Generalized Portable SHMEM library for high performance computing

Krzysztof Parzyszek
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
http://lib.dr.iastate.edu/rtd/1455

This Dissertation is brought to you for free and open access by Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Generalized Portable SHMEM library for high performance computing

by

Krzysztof Parzyszek

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Ricky A. Kendall, Co-major Professor
Robyn Lutz, Co-major Professor
Gary Leavens
Mark Gordon
Bruce Harmon

Iowa State University
Ames, Iowa
2003
This is to certify that the doctoral dissertation of
Krzysztof Parzyszek
has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Co-major Professor

Signature was redacted for privacy.

Co-major Professor

Signature was redacted for privacy.

For the Major Program
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Acknowledgments</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Introduction</td>
<td>1</td>
</tr>
<tr>
<td><strong>PART I   The GPSHMEM library</strong></td>
<td>2</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2 Shared memory model</td>
<td>5</td>
</tr>
<tr>
<td>2.1 The concept</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Uniprocessor systems</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Hardware support</td>
<td>7</td>
</tr>
<tr>
<td>2.4 Software implementations</td>
<td>9</td>
</tr>
<tr>
<td>2.5 Programming languages</td>
<td>12</td>
</tr>
<tr>
<td>2.6 Other software DSM projects</td>
<td>15</td>
</tr>
<tr>
<td>2.7 History of SHMEM</td>
<td>18</td>
</tr>
<tr>
<td>2.8 Motivation for GPSHMEM</td>
<td>18</td>
</tr>
<tr>
<td>3 Introduction to SHMEM and GPSHMEM</td>
<td>21</td>
</tr>
<tr>
<td>3.1 Collective and one-sided communication</td>
<td>21</td>
</tr>
<tr>
<td>3.2 Processing elements</td>
<td>22</td>
</tr>
<tr>
<td>3.3 Symmetric data objects</td>
<td>22</td>
</tr>
<tr>
<td>3.4 Data types</td>
<td>23</td>
</tr>
<tr>
<td>3.5 Active sets</td>
<td>24</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.6 Workspace arrays</td>
<td>24</td>
</tr>
<tr>
<td>3.7 Cray pointers</td>
<td>25</td>
</tr>
<tr>
<td>4 The GPSHMEM API reference</td>
<td>28</td>
</tr>
<tr>
<td>4.1 One-sided communication</td>
<td>29</td>
</tr>
<tr>
<td>4.1.1 Swap</td>
<td>29</td>
</tr>
<tr>
<td>4.1.2 Get and put</td>
<td>29</td>
</tr>
<tr>
<td>4.1.3 Strided get and put</td>
<td>30</td>
</tr>
<tr>
<td>4.1.4 Gather/scatter</td>
<td>31</td>
</tr>
<tr>
<td>4.2 Collective routines</td>
<td>32</td>
</tr>
<tr>
<td>4.2.1 Barrier</td>
<td>32</td>
</tr>
<tr>
<td>4.2.2 Broadcast</td>
<td>33</td>
</tr>
<tr>
<td>4.2.3 Collection</td>
<td>33</td>
</tr>
<tr>
<td>4.2.4 Reduction</td>
<td>35</td>
</tr>
<tr>
<td>4.3 Miscellaneous functions</td>
<td>36</td>
</tr>
<tr>
<td>4.3.1 Fence</td>
<td>36</td>
</tr>
<tr>
<td>4.3.2 Wait</td>
<td>36</td>
</tr>
<tr>
<td>4.3.3 Time</td>
<td>37</td>
</tr>
<tr>
<td>4.3.4 Error</td>
<td>37</td>
</tr>
<tr>
<td>4.3.5 PE queries</td>
<td>38</td>
</tr>
<tr>
<td>4.3.6 Cache control functions</td>
<td>38</td>
</tr>
<tr>
<td>4.3.7 shmem_ptr</td>
<td>38</td>
</tr>
<tr>
<td>4.4 Memory allocation</td>
<td>38</td>
</tr>
<tr>
<td>4.5 Initialization and finalization</td>
<td>40</td>
</tr>
<tr>
<td>5 Implementation</td>
<td>41</td>
</tr>
<tr>
<td>5.1 ARMCI</td>
<td>41</td>
</tr>
<tr>
<td>5.2 Message passing libraries</td>
<td>43</td>
</tr>
</tbody>
</table>
5.3 GPSHMEM .................................................. 43
5.4 Memory allocator ........................................... 46
  5.4.1 Memory address translation .......................... 46
  5.4.2 Fortran 77 memory allocation ....................... 48

6 Performance .................................................. 52
  6.1 The Splash-2 suite ....................................... 52
  6.2 IBM-SP .................................................... 53
  6.3 The benchmarks .......................................... 53
    6.3.1 Scalability tests .................................. 54
    6.3.2 Problem size timings ................................ 55
  6.4 FFT ...................................................... 55
  6.5 LU decomposition ........................................ 57
  6.6 Radix sort ............................................... 58
  6.7 Remarks ................................................ 58
    6.7.1 Benchmark failures ................................ 58
    6.7.2 G4 cluster .......................................... 59

7 Performance graphs ......................................... 60
  7.1 FFT ..................................................... 60
    7.1.1 Varying tasks_per_node ............................ 60
    7.1.2 Varying nodes ..................................... 64
    7.1.3 Varying problem size .............................. 68
  7.2 LU decomposition ........................................ 73
    7.2.1 Varying tasks_per_node ............................ 73
    7.2.2 Varying nodes ..................................... 77
    7.2.3 Varying problem size .............................. 81
  7.3 Radix Sort .............................................. 86
ACKNOWLEDGMENTS

This work was performed under auspices of the U. S. Department of Energy under contract W-7405-Eng-82 at Ames Laboratory operated by the Iowa State University of Science and Technology. The United States government has assigned the DOE Report number IS-T 1938. Funding was provided by the Mathematical, Information and Computational Science division of the Office or Advanced Scientific Computing Research. This research was performed in part using computational resources of the National Energy Research Scientific Computing Center. GPSHMEM, together with many other projects, is a component of the Programming Models effort of the U.S. Department of Energy aimed at providing the scientific community with a set of tools for efficient use of parallel computational resources.

