework also needs to determine other policies like data layout and redistribution policies that are ideal for these large arrays.
6.1.1 Overview

Our ‘Large Arrays’ (LA) library provides a high-level programming model in Matlab without the programmer having to manage system parameters. Large Arrays is a runtime system and is supported by the analyzer and profiler. The profiler collects the instances of the I/O calls during the prototype runs. The analyzer builds a model based on the profiling data collected during the prototype runs for various optimizations. The runtime system decides its policies based on the model constructed by the analyzer. The programmer first prototypes his/her memory-intensive application using the simple programming model. Developing LA code involves the user to read the relevant blocks of data, perform the computation and write back the data to the disk if necessary. The user can prototype his LA code using smaller values for the problem sizes so that the problem run is an in-core version. The user then submits the LA code to the analyzer. The analyzer using a series of steps determines various important parameters about the application. The analyzer also generates inspector code to be used by the runtime system. The analyzer first profiles the in-core version using a set of values for the ‘structural parameters’, called the ‘base run’. Structural parameters are the program parameters on which the array index functions and the loop iteration space functions are dependent upon. The analyzer also collects the profiles for the runs by varying the structural parameters one at a time from the values used for the ‘base run’. We later prove that such a collection of profiles is sufficient (under certain assumptions) to accurately determine the ‘LA structure’ 6.4.1 of the application. The analyzer using the structure derived and from the in-core profiles determines various other properties (for ex., data layout, caching and prefetching policies) that are necessary to efficiently run the large problem case. Finally, the runtime
library executes the application for the large problem case using the decisions made by the analyzer. The analyzer may also generate inspector code to determine various parameters based on the input values of the structural parameters. The runtime library executes the inspector code in the background thread to determine the policies to be used for the execution of the application.

6.2 Programming Model

‘Large Arrays’ presents a Get-Compute-Put programming model. This type of programming model is already demonstrated to be highly productive by the GAMMA toolbox. The ‘Large Arrays’ library primarily consists of the following functions:

- **Create**: Creates a disk-based array of a specified size
- **Get**: Reads any logical rectangular block of the array from the disk
- **Put**: Writes data to any logical rectangular block of the array on the disk
- **Delete**: Deletes the array from the disk

The interface contains the array creation/deletion routines and routines to retrieve and store the data into these large arrays. The retrieval/storage is performed in ‘logical blocks’, where each block is rectangular and is uniquely identified by its lower and higher indices in each of the dimension. The programmer is expected to program in a manner that he/she requests the appropriate block for computation; performs the computation using Matlab’s libraries and finally store the data back into an array (if required). The key function of the ‘Large Arrays’ library is to efficiently store and retrieve the data into the Matlab environment.
6.3 Execution Model

The LA framework is implemented as a Matlab library. Each large array array is implemented as a file on the disk. The ‘Create’ routine creates an array and initiates a corresponding file on the disk. The data in the file is stored in a linearized fashion and the writing/retrieval of the data is always done in units of chunks. A call to ‘Get’ can request any arbitrary section of the array. The ‘Get’ routine divides the request into chunks and reads the chunks from the file into memory. The ‘Get’ routine returns the relevant data from the chunks. The ‘Put’ routine also divides the request into chunks. These chunks are read from the file into memory. The relevant portions of the chunks are altered with the data that is to be written. The routine further writes the data back to the disk. This simple library enables Matlab users to program applications with a large problem sizes. Note that it is the programmer’s responsibility to ensure that the requested block of the array fits in memory.

6.3.1 Transparent framework

The above described ‘Get-compute-Put’ model enables users to write large scale applications directly in Matlab. However, it requires the programmer to rewrite the existing codes. The LA framework is thus extended by overloading the operators of a class of objects in Matlab, ‘LA’. The operators on the arrays and the functions are overloaded in the library. This enables the users to directly use the numerous libraries that are available in regular Matlab. It ensures that the user need not have to rewrite the applications. However, this approach is limited by the availability of the overloaded library and needs to be extended as and when new libraries are added to regular Matlab.
6.3.2 Efficient Execution

The LA framework as described earlier only enables the data intensive computing directly in Matlab. However, for the large problem sizes, an efficient implementation of the library is essential. The high disk latency and slow disk bandwidth makes it necessary to minimize the accesses to the disk and to use various latency hiding techniques to enhance the performance of the application. Prefetching and Caching are the two commonly used optimizations for this purpose. A good prefetching and caching policy requires a good guess of the future access patterns. The LA framework uses a separate thread for prefetching to ensure that the data movement is done overlapping the computation portion of the application.