I am grateful to Hongzhang Shan, who provided us with the SHMEM versions of the Splash-2 codes that we adopted to GPSHMEM. I am also grateful to Jarek Nieplocha and Vinod Tipparaju who develop and maintain ARMCI. I would like to express my gratitude to the International Business Machines Corporation (IBM) for awarding me with the Cooperative IBM PhD Fellowship for the academic year 2001–2002.

The thesis was prepared using the \LaTeX{} and \bibtex{} packages. Many of the entries for the bibliography database were taken from CiteSeer—The Scientific Literature Digital Library and the \bibtex{} file with Rasit Eskicioglu's "Comprehensive Bibliography of Distributed Shared Memory."

I would also like to thank my friend, Irene Faass, who helped me with editing this dissertation.
Last, but not least, I would like to express my gratitude to the committee members, especially Dr. Ricky Kendall, who provided me with continuous help and support, without which this project would not have succeeded.
GENERAL INTRODUCTION

This dissertation describes the efforts to design and implement the Generalized Portable SHMEM library, GPSHMEM, as well as supplementary tools. There are two major components of the GPSHMEM project: the GPSHMEM library itself and the Fortran 77 source-to-source translator. The rest of this thesis is divided into two parts. Part I introduces the shared memory model and the distributed shared memory model. It explains the motivation behind GPSHMEM and presents its functionality and performance results. Part II is entirely devoted to the Fortran 77 translator called fgpp. The need for such a tool is demonstrated, functionality goals are stated, and the design issues are presented along with the development of the solutions.
PART I

The GPSHMEM library
1 Introduction

One of the most important aspects of programming in distributed environments is the communication scheme, the data movement within an application. One of the programming models is the shared memory programming model introduced in chapter 2 and, in particular, distributed shared memory model. The essence of the shared (and distributed shared) memory model is one-sided communication, often referred to as remote direct memory access (RDMA). Various implementations of this programming model have been developed, but the *de facto* standard in the scientific computing community is the SHMEM library introduced by Cray Research Inc. in 1994 for their supercomputer, Cray T3D. There has been a large number of computational applications written specifically for use with SHMEM. SHMEM, however, was tightly coupled with the Cray hardware and thus only available on Cray machines. In the rise of clustered systems, ability to efficiently solve scientific problems in such environments has been of an increasing importance, however, inability to run legacy code hindered the growth of the use of clusters.

GPSHMEM was poised to achieve two main goals: allow easy transition of the scientific applications from the Cray computers and provide an alternative software-based implementation of the (distributed) shared memory programming model.

The contribution of this work is a portable implementation of SHMEM. Its availability is increased by, besides portability, the fact that it is free and open-source. It can serve as a ready-to-use product as well as a basis for more specialized implementations of SHMEM.
The rest of this report describe our design and implementation efforts. Chapter 2 introduces the shared memory programming model in more detail. Related work implementing both similar and alternative solutions is presented in sections 2.2–2.6. Next, the background information about SHMEM and motivation for GPSHMEM is described in sections 2.7 and 2.8, followed by a detailed reference to the GPSHMEM application programming interface in chapter 4. Chapter 5 discusses GPSHMEM's implementation issues, including ARMCI, the library that effectively performs the one-sided communication of GPSHMEM. Chapters 6 and 7 describe the results of the performance tests. Next, GPSHMEM verification suite is briefly presented in chapter 8 followed by the discussion of some practical aspects of the C and Fortran interoperability in chapter 9. Finally, conclusions and possible future work are presented in chapter 10.
2 Shared memory model

This chapter presents a general introduction to the shared memory programming model. First, the general concept is described with some historical notes on its development. Various attempts to implement this programming model both in hardware and software are described. These implementations, except, of course, for the original SHMEM library, do not have a direct impact on the design and development of GPSHMEM. However, they offer alternatives to SHMEM and GPSHMEM and should be considered when deciding on the particular solution in a new programming project. Two major kinds of software-based solutions are described: those in which the shared memory model is achieved by a standalone libraries and that in which it is implemented via extensions to existing programming languages. Finally, other related software-based approaches are discussed. The basic presentation of SHMEM together with motivation and goals for GPSHMEM conclude this chapter.

2.1 The concept

In the shared memory programming model, different threads of execution of a given program share a part of their address spaces. A "thread of execution" can be a process, a thread (also known as a lightweight process), a kernel scheduling entity [1], or whatever the operating system has to offer. Conceptually, the stores to a given location in a shared memory are immediately visible to all other processes. In practice, this may or may not be true and depends on several factors, such as cache coherency, communication
links, etc. In some situations, explicit synchronization calls are necessary. In no case is there a need for both sides to actively participate in the communication. The essence of the shared memory programming model is taking advantage of the asynchronous, nonblocking data passing mechanism.

Sometimes, especially in case of implementations of shared memory on distributed systems, the remote memory regions do not appear directly accessible via pointers. In such cases, a program cannot perform computations on the remote objects. A separate library call must be made to explicitly fetch data from the remote location to a local buffer, or to store the contents of a local buffer into an area of the remote memory. Nevertheless, the programming model dictates that only the initiator of the data movement is involved in the transfer and the shared memory model principles are thus preserved. Such shared memory models are often called distributed shared memory.

2.2 Uniprocessor systems

Early computers were not working in networked environments. Many of them eventually ran multitasking operating systems allowing simultaneous execution of several programs. All of their processes were located in different parts of the same physical memory. Segments of the physical memory were associated with a given program and appeared to it as a part of a continuous address space. The memory mapping mechanisms of multitasking processors usually allowed the same region of the physical memory to be mapped into the address space of more than one process. This way a shared memory model had its very natural implementation. One of the most popular programming interfaces for shared memory was the one initially implemented in Unix System V and most widely known as SysV shared memory. The shared memory programming model was also sanctioned by the IEEE when it became a part of the Portable Operating System Interface, described in ANSI/IEEE Std. 1003.1b-1993, commonly referred to as
POSIX.1b [69]. The POSIX interface differs, however, from the SysV interface, although conceptually they are similar. To share a part of the address space using the SysV interface, an application would first obtain a shared memory segment identifier using `shmget` and later attach that segment into the address space with `shmat`. The POSIX interface provides a function `shm_open` that returns a file descriptor suitable for use with `mmap`.

2.3 Hardware support

Most current parallel systems have distributed memory. They are either explicitly distributed memory machines or, more often, clusters of machines or computational nodes connected with a fast network. Our interest will be focused on such systems and we will assume systems have distributed physical memory in the rest of this discussion.

Initially, the only parallel computers were high-performance systems designed and manufactured by a small group of hardware vendors. Those systems usually consisted of physically separate nodes connected with a fast and usually proprietary interconnect allowing fast data transfers from one node to another. Some of these systems implemented a Cache Coherent NUMA\(^1\) memory model. They were equipped with specialized hardware allowing simulation of shared memory on their distributed memory architectures. An example of such an approach is the Cray T3D and T3E. Each node has its own processor and its own memory. Inter-node memory accesses are facilitated by an External Register Set, commonly known as E-Registers. Such hardware would be of no use if there were no software allowing the programmers to take advantage of it. Cray Research Inc. released a SHMEM library [7] that has become a de facto standard programming interface for distributed shared memory. Implementations of SHMEM were later provided by other hardware vendors, in particular by Silicon Graphics Inc. and their SGI Origin 2000 and the recent SGI Origin 3000 systems. The Crays and SGI Origins were

\(^1\)Non-Uniform Memory Access
not the only systems supporting shared memory. The HP-Convex Exemplar is a system with distributed shared memory, that was facilitated by the special Compiler Parallel Support Library, known as CPSLib. The IBM’s Scalable Powerparallel, commonly referred to as IBM-SP was a cluster of RS/6000 nodes connected with a high-performance interconnect switch. The communication features of IBM-SP were made available to the programmer by a proprietary, but published Low-level Application Programming Interface (LAPI [112]).

All of these hardware solutions used proprietary technologies and required different software libraries to provide an appropriate application programming interface. With time, some industry standards have emerged to unify hardware architectures. The ANSI/IEEE document number 1596-1992, “Standard for Scalable Coherent Interface” [95] defines a communication interconnect and a set of accompanying protocols. The SCI protocols allow direct memory sharing by registering memory regions with the interconnect. Some existing implementations of SCI-enabled network hardware include the Wulfkit system manufactured by Dolphin Interconnect Solutions Inc. There also exist implementations of the Message Passing Interface and SHMEM for SCI systems, in particular ScaMPI [59] and ScaShmem [58] from Scali.

Many clusters use TCP/IP-based communication libraries. The TCP/IP stacks are available for most networking hardware, including Fast Ethernet, Gigabit Ethernet and Myrinet. Since most of the time spent in the processing of the TCP/IP packets, besides the operating system overhead, is attributed to memory-to-memory copying of the message buffers [28], an attempt was made to eliminate the copying—a technique known as zero-copy sockets [27]. Another way to reduce the transfer penalty was to bypass the complicated protocol mechanisms that are known not to be used in given environments. Those could include routing algorithms, and defragmentation routines as well as the aforementioned buffer copying. Such efforts were summarized in an industry standard known as Virtual Interface Architecture (VIA) [29]. The Virtual Interface Architecture
provides a process with a direct access point to a networking interface—a Virtual Interface. The data transfers made through such an interface are not subject to the usual processing and so the network latency is reduced. The software API presented in the VIA specification is provided by the Virtual Interface Provider Library (VIPL) described in the Virtual Interface Architecture Developer's Guide [70]. A part of the PC Cluster Project developed at the National Energy Research Scientific Computing Center was M-VIA [90], a VIA implementation for Linux.

Among other technologies trying to maximize network performance and offering Remote Direct Memory Access capabilities are Fibre Channel [9], and InfiniBand [62]. To address the incompatibility issues, some efforts have been done to standardize the network architectures supporting RDMA operations. The RDMA Consortium [107] has proposed a set of specifications including Remote Direct Memory Access Protocol, Direct Data Placement protocol, and guidelines for adaptation of those protocols with the Stream Control Transmission Protocol (STCP) [116]. The STCP is a network protocol extending the capabilities of TCP and UDP. One of the most important of these additions is support for multiple independent data streams within the same connection. In order to provide a programmer with a standardized, platform- and transport-independent set of functions for RDMA operations, the User Direct Access Library (uDAPL) [38] and the Kernel Direct Access Library (kDAPL) [37] were proposed by the DAT Collaborative [39].