The LA system consists of three primary components:

- Analyzer
- Library

The Analyzer component collects the application level data at a coarser granularity from the prototype runs. It then builds a generator for predicting data accesses for a different set of problem parameters using the profiling data collected by the profiler to make intelligent decisions on caching and prefetching policies. The library component implements the routines to read and write from the disk. These implementations are supported by the caching and prefetching systems.

A typical use-case of Matlab is that scientists use Matlab for prototyping and then recode in C/Fortran for the actual production runs. The framework uses traces from prototype runs to predict the trace events for the actual run. The framework uses an ‘Analyzer’, which is an extension of the ‘Trace Generator’ described in the earlier
The `Trace Generator' is extended by extending the affine functions involved in the generator to contain the application parameters as additional variables. The application parameters are the parameters that differ between a prototype run and the actual run.

### 6.4 Analyzer

The Analyzer takes as input the application written using the LA framework, the set of values for the prototype runs and generates a model of the application that can generate the trace of the application corresponding to the actual set of values for the parameters. The trace of a run consists of the sequence of references to the `Get' and `Put' functions and the indexes describing the data block of the Large Arrays involved in the application. The Analyzer collects the trace data from various profiling runs corresponding to different sets of parameter values. The Analyzer uses an extended version of the trace generator described in Chapter 5 to build the model of the application. The model thus built is used for generating the trace sequence for the values of parameters corresponding to the actual run.

#### 6.4.1 Parametric Trace Generator

The construction of a simple trace generator is described in Chapter 5 in great detail. The trace generator can be extended to be used for generating a good guess of the trace for any given set of values for the parameters. Figure 6.4.1 shows a generic form of any phase of the program. The algorithm described in the Chapter 5 is extended as described in this section.

The traces of the application are first collected for various sets of values for the application parameters. The trace generator algorithm is executed for each of the
trace. Each of the run generates a separate trace generator for each trace. The affine functions involved in the generator are extended to contain the application parameters as additional set of variables. The affine parameters for the parameters can be evaluated as \( a_p = (f(p_1) - f(p_2))/(p_1 - p_2) \), where \( a_p \) is the affine coefficient of the affine function \( f \); \( f(p_1) \) is the value of the constant in the function \( f \) for the value of the parameter \( p_1 \); \( p_1 \) and \( p_2 \) are the values of the parameter \( p \) for the two different runs. Note that we need the traces corresponding to each parameter varying at least once. However, this is possible if each of the trace generator obtained through runs on different traces is of the same ‘shape’. However, it is possible that the shapes obtained through runs on different traces are entirely different and hence, the corresponding affine functions may not be compared. In order to ensure that the shapes obtained by the trace generator on different traces is same, the trace generator uses a uniform shape for all the traces under consideration. The trace generator is implemented as a multi-threaded application. Each thread runs on a different trace. The trace generator runs only the main thread during the intiation and expansion phases. The affine coefficients are derived during every expansion phase using the method described earlier. Each thread executes independently on respective traces during the progression phase and compliance check. If the compliance of any thread fails, the main thread aborts all the remaining threads and enters into the backtracking phase. This process continues until a common trace generator is derived for all the traces or until all the candidate structures have been explored. The coefficients of the parameters for the outer most loop are derived during the termination phase.

It is to be noted here that greater the number of traces that are available to the trace generator greater is the accuracy of the trace generator.
Illustration

Consider the two traces $2, 3, 4, 3, 4, 5, 4, 5, 6, 5, 7, 6, 4, 5, 7, 8, 6, 7, 8, 9$ for $N = 2$ and $N = 3$ respectively. The trace generator initiates with the single statement node `out(2)`. During the first expansion phase, the structure would be for $i_1 = 0:1$, `out(i1+2)`; end. The progress phase for each of the trace is executed separately. During the next expansion phase, the candidate structure would be for $i_2=0:1$, for $i_1 = 0:N$, `out(i1+i2+2)`; end end. Note that the coefficient of the parameter $N$ is derived to be 1 from the two traces during this expansion phase. The procedure continues and during the termination phase, the candidate structure is derived to be for $i_2 = 0:N+1$, for $i_1 = 0:N$, `out(i1+i2+2)`; end end.

6.4.2 Library

Library is the component that actually implements the I/O routines that are exposed to the user. This component also handles the prefetching and caching systems. Specifically, the library includes the routines for

- **Init:** This routine initializes the LA framework. It takes as input the values for the parameters of the application and initializes the prefetching thread. The prefetching thread then starts generating the trace events corresponding to the set of parameters given as the input.