2.4 Software implementations

"Truly" shared memory, that is, such that it would appear as shared from the point of view of a C programmer, is not easy to implement without any support from the underlying hardware or the compiler, or both. There is a "relaxed" model of shared memory, in which the remote memory cannot be directly manipulated via pointers, but
can be accessed by using special data transfer functions. The relaxed model of shared memory, the distributed shared memory model, is often implemented by providing agents who reside on all participating nodes and handle requests for remote memory operations asynchronously and without explicit cooperation from the passive side of the transaction. The most basic way to implement distributed shared memory in a portable manner is to use Remote Procedure Calls (RPC). Each node in the network would provide an RPC server and a RPC handler to perform the actual data transfers. Similar in concept are Active Messages [119], developed at the University of California at Berkeley. Active Messages were used to implement explicit parallelism constructs in Split-C [36]. One of the main components of the High Performance Virtual Machines project undertaken at the University of Illinois at Urbana-Champaign is the Fast Messages [97] library. Along with Fast Messages, there is FM-Shmem, a SHMEM interface built on top of the Fast Messages [51]. The FM-Shmem, however, does not implement the full functionality of the original Cray SHMEM and therefore is not a viable porting tool for applications developed on Cray systems.

As a part of the programming toolkit for distributed memory machines, the Pacific Northwest National Laboratory has developed the Global Arrays (GA) library [93]. The GA provides the programmer with an ability to create arrays whose contents spanned across different nodes. Each process could have its own “local” part of the array and accesses to the locally present data were faster than accesses to the remote locations. In that respect GA offered an implementation of the NUMA memory model on distributed memory systems. A process was, however, unable to directly manipulate remote data. A nonlocal part of the array would have to be fetched to a local buffer and, after the computations have taken place, the results would need to be explicitly transferred to their desired remote destinations. The communication abilities of the Global Arrays toolkit were built upon another library created at the PNNL, the Aggregate Remote Memory Copy Interface or ARMCII [92]. ARMCII provides a concise programming interface
capturing the essence of the distributed memory programming model. It implements explicit one-sided remote memory access operations such as read, write or atomic swap. The communication model implemented by ARMCI strongly resembles the one delivered by SHMEM. This was one of the major factors that made us choose ARMCI to be one of the underlying libraries for the GPSHMEM project. ARMCI will be presented in more detail in section 5.1. Similar functionality to that of Global Arrays was implemented in the Distributed Data Interface (DDI) [45]. DDI has been developed and used to support GAMESS [110], the quantum chemistry code from Mark Gordon's research group at Iowa State University.

One-sided communication, (i.e. communication occurring without explicit handshake, see section 3.1) has been recognized as an important programming model, and this recognition was officially sanctioned by the MPI Forum [88] with the release of the MPI-2 standard [87]. The MPI Forum is a collaborative effort of over 40 companies, national laboratories and academic institutions to reach a consensus on a uniform application programming interface for communication in distributed systems. MPI stands for "Message Passing Interface" and the first version of the MPI document described only a two-sided, message passing programming model. Although, according to the MPI-1 standard [86], the MPI Forum is not supported by any official standards organization, the documents resulting from its meetings are recognized as industry standards by nearly all high-performance hardware vendors.

MPI-2 was designed to run on as large variety of hardware systems as possible. The MPI-2 specification avoided creating situations which would require special features from the underlying equipment. Therefore many restrictions had to be placed on the one-sided communication protocol. All remote memory access (RMA) operations in MPI-2 can only be performed within specially designed memory areas, called windows. Depending on the exact nature of the data transfers there are additional requirements specifying the time frames during which such communication can occur. In all cases, the state of
the windows is not guaranteed until a special synchronization call is made. Chapter 6 of the MPI-2 document [87] contains detailed description of the one-sided interface. The rationale presented therein makes an analogy between the remotely accessible windows and the cache pages in a distributed shared memory systems. Many restrictions placed upon the communication mechanism stem from attempts to maintain memory coherency without expecting too much from the hardware. The burden of keeping the memory state consistent is therefore placed in large part on the programmer. The one-sided communication interface of MPI-2 can provide the programmer with the facilities to utilize the distributed shared memory programming model, however it is too complicated and cumbersome to be a good candidate for porting applications that use SHMEM [15]. The complexity of MPI-2 may also defer creation of full-fledged implementations as well as deter programmers unfamiliar with it from applying it to their application software.

2.5 Programming languages

Shared memory libraries, either stand-alone or built on top of particular hardware, provide distributed shared memory functionality that can be used with existing programming languages, usually Fortran, C and C++. Programs using these libraries do not need any special support from language implementation and thus are compiler-independent. It is questionable however, whether this results in increased portability, as the underlying hardware and library requirements must still be met. Moreover, such a program is tightly coupled with a given library, especially when it is not widely recognized and does not have many implementations. The coupling is caused by adjusting the programming model to meet the specification of the library. MPI-2 would be a good example: a program using MPI-2 efficiently would perform numerous function calls to MPI-2 routines and adjust its structure to meet the protocol requirements that MPI-2 imposes. Because of that, in addition to choosing the right programming model for a
given application, the developers often include the set of available libraries and their features as a factor when deciding on the implementation details.

To reduce the number of constraints placed on the development process, a direct language support for explicit sharing of distributed memory can be implemented. The common concept for many such approaches is that of a Global Address Space (GAS) [117]. The GAS assumes an SPMD\(^2\) model: multiple instances of a single program image are executed in parallel. The address space is globally accessible; however, parts of it are owned by individual processes. One example of such a language is Split-C [36], mentioned in the discussion of Active Messages. Split-C is an extension of the C programming language. It adds global pointers to the language. A global pointer is a pointer to an object owned entirely by another process. Split-C also has a split-phase assignment, breaking an assignment up into two parts: the request for data transfer and the completion thereof, as well as signaling stores to locally owned objects, spread arrays and several other features.

An actively developed descendant of, among others, Split-C, is Unified Parallel C or UPC [19]. UPC is a joint effort of several research institutions, including University of California at Berkeley, Lawrence Berkeley National Laboratory, Lawrence Livermore National Laboratory, George Washington University and the IDA Center for Computing Sciences. UPC introduces a new type qualifier, shared, indicating that a given object is to be shared across different processors. Such an object is divided into fragments, each of which is owned by a single node. The division can be, to some extent, controlled by the programmer. UPC defines semantics for pointers pointing to shared objects, adds synchronization methods and a loop keyword, forall, allowing the programmer to avoid explicit ordering of operations, and thus also allow the compiler to make appropriate optimizations.

Apart from C and C++, the global address space concept has been implemented

\(^2\text{Single Program, Multiple Data.}\)
in Titanium [122], a high-performance dialect of Java. Titanium, like UPC, is based on the SPMD model. Major extensions of the Java language include multidimensional arrays allocated via domains and points, immutable classes that can be passed by value, local and global references, and allocation regions. A domain is a set of coordinates of array elements. In case of rectangular domains, points specify the corners of the domain. Similar to UPC, an unordered access to elements of a domain can be facilitated by a foreach construct. Titanium offers several more extensions to the language, all of which are described in the reference manual [55].

A slightly different approach was used in the Co-Array Fortran [94]. Co-Array Fortran is a set of extensions to Fortran 95 [66]. Like the other languages described above, it also assumes the SPMD execution model: there are multiple images of the same executing program, and those images can share arrays of data. Unlike UPC and Titanium, however, those arrays are duplicated at each image. Each process can directly access the copy of the array owned by any other process. An additional syntactical feature is used to specify the program image whose copy of the given array needs to be accessed.

The GAS paradigm can be implemented directly on top of the specific development environment available on a given platform. This, however, would make such implementations nonportable. Thus, there was a need for a unified way to access the communication facilities, independent from the actual hardware and any system-specific software. An emerging industry standard, MPI-2 with its one-sided interface, may seem like a good candidate for the underlying library, however, due to the complexity of its usage and the strict requirements that the defining document imposes on the conforming programs, it turns out not to be an acceptable solution [15]. To address this issue and to provide a common implementation foundation for GAS-oriented languages, a GASNet library [14] was proposed by the University of California at Berkeley. GASNet provides a rich application programming interface connecting the features of Active Messages with the asynchronous, one-sided communication mechanisms found in SHMEM and ARMCI.
Unified Parallel C, Titanium, Co-Array Fortran, as well as ARMCI, Global Arrays and GPSHMEM are all components of the Programming Models effort of the U.S. Department of Energy aimed at providing the scientific community with a set of tools for the efficient use of parallel computational resources. Under this effort the participants are in the process of developing a common runtime environment that will facilitate each of these programming models on a wide variety of current architectures, as well as on future offerings from the vendor community.

2.6 Other software DSM projects

The concept of distributed shared memory on distributed memory machines is not new. Various solutions have been proposed to provide access to the non-local memory, either directly via pointers or indirectly, using function calls. The implementations vary from those strongly relying on hardware support to those that are purely software-based. Both of these approaches have their pros and cons. There seems to be a tradeoff between performance and portability on one hand, and simplicity and availability on the other. Hardware-based solutions usually rely on the interconnect to resolve memory coherency problems, at least to some extent. While the circuitry can be complicated, the supporting software no longer has to worry about coherency problems. Software-based approaches are not in such a comfortable situation and various steps must be taken to avoid the problems stemming from physical distribution of memory. Various memory coherency protocols have been proposed [84] [73]. The simplest of these is the strict coherency model, in which a read operation at a given process returns the value that was most recently written to that memory location. It is worth pointing out that the most recent value is not necessarily that which the given process has stored.

This consistency model was used in the first software based implementation of the distributed shared memory model, Ivy [83], developed by Kai Li from Princeton Uni-
versity. Strict coherency implies that all memory updates are immediately visible to all other processes. There is usually a performance penalty associated with the immediate propagation of all stores, therefore, the Munin [10] system introduced delayed updates. The authors of Munin have defined a loose coherence model: reads of a given object return a value that could be a result of some allowed process execution schedule. Additionally, to improve the performance and provide the most suitable treatment, objects have been classified according to their access patterns; there are, for example, producer-consumer objects or read-mostly objects.

Another consistency model was proposed in Midway [12]. A set of shared data objects can be associated with a synchronization object, or a lock. The updates of the data objects should occur only within critical sections guarded by that lock. The updates are finalized when the lock is acquired upon entry to such critical sections. This scheme is an entry consistency memory model. The Midway project consisted of a runtime environment and a compiler implementing a set of extensions to C, C++ and ML.

An alternative to Midway with its entry consistency was TreadMarks [77]. TreadMarks, similarly to Munin, implemented a release consistency model [50]. In the release consistency model a synchronization access (either acquire or a release) to the shared memory can occur. The prior memory updates are propagated when the release access is made. The lazy release consistency postpones the updates even further, until the next acquire operation. One of the differences between TreadMarks and Munin is the lazy diff creation: both, Munin and TreadMarks, create a modification record for each updated page; however, TreadMarks delays creation and propagation of these records until they need to take effect. The lazy release consistency model has captured the attention of the developers of the SHRIMP [13] multicomputer at Princeton University. SHRIMP provides some hardware support for point-to-point communication. This feature has been utilized in the concept of automatic updates, that is automatic propagation of local writes to remote nodes [60]. The result of merging the hardware support with
the release consistency model is called automatic update release consistency (AURC). AURC is a combination of two protocols, Copyset-2 and Copyset-N, that define the automatic update mappings for pages shared between two and an arbitrary number of nodes, respectively.

Another, purely software-based, distributed shared memory implementation is the C Region Library [71]. CRL intends to be fully portable and does not require support from either the hardware or the compiler. In the CRL model, processes share data via regions. A run-time library provides a set of functions to create, manage and destroy regions. Preparation of a region resembles allocation of a shared memory block in the SysV or POSIX interface. An allocation must be followed by mapping the region into the local memory. An important distinction from the "pure" shared memory model is that operations on the shared data must be grouped and delimited by calls to appropriate CLR functions. Such groups are treated as indivisible operations and thus CLR implements the sequential consistency model. Yet another approach was undertaken in Shasta [109]. Shasta is an all-software implementation of distributed shared memory. This allows for portability and flexibility in incorporating various coherency protocols. Unlike CLR, Shasta relies on a special compiler support. The compiler intercepts accesses to the shared data objects and generates code that invokes appropriate synchronization mechanisms, if necessary.