- **Create:** This routine creates a Large Array. It creates a file corresponding to the array on the disk.

- Reading from any arbitrary block of data of the Large Array
% Let <s> represents the set of all structural parameters

\[
\text{for } i = l_p h_i(1) : l_p h_i(2) - l_p h_i(1) : l_p h_i(p_i),
\]
% ‘p_i’ represents the number of iteration points
% ‘p_i’ is a function of structural parameters only

% Let ‘Ahndl’ represents is a n-dimensional array and
lo_i and hi_i represent the lower and higher index functions

\[
\text{Get}(\text{Ahndl}, [lo_1(i, < s >), ..., lo_n(i, < s >)], [hi_1(i, < s >), ..., hi_n(i, < s >)]);
\]
...
\[
\text{for } j = j_p h_i(1) : j_p h_i(2) - j_p h_i(1) : j_p h_i(p_j)
\]
% ‘p_j’ is a function of i and the structural parameters

\[
\text{Get}(\text{Bhndl}, [lo_1(i,j, < s >), ..., lo_n(i,j, < s >)], [hi_1(i,j, < s >), ..., hi_n(i,j,< s >)]);
\]
...
end
...
end

Figure 6.1: Structure of a ‘Phase’
• Writing to any arbitrary block of data of the Large Array

• Deleting any large array

The Get and Put routines receive input regarding the policies related to cache replacement policy, prefetching policy and the actual model that is built by the analyzer component. The Get and Put routines break down the requested block into multiple chunks. The routines first check if the requested chunks are present in the cache buffer. If the requested chunks are absent, they issue file I/O requests after suspending all the outstanding I/O requests that are currently underway.

6.5 Prefetching

Prefetching is one of the most important optimization in hiding the I/O latency. Effective prefetching policy involves accurately predicting the next few future access patterns. Predicting next few access patterns requires building an accurate model of the application from the profiling data. The runtime system implements prefetching as a separate threading executing in the background. In addition to accurate prediction of the future access pattern, it is also important to minimize the prefetching execution time. The prefetching policy of the separate thread is based on the Trace Generator that has been generated based on the prototype runs. The trace generator generates the next trace event. The prefetching thread breaks the next block to be fetched into chunks of a fixed size. It further searches for these chunks in the memory for the availability of these chunks. If the chunk is not found in memory, the prefetching thread issues a read from the disk.
6.6 Caching

Caching is another important optimization that is included in the LA framework in hiding the I/O latency. Good caching policy reduces the number of misses. The penalty of a miss in out of core accesses is far too significant and hence it is extremely important to reduce the number of cache misses. Belady et. al [6] prove that the best cache replacement policy is to replace the block that has the maximum reuse distance. However, the runtime system implementing this cache replacement policy requires the knowledge of all the future accesses. The replacement policy used by the system is to replace all the chunks that have a reuse distance of greater than the buffer size. The reuse distance can be checked if it is greater than the buffer size by calculating using the next trace events predicted by the Trace Generator. In case where the replaced chunks are insufficient to accommodate the newly requested blocks by the main execution thread, the chunks with the greatest reuse distance are replaced.

6.7 Experimental Evaluation

In this section, we present a detailed assessment of LA along the dimensions of programmability and performance. Using LA, we have implemented the NAS parallel benchmarks: FT(Fourier Transform), CG(Conjugate Gradient), IS(Integer Sort), and EP(Embarrassingly Parallel). The experimental evaluation primarily evaluates the feasibility of large scale problems using LA and the effectiveness of the prefetching and caching optimization policies in our framework.
6.7.1 Experimental Setup

The experiments were conducted on the Ohio Supercomputer Center’s Intel Pentium 4 cluster constructed from commodity PC components running the Linux operating system. The hardware and software configuration of each of the cluster is as follows: two 2.4 GHz Intel P4 Xeon processors on each node; 4GB RAM on each node; Infiniband interconnection network; Red Hat Linux with kernel 2.6.6; Matlab Versio 7.0.1.24704 (R14) Service Pack 3. All experiments were conducted such that no two processes were on the same node in the cluster, ensuring that the parallel processing environment is fully distributed.