The variety of implementations of distributed shared memory models is quite large. They vary from all-software to those requiring special hardware. They all differ from SHMEM, though. SHMEM is a simple, efficient interface to the remote memory access facilitated by Cray's hardware. There is no coherency protocol [100], as there is no such protocol in the local shared memory model. This caused cache coherency problems on the Cray T3D, hence the first release of SHMEM contained a set of functions for explicit cache invalidation. The hardware of the Cray T3E solved this issue and explicit programmer's intervention was no longer necessary. Due to the lack of consideration
of complicated consistency issues, SHMEM delivers unsurpassed performance. It leaves synchronization control to programmers thus relieving them from often unnecessary overheads induced by any automatic coherency control.

2.7 History of SHMEM

In 1993 Cray Research Inc. introduced its first Massively Parallel Processing (MPP) computer, the Cray T3D. The architecture of the Cray T3D followed the non-uniform memory access (NUMA) model. The system's memory was physically distributed among processing elements (PE), but each processing element could access memory belonging to other processing elements. The remote memory access was facilitated by a special hardware feature known as E-registers. An E-register is a memory mapped register allowing the programmer to request an asynchronous remote memory transfer, either put or get. To enable programmers to take advantage of these features without resorting to low-level hardware manipulation a Shared Memory Access library, better known as SHMEM, was created and released in 1994. SHMEM offered a high-level interface to the underlying hardware capabilities on Cray MPP and PVP systems. According to the SHMEM User's Guide for C [7], among the main design principles of SHMEM were speed, small size and expandability.

2.8 Motivation for GPSHMEM

In the 1990s, the supercomputers created by Cray Research Inc. were among the fastest and the most powerful machines that the scientific community had. The unmatched computational power achieved by parallel architecture and low latency interconnect attracted scientists from nearly all disciplines. Their research projects, such as weather prediction, simulations of chemical reactions, or protein folding, to name a few, would consume all computational power available. The limiting factor for the quality of
their results was the volume of data that could get processed within an acceptable time frame.

Because of that, just having a very fast computer was not enough. An application that could use every free cycle of the machine was necessary. When Cray Research Inc. introduced the Cray T3D, the full potential of its distributed shared memory could be utilized by using the SHMEM library. Since then, SHMEM has become a de facto standard application programming interface for the distributed shared memory programming model.

With the development of networking technology and the declining prices of small workstations, clusters of inexpensive computers have become an interesting alternative for the scientific community. Although the computational power of “real” supercomputers was still not matched, the relatively low cost and high availability made those systems very attractive. There was still one component missing, however. The purpose of hardware is to execute software. The scientific software had been written and used on specialized machines, and it often used vendor-specific solutions to achieve the most from the underlying architecture. SHMEM was no exception. While it was often praised for its performance, its lack of portability was frequently listed among its greatest disadvantages. A portable implementation was needed and thus the GPSHMEM project was started. The name GPSHMEM means “Generalized Portable SHMEM” and its main goal was to be an open source, portable implementation of the SHMEM programming interface. GPSHMEM addresses several issues:

1. Provide a way of easily porting the existing software from distributed shared memory machines to other platforms and architectures.

2. Provide an implementation of the distributed shared memory programming model on systems that offered no hardware support for this model. A new portable asynchronous implementation of the programming model would widen the variety of
possible approaches to solve a given problem. In addition to that, programmers fa-
miliar with SHMEM could directly apply their experience to new projects without
a learning new tools. As experience is important for the optimal use of available
resources, some mistakes and suboptimal solutions could be avoided.

3. Allow the development of intricate applications that can be tuned for architectures
that range from clusters to high-end MPPs.

4. Provide a Fortran interface to the one-sided ARMCI functionality.
3 Introduction to SHMEM and GPSHMEM

The SHMEM and GPSHMEM libraries are presented in more detail in this chapter. The one-sided communication, which is the essence of SHMEM programming is introduced. All concepts necessary for understanding SHMEM and GPSHMEM programming model are defined.

3.1 Collective and one-sided communication

As an implementation of a distributed shared memory model, SHMEM offers the means to perform remote memory access (RMA) operations. Such operations are one-sided, that is, only the process actively transferring data is involved in the transaction. The owner of the remote memory is not required to perform any actions for such a transfer to complete successfully; moreover, there can be scenarios in which it would not be able to detect attempts to access its memory from other processing elements. Since the motivation for SHMEM was taking advantage of the hardware features that allow efficient implementation of such data transfers, one-sided communication is a major component of SHMEM's application programming interface. The functionality of SHMEM does not end with such communication methods, however. SHMEM offers a relatively rich set of collective routines, including broadcast, collection and reduction routines. As the name suggests, the collective routines involve more than one processing element. These functions are blocking, that is they do not return until the execution completes locally. The local completion does not, however, imply completion of the routine on the
other processing elements, i.e. it is not a true barrier.

3.2 Processing elements

A processing element, often abbreviated as PE, is a single computational unit of a Cray T3D and Cray T3E. It consisted of a single processor and accompanying memory. The total number of PEs in the Cray systems was not fixed and depended on the particular computer. From the application’s point of view, a processing element appeared as a separate processor. For every program, the operating system allocated a set of processing elements that would execute it. The SHMEM’s execution model was SPMD. This means that all processing elements running given application executed the same program. Each PE within the set of processing elements associated with a given application was assigned an identifier. The identifiers were integers from 0 to $N - 1$, if $N$ was the number of PEs in the set. There were system-specific functions which allowed the program to obtain both the number of allocated PEs and the identifier of the PE on which it was running.

3.3 Symmetric data objects

SHMEM does not expose the entire memory of one processing element to others. It defines a concept of a symmetric data object also known as a remotely accessible data object. An object is symmetric if it exists on all processing elements and has the same size and address everywhere. There are several ways in which such constraints can be satisfied. The first is to use a special memory allocation function, shmalloc. This function is only available in C and C++. Fortran 90 and 95 users can use shpalloc instead. Those using Fortran 77 are left with only statically allocated objects:

1. members of common blocks,
2. objects with the SAVE attribute.

GPSHMEM uses ARMCI for one-sided communication. Since ARMCI can only operate on the memory that it had allocated itself, the same restrictions apply to GPSHMEM. Therefore all objects that can be used as remotely accessible objects in GPSHMEM must have been previously allocated by a call to GPSHMEM’s gpshmalloc function. This means that the usual Fortran 77 programs cannot take much advantage of the GPSHMEM’s functionality. These problems have been addressed and two solutions are proposed: GPSHMEM’s memory allocator, described in section 5.4, and the Fortran 77 translator/preprocessor, to which the entire part II is devoted.

3.4 Data types

The first edition of SHMEM, released on the Cray T3D had a somewhat limited set of data types used for communication. The two main data types recognized by SHMEM functions were words: 64- and 32-bit. Every object to be transferred via SHMEM functions was therefore treated as a sequence of words of appropriate size. The size of the word had to be chosen based on the object’s alignment in memory and on the alignment of its target location. GPSHMEM follows SHMEM in this matter, but it does not restrict itself to only word-based transfers. We have added two more functions from the later version of SHMEM on Cray T3E: gpshmem_getmem and gpshmem_putmem that allow data transfers with byte granularity.

An exception to this limitation were the reduction routines. The reduction operations included arithmetic sum, product, minimum, maximum, for types short, int, float and double as well as bitwise and, or and xor for the integral types. GPSHMEM has extended the available set of types by adding types long and long long to the list. The type long long was not mandated by the previous C standard (known as C89 [64]). It has been introduced in C99 [68], but has existed in many implementations as
an extension. GPShMEM's build process can detect the presence of this type and the functions using it are only enabled if the type long long is supported by the compiler.

3.5 Active sets

One-sided communication in SHMEM takes place between only two processing elements, one of which, the passive one, is not actively involved. The collective routines engage a whole group of processing elements and so there must be a way to specify exactly which of the processing elements are to participate in a given transaction. Such a subset is, in SHMEM's terminology, called an active set. Not every subset of the set of all processing elements can be an active set. The identifiers of the processing elements in an active set must be equally strided and the stride must be a power of 2. For all collective routines, the active set is defined in the same, uniform way, using three parameters:

1. **PE_start**: the identifier of the first processing element,

2. **logPE_stride**: the logarithm base 2 of the stride between the PE identifiers,

3. **PE_size**: the number of PEs in the active set.

The requirement for the stride between processing elements in an active set to be a power of 2 seemed unnecessarily restrictive. GPShMEM allows active sets with arbitrary strides. Since the SHMEM interface does not leave room for such extension, an additional set of functions has been added. The GPShMEM functions accepting arbitrary stride have names ending with _st.

3.6 Workspace arrays

SHMEM's collective functions use additional *workspace arrays*. Most of them require only one, *sync* workspace, but the reduction routines take two arrays: *sync* and *work*. 
These arrays must be remotely accessible and large enough to hold SHMEM's temporary data. The exact requirements are specified in the SHMEM manual [7]. GPSHMEM does not use any of those arrays, but the arguments are present in the programming interface for compatibility with SHMEM. Since these arrays are never used, the programmer can pass 0 or `NULL` as a pointer to those locations in a GPSHMEM specific code. It needs to be pointed out that the lack of the use of the workspace arrays does not affect the functionality of GPSHMEM compared to that of SHMEM.

3.7 Cray pointers

The standard Fortran 77 language does not offer any support for dynamic memory allocation or addressing memory locations. This limits the use of the Cray hardware to some extent (including use of the SHMEM library) and therefore the Fortran 77 implementation on Cray MPP systems contains an extension to the standard known as Cray pointers. It is worth mentioning that Cray pointers and pointers available in Fortran 90 are not the same thing, are not compatible and cannot be used interchangeably. In fact, the details of the representation of a Fortran 90 pointer are left up to the implementation. For that reason, the term *Cray pointer* is a name with a precise meaning and is used here to refer to the Cray's extension to Fortran 77.

A Cray pointer is a separate data type and is always associated with a data object. For example,

```fortran
REAL F00(10)
POINTER (PF00, F00)
```

declares a Cray pointer PF00 and associated *pointee* F00. The array F00 is not placed anywhere in the memory until the pointer PF00 is assigned a value. A Cray pointer can be defined and subsequently redefined by a function LOC. Each such definition of PF00 would cause F00 to refer to a new memory location. Let's see an extended example:
REAL FOO(10), BAR(10), BAZ(100)

POINTER (PFoo, FOO)

PFoo = LOC(BAR(1))

FOO(1) = 1.0

PFoo = LOC(BAZ(91))

FOO(10) = 100.0

First, the pointee array FOO is placed at the location of the array BAR. The assignment FOO(1) = 1.0 is equivalent to BAR(1) = 1.0. Next, FOO is overlaid with the last 10 elements of the array BAZ and the following assignment stores a value of 1.0 in its last element. The function LOC can be thought of as a dynamic version of the EQUIVALENCE statement available in Fortran 77.