6.7.2 Programmability

LA was designed to retain the programmability features of Matlab which makes it an attractive system for achieving both high productivity and high performance for memory intensive applications. As described earlier in the chapter 4, we use source lines of code (SLOC) as a metric to measure the programmability. Table 6.1 compares the SLOC required to implement the NAS parallel benchmarks using LA with those required to implement the sequential versions in Matlab and the sequential versions in C/Fortran. The SLOC measurements in Table 6.1 clearly show the LA based implementations require only a modest increase in the code size compared to sequential Matlab. Further, compared to the standard C/Fortran based implementations of the NAS benchmarks, the number of SLOC is reduced significantly. The results clearly indicate that programming in LA could potentially enable programming large scale versions of the applications with only a small increase in the SLOC count.
### Table 6.1: Lines of source code for the NAS benchmarks

<table>
<thead>
<tr>
<th>Application</th>
<th>Serial C/Fortran</th>
<th>Matlab</th>
<th>LA</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT</td>
<td>665</td>
<td>189</td>
<td>215</td>
</tr>
<tr>
<td>CG</td>
<td>506</td>
<td>59</td>
<td>106</td>
</tr>
<tr>
<td>IS</td>
<td>422</td>
<td>128</td>
<td>221</td>
</tr>
<tr>
<td>EP</td>
<td>130</td>
<td>35</td>
<td>47</td>
</tr>
</tbody>
</table>

### Table 6.2: Execution time of NAS Benchmarks with LA

<table>
<thead>
<tr>
<th>Application</th>
<th>C/Fortran(Class B)</th>
<th>LA(Class B)</th>
<th>C/Fortran(Class C)</th>
<th>LA(Class C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT</td>
<td>195</td>
<td>605</td>
<td>OOM</td>
<td>2420</td>
</tr>
<tr>
<td>CG</td>
<td>650</td>
<td>1275</td>
<td>OOM</td>
<td>45226</td>
</tr>
<tr>
<td>IS</td>
<td>10.19</td>
<td>199</td>
<td>OOM</td>
<td>802</td>
</tr>
<tr>
<td>EP</td>
<td>210</td>
<td>852</td>
<td>OOM</td>
<td>3648</td>
</tr>
</tbody>
</table>

#### 6.7.3 Performance Analysis

The LA framework enables programming large scale versions of the applications while not only retaining the programmability features but also achieves scalable performance. This section presents the execution time of the NAS benchmarks FT, CG, IS and EP, written using LA framework. We also present results for the standard Fortran (or C) based implementations (version 3.2). The sequential Matlab versions for these benchmarks are not feasible at the problem sizes considered.

Table 6.2 presents the execution times of LA implementation and Fortran (or C) based implementation wherever possible for Class B and Class C. The results clearly show that LA enables programming directly in Matlab for larger problem sizes. The sequential Matlab runs out of memory while executing FT and EP benchmarks of
Class B. It also runs out of memory (OOM) for each of CG, FT, IS and EP benchmarks for Class C. The C/Fortran versions of the benchmarks are able to execute Class B versions. However, even the C/Fortran versions run out of memory while executing these benchmarks for the Class C problem sizes. The results also show the performance improvement achieved due to prefetching and caching optimizations included in the framework. The prefetching thread primarily predicts the next I/O request based on the trace generator and fetches the predicted block of data.

6.8 Related Work

Virtual memory is a good productive solution for handling data sets large than the available physical memory. It provides a transparent framework and performs disk swaps when a reference is made to the data present on the disk. However, virtual memory is good only when disk swaps are infrequent and is very bad for many applications eg. signal processing. The complexity of I/O systems on modern computers and the performance requirements of scientific applications have so far prevented this approach being effective. Application specific virtual management [44] may address some of these issues, but it has not yet been proven in large-scale systems. Out of core methods provide advantages over virtual memory since they gives explicit control to manage and reduce the movement of data between the memory and disk. However, for obtaining good performance, the out-of-core program should also consider various methods to hide/reduce the I/O latency. Overlapping computation and I/O, prefetching and caching are among the many common methods used for this purpose. However, all of these optimizations come at a significant programming complexity which requires efficient orchestrating of the resources.
Mowry et al. [30] use compilation based approach to fully-automatically insert I/O prefetching statements. The compiler provides crucial information about the future access patterns without burdening the programmer. The operating system however, needs to support non-binding prefetch and release hints for managing I/O. The system is supported by a runtime layer which receives hints from the operating system adapting a dynamic behavior to minimize the prefetch overhead.

Active buffering is a technique that exploits idle resources on parallel computers to hide the cost of writes from parallel application programs. Active buffering reduces or even eliminates the application-visible cost of I/O, by taking advantage of available idle memory on processors. Experiments with many applications [1] showed that active buffering is very effective and provides significant potential performance upside with little overhead. The suggested active buffering mechanism relies on the fact that the cost of writing to remote memory on high performance networks is faster compared to writing to the disk (ROMIO etc.).

The idea behind I/O-aware compilation is to expand a parallel programming language to include simple high-level I/O commands (ex. output array A to data repository R) and then make use of a compiler for that language that knows how to map those calls to library-specific I/O calls (e.g., calls to HDF or netCDF to write out the specified array), and also knows how to optimize such calls. For example, the compiler can turn such calls into asynchronous I/O calls that can be started as soon as possible and terminated as late as possible, to maximize the overlap between I/O activities and other program activities. However, experiments showed that I/O-aware compilation does not offer performance benefits beyond those obtained by the use of active buffering, in typical scientific codes. The reason is that I/O-aware
compilation moves all I/O-related activities to a separate thread, which runs in the background. However, many I/O activities in a high-performance code require communication between processors, typically because an array’s in-memory distribution across processors is different from the desired distribution on disk. Moving these communication activities to the background hurts typical applications as they have to wait for their turn to use the communication facilities. The resulting switching between threads slows down overall application run time. It has been observed that I/O-aware compilation did tend to increase run times slightly. In contrast, active buffering performs communication-intensive activities in the foreground, avoiding the slowdown. However, with hyperthreaded architectures may make it possible for communication-intensive I/O activities to run in the background, without causing significant slowdowns for applications.

Log based I/O projects [2] try to improve the I/O performance of these simulation applications by exploiting their special I/O characteristics. Furthermore, different processes of the simulation either write to different files or disjoint locations of the same file, so concurrency control (order of writing) is also not an issue.
CHAPTER 7

CONCLUSIONS

This proposal has demonstrated the feasibility of a highly productive runtime system that enables users to develop scalable parallel applications directly in Matlab environment. Experimental results on several examples support this conclusion. These results represent a significant success towards adopting scripting languages for parallel computing in order to benefit from the immense productivity advantages provided by them.

GAMMA provides a Get-Compute-Put programming model which is demonstrated to be highly productive. Productivity advantages of GAMMA are demonstrated using NAS benchmarks. In many cases, the productivity advantages were over a factor of 10. The results have also demonstrated that it is possible to develop parallel applications with good scalability. Further, it enables the reuse of numerous existing sequential Matlab libraries as part of a parallel application.

The proposal also discusses an out-of-core toolbox, LA that would enable Matlab users to run large scale problems from within the Matlab environment. Again, it provides a Get-Compute-Put programming model which has already been demonstrated to be highly productive. The runtime system uses caching and prefetching techniques to improve the performance of the applications developed using the toolbox. The
runtime system has two significant components, profiler and analyzer. Profiler collects the profiles of block-level accesses to the disk for the prototype runs. Analyzer uses the profiles collected to make caching, prefetching and data layout decisions. Preliminary results have shown the promise of the approach and solving the issues raised in the proposal would enable LA to be a highly efficient runtime system.

The mexMPI and GAMMA toolboxes are currently being used by the staff and users at the Ohio Supercomputer Center and are available as a public release. GAMMA and Large Arrays are a part of a larger effort to develop a high-productivity environment called ParaM [33]. The success of these toolboxes would demonstrate that it is possible to develop a highly productive runtime system that would enable scientific community to develop large scale applications without significant compromise on performance.
APPENDIX

GAMMA API

Initialization

- GA_Begin

Syntax:

GA_Begin()

GA_Begin(heap)

GA_Begin(heap,stack)

Description:

Initializes the GA environment. This includes initializing the MPI layer, the memory allocator (MA), and the Global Arrays space. The user may specify the size of heap and stack space required for the entire program. The default value for heap space is 512MB and stack space is 64MB. If the application requires larger amount of memory to be managed dynamically in the global space, heap and stack values can be mentioned accordingly.

Note:
This is a collective operation.

- **GA_End**

  Syntax:
  
  GA_End()
  
  GA_End([ArrayHandle1,...])

  **Description:**
  Terminates the GA environment and destroys the Global Arrays that were created and frees all allocated memory.

  **Note:**
  This is a collective operation.

- **GA_Create**

  Syntax:
  
  [handle] = GA_Create(dims)
  
  [handle] = GA_Create(dims, chunk)

  dims - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the size or extent of the ith dimension of the global array
chunk - vector of 'n' elements where 'n' is the number of dimensions of the
global array to be created and the ith element in the vector denotes the mini-
mum size that the ith dimension of the global array must be divided into among
the processes

Description:
Creates a Global Array with distribution as specified by dims and chunk. Spec-
ifying $chunk[i] \leq 1$ will cause that dimension to be distributed evenly. If chunk
is not specified, the entire array is distributed evenly, i.e. the elements are
distributed such that each process gets equal number of elements along each
dimension.

Note:
This is a collective operation.
The handle is a numeric handle. It has to be used only with GAMMA functions.
Using it with non-GAMMA functions might give undesired results.

• GA_Create_irreg

Syntax:
[handle] = GA_Create_irreg(dims, block, map)
dims - vector of 'n' elements where 'n' is the number of dimensions of the global
array to be created and the ith element in the vector denotes the size or extent
of the ith dimension of the global array
block - vector of 'n' elements where 'n' is the number of dimensions of the global
array to be created and the ith element in the vector denotes the number of blocks that the ith dimension of the global array must be divided into among the processes

map - vector indicating the starting index of each block

**Description:**
Creates an array as per the user-specified distribution information. The distribution is specified as a Cartesian product of distributions for each dimension.

**Note:**
This is a collective operation.

functions might give undesired

- GA_Destroy

**Syntax:**
GA_Destroy(ArrayHandle)

**Description:**
Destroys the Global Array *ArrayHandle* and frees the allocated memory.

**Note:**
This is a collective operation.
• **GA_Duplicate**

Syntax:

\[
\text{[new\_handle]} = \text{GA\_Duplicate}(\text{ArrayHandle})
\]

Description:

Creates a new array by applying all the properties of another existing array. It returns a new array handle.

Note:

This is a collective operation. Note that only the array properties are reflected and not the values of the elements.

• **GA_Copy**

Syntax:

\[
\text{GA\_Copy(From\_ArrayHandle, To\_ArrayHandle)}
\]

Description:

Copies elements from source array (From\_ArrayHandle) to destination array (To\_ArrayHandle).

Note:

This is a collective operation. Note that the arrays must be of the same shape
and identically aligned.

One-Sided

- **GA_Get**

  **Syntax:**
  
  \[
  \text{[local\_buffer]} = \text{GA\_Get(ArrayHandle,lo, hi)}
  \]

  **Description:**
  
  Copies data asynchronously into the local array buffer of the calling process from a section of global array.

  **Note:**
  
  Task parallelism can be exploited using **GA_Get** and **GA_Put** functions. Perform **GA_Sync** before **GA_Get** to avoid any kind of inconsistencies. Inconsistencies might result due to load imbalance if **GA_Sync** is not used before **GA_Get**.

- **GA_Put**

  **Syntax:**
  
  \[
  \text{GA\_Put(ArrayHandle, lo, hi, buf)}
  \]

  **Description:**
Copies data asynchronously from the local array buffer of the calling process to a section of global array.

**Note:**
Task parallelism can be exploited using GA Get and GA Put functions. Perform GA Sync after GA Put to avoid any kind of inconsistencies. Inconsistencies might result due to load imbalance if GA Sync is not used after GA Put.

- **GA Get local**
  
  **Syntax:**
  
  \[
  \text{[local\_buffer]} = \text{GA\_Get\_local(\text{ArrayHandle})}
  \]

  **Description:**
  Copies data from the portion of global array owned by the calling process to the local array buffer.

- **GA Put local**
  
  **Syntax:**
  
  \[
  \text{GA\_Put\_local(\text{ArrayHandle, buf})}
  \]

  **Description:**
  Copies data from the local array buffer to the portion of global array owned by
the calling process.

Synchronization

- **GA.Sync**

  Syntax:
  
  GA.Sync()

  Description:
  
  Synchronizes processes (a barrier) and ensures that all GA operations are complete.

  Note:
  
  This is a collective operation.

- **GA.Init_fence**

  Syntax:
  
  GA_Init_fence()

  Description:
  
  Initializes tracing of completion status of data transfer operations.
Note:
This is a local operation. GA_Init_fence and GA_Fence must be used in pairs.

- GA_Fence

Syntax:
GA_Fence()

Description:
Blocks the calling process until all the data transfers corresponding to GA operations called after GA_Init_fence complete.

Note:
This is a local operation. GA_Init_fence and GA_Fence must be used in pairs.

Collectives

- GA_Fill

Syntax:
GA_Fill(ArrayHandle, value)

Description:
Assign a single value to all the elements in the array.

- **GA_Fill_patch**

  Syntax:
  
  GA_Fill_patch(ArrayHandle, lo, hi, value)

  Description:
  
  Assign a single value to all the elements in the global array patch.

- **GA_Zero**

  Syntax:
  
  GA_Zero(ArrayHandle)

  Description:
  
  Sets the value of all the elements in the array to zero.

- **GA_Zero_patch**

  Syntax:
  
  GA_Zero_patch(ArrayHandle, lo, hi)
Description:
Sets the value of all the elements in the global array patch to zero.

- **GA_Scale**

  Syntax:
  
  `GA_Scale(ArrayHandle, value)`

  Description:
  Scales an array by a constant.

- **GA_Add**

  Syntax:
  
  `GA_Add(SrcArrayHandle1, SrcArrayHandle2, DestnArrayHandle)`

  Description:
  Performs an element-wise addition of the source arrays, `SrcArrayHandle1` and `SrcArrayHandle2`, and stores the result in the destination array.

- **GA_Minus**

  Syntax:
GA_Minus(SrcArrayHandle1, SrcArrayHandle2, DestnArrayHandle)

Description:
Performs an element-wise subtraction of the source arrays, \textit{SrcArrayHandle1} and \textit{SrcArrayHandle2}, and stores the result in the destination array.

• GA_Symmetrize

Syntax:
\texttt{GA_Symmetrize(ArrayHandle)}

Description:
Symmetrizes a matrix.

• GA_Transpose

Syntax:
\texttt{GA_Transpose(SrcArrayHandle, DestArrayHandle)}

Description:
Transposes a matrix.
• GA_OP

Syntax:

GA_OP(vect, op)

Description:
Performs a reduction of the elements of the vector, vect, across all nodes using the commutative operator, op. The result is broadcast to all nodes. Supported operations include '+', '*', 'max', 'min', 'absmax', 'absmin'.

ElementWise

• GA_Abs

Syntax:

GA_Abs(ArrayHandle)

Description:
Takes the in-place absolute value of the entire Global Array.

Note:
This is a collective operation.
• **GA_Abs_patch**

  **Syntax:**
  
  GA_Abs_patch(ArrayHandle, lo, hi)

  **Description:**
  Takes the in-place absolute value of the Global Array patch specified.

  **Note:**
  This is a collective operation.

• **GA_Add_constant**

  **Syntax:**
  
  GA_Add_constant(ArrayHandle, value)

  **Description:**
  Adds the scalar, value, to each element of the Global Array.

  **Note:**
  This is a collective operation.
• **GA_Add_constant_patch**

  **Syntax:**
  
  GA_Add_constant_patch(ArrayHandle, lo, hi, value)

  **Description:**
  
  Adds the scalar, value, to each element of the Global Array patch.

  **Note:**
  
  This is a collective operation.

• **GA_Elem_multiply**

  **Syntax:**
  
  GA_Elem_multiply(ArrayHandle1, ArrayHandle2, ArrayHandleResult)

  **Description:**
  
  Computes the element-wise product of two arrays.

  **Note:**
  
  This is a collective operation.

  The arrays must be of the same shape.
• GA_Elem_divide

Syntax:

GA_Elem_divide(ArrayHandle1, ArrayHandle2, ArrayHandleResult)

Description:

Computes the element-wise quotient of two arrays.

Note:

This is a collective operation.
The arrays must be of the same shape.
The result (quotient) array may replace one of the input (dividend or divisor) arrays.

• GA_Elem_divide_patch

Syntax:

GA_Elem_divide_patch(ArrayHandle1, ArrayHandle2, ArrayHandleResult)

Description:

Computes the element-wise quotient of two array patches.

Note:

This is a collective operation.
The arrays must be of the same shape.
The result (quotient) array may replace one of the input (dividend or divisor) arrays.

- **GA_Elem_maximum**

  Syntax:   GA_Elem_maximum(Handle1, Handle2, HandleResult)

  **Description:**
  Computes the element-wise maximum of two arrays.

  **Note:**
  This is a collective operation.
  The arrays must be of the same shape.

- **GA_Elem_minimum**

  Syntax:
  
  GA_Elem_minimum(ArrayHandle1, ArrayHandle2, ArrayHandleResult)

  **Description:**
  Computes the element-wise minimum of two arrays.
Note:

This is a collective operation.

The arrays must be of the same shape.

**Ghosts**

- **GA_Create_ghosts**

  **Syntax:**

  ```
  [handle] = GA_Create_ghosts(dims, width, chunk)
  ```

  `dims` - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the size or extent of the ith dimension of the global array.

  `width` - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the ghost cell width in the ith dimension of the global array.

  `chunk` - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the minimum size that the ith dimension of the global array must be divided into among the processes.

  **Description:**

  Similar to **GA_Create**; creates a Global Array with distribution as specified by `dims` and `chunk`. Further, the local portion of the global array residing on each
processor will have a layer of ghost cells of width, $width[i]$, on either side of the visible data along the ith dimension.

Note:
This is a collective operation.

- **GA\_Create\_ghosts\_irreg**

Syntax:

```matlab
[handle] = GA\_Create\_ghosts\_irreg(dims, width, block, map)
```

dims - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the size or extent of the ith dimension of the global array.

width - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the ghost cell width in the ith dimension of the global array.

block - vector of 'n' elements where 'n' is the number of dimensions of the global array to be created and the ith element in the vector denotes the number of blocks that the ith dimension of the global array must be divided into among the processes.

map - vector indicating the starting index of each block.

Description:
Similar to \textit{GA\_Create\_irreg}; creates an array as per the user-specified distribution information (specified as a Cartesian product of distributions for each dimension). Further, the local portion of the global array residing on each processor will have a layer of ghost cells of width, $width[i]$, on either side of the visible data along the $i$th dimension.

\textbf{Note:}
This is a collective operation.

- \textbf{GA\_Has\_ghosts}

\textbf{Syntax:}
\[ \text{[handle]} = \text{GA\_Has\_ghosts(ArrayHandle)} \]

\textbf{Description:}
Returns 1 if the global array, \textit{ArrayHandle} has some dimensions for which the ghost cell width is greater than zero, returns 0 otherwise.

\textbf{Note:}
This is a collective operation.

- \textbf{GA\_Set\_ghosts}
Syntax:

[handle] = GA_Set_ghosts(ArrayHandle, width)

Description:
Sets the ghost cell widths for a global array.

Note:
This is a collective operation.

Utilities

- **GA_Distribution**

Syntax:

[lo, hi] = GA_Distribution(ArrayHandle)

Description:
Returns the index range of the global array portion owned by the calling process. If no array elements are owned by the calling process, the range is returned as \( lo[i] = 0 \) and \( hi[i] = -1 \) for \( i = 1 : ndim \) dimensions.

- **GA_Compare_distr**

Syntax:
isSimillar = GA_Compare_distr(ArrayHandle1, ArrayHandle2)

**Description:**

Compares the distributions of two global arrays.

**Note:**

This is a collective operation.

- **GA_Nnodes**

  **Syntax:**

  ```
  nNodes = GA_Nnodes()
  ```

  **Description:**

  Returns the number of user (compute) processes.

  **Note:**

  This is a local operation.

- **GA_Nodeid**

  **Syntax:**

  ```
  rank = GA_Nodeid()
  ```
Description:
Returns the rank or the process id (0 ... (np-1)) of the calling process.

Note:
This is a local operation.

• **GA Check handle**

Syntax: `isValid = GA Check handle(ArrayHandle)`

Description:
Checks if the ArrayHandle is valid.

Note:
This is a collective operation.

**Process Groups**

• **GA Create pgroup**

Syntax:
`pGrpHandle = GA Create pgroup(list)`

Description:
Creates a processor group given the list of process ids and returns a process group handle.

**Note:**
This is a collective operation.

- **GA_Pgroup_nnodes**

  **Syntax:**
  
  \[ \text{nNodes} = \text{GA_Pgroup_nnodes(pGrpHandle)} \]

  **Description:**
  Returns the number of processors contained in the group specified by `pGrpHandle`.

  **Note:**
  This is a local operation.

- **GA_Pgroup_nodeid**

  **Syntax:**
  
  \[ \text{rank} = \text{GA_Pgroup_nodeid(pGrpHandle)} \]

  **Description:**
Returns the relative index of the calling process in the processor group specified by \textit{pGrpHandle}.

\textbf{Note:}

This is a local operation.

- \textbf{GA\_Pgroup\_sync}

\textbf{Syntax:}

\texttt{GA\_Pgroup\_sync(pGrpHandle)}

\textbf{Description:}

Synchronizes processes (a barrier) in the process group specified by \textit{pGrpHandle} and ensures that all GA operations for the process group are complete.

\textbf{Note:}

This is a collective operation.
BIBLIOGRAPHY


[36] RTExpress. Integrated Sensors Inc.