Using the function LOC is not the only way to define a Cray pointer. The real usefulness of Cray pointers is demonstrated in conjunction with dynamic memory allocation, a feature that the standard language lacks. A programmer has been provided with the collective function SHPALLOC that allocates a block of memory from the shared heap and assigns its location to a Cray pointer given as one of the arguments. An object associated with a pointer created by SHPALLOC routine is considered to be symmetric and thus can be used in SHMEM calls. Below is a more complete, although rather trivial, example of using Cray pointers and dynamic memory allocation:

PARAMETER (N=10)

REAL FOO(N)

INTEGER ERROR, ABORT

POINTER (PFoo, FOO)

CALL SHPALLOC(PFoo, N, ERROR, ABORT)

C Use the memory

FOO(1) = 123.456
C    Done using the memory

    CALL SHPDEALLOC(POO, ERROR, ABORT)

END

Using SHPALLOC and SHDEALLOC is not the only way to use dynamic memory allocation
with Fortran 77 on Cray systems. More information on the subject can be found in the

The Fortran 2000 Draft Standard [42] from the J3 subcommittee of the International
Committee for Information Technology Standards (INCITS) has addressed Fortran and
C/C++ pointer interoperability issues. However, over ten years after the Fortran 90
standard we are only now getting reliable Fortran 90 compilers. Moreover, to date,
there is no widely ported open-source Fortran 90 compiler available to the computational
science community. Thus, the interoperability of pointers is not a portability requirement
of our GPSHMEM design.
4 The GPSHMEM API reference

GPSHMEM strives to provide equivalent functionality to SHMEM with an identical interface. During the design and later the implementation of GPSHMEM we strove to maintain as much compatibility with SHMEM as possible. However, there were some features that we have decided not to implement. As we present GPSHMEM, deviations from the original will be noted and discussed.

Many of the functions of SHMEM on the Cray T3D operate in terms of words. Such words are assumed to be 64-bit long. The number 32 in the name of a function usually indicates that the word size assumed by that function is 32 bits. GPSHMEM extends the T3D interface by a few functions operating on bytes. GPSHMEM is not guaranteed to run on systems where bytes are not 8-bit long.

GPSHMEM provides an application programming interface for C, Fortran 90/95 and Fortran 77. Unless indicated otherwise, the C functions can be used directly from both of the Fortran interfaces. Fortran 77 code must use the memory allocator to obtain remotely accessible data objects and the reference arrays to manipulate them (see section 5.4 for more details). The Fortran 90/95 code on Cray systems can use Cray pointers as arguments to the GPSHMEM calls (see section 3.7). By default, all function names in GPSHMEM have a prefix “gp”, that is, a function shmem_swap appears as gpshmem_swap. The original reason for this was to allow coexistence of SHMEM and GPSHMEM on the same platform. This prefix can be changed to any value at the time of GPSHMEM compilation, in particular, it can be eliminated and thus the compatibility with SHMEM can be achieved. For more information on the treatment of GPSHMEM function names
see chapter 9.

4.1 One-sided communication

4.1.1 Swap

long shmem_swap (long* target, long value, int pe);
long shmem_long_swap (long* target, long value, int pe);
int shmem_int_swap (int* target, int value, int pe);

Remarks: Functions shmem_long_swap and shmem_int_swap are available on the Cray T3E.

The swap functions atomically put the value at a location denoted by target on the remote PE pe. The previous value is returned. The shmem_swap, operates on values of type long, regardless of the size of that type. This is different from the original SHMEM on Cray T3D, where the assumed size of the values is 64 bits.

4.1.2 Get and put

void shmem_getmem (void* target, void* source, int nlong, int pe);
void shmem_get (void* target, void* source, int nlong, int pe);
void shmem_get32 (void* target, void* source, int nlong, int pe);

void shmem_putmem (void* target, void* source, int nlong, int pe);
void shmem_put (void* target, void* source,
int nlong, int pe);
void shmem_put32 (void* target, void* source,
    int nlong, int pe);

Remarks: Functions shmem_getmem and shmem_putmem are available on Cray T3E.

The get and put functions perform one-sided data transfer of a contiguous memory
block consisting of nlong data units from an address pointed to by source to the location
denoted by target. The get functions fetch data from the remote source on the PE pe
and store it locally. These functions return after the data has been placed in the local
buffer. The put functions store data from the local source to the remote target at PE
pe. Local completion of these functions does not imply remote completion. The data
unit size is assumed to be 64 bits for shmem_get and shmem_put, 32 bits for shmem_get32
and shmem_put32, and 8-bits for shmem_getmem and shmem_putmem.

4.1.3 Strided get and put

void shmem_iget (void* target, void* source, int target_inc,
    int source_inc, int nlong, int pe);
void shmem_iget32 (void* target, void* source, int target_inc,
    int source_inc, int nlong, int pe);

void shmem_iput (void* target, void* source, int target_inc,
    int source_inc, int nlong, int pe);
void shmem_iput32 (void* target, void* source, int target_inc,
    int source_inc, int nlong, int pe);

The iget and iput functions perform one-sided data transfers of strided memory
blocks from the source location beginning at source to the target location starting at
target. If size is the size of the data unit in bytes, the kth data unit is fetched from
address

\[ source + k * source \_inc \_size \]

and stored at

\[ target + k * target \_inc \_size \]

for \( k = 0, \ldots, nlong - 1 \). The \textit{size} is 64 bits for \texttt{shmem\_iget} and \texttt{shmem\_iput}, and 32 bits for \texttt{shmem\_iget32} and \texttt{shmem\_iput32}.

### 4.1.4 Gather/scatter

\[
\text{void shmem\_ixget (void* target, void* source,}
\]

\[
\text{long* source\_index, int nlong, int pe);}\]

\[
\text{void shmem\_ixget32 (void* target, void* source,}
\]

\[
\text{short* source\_index, int nlong, int pe);}\]

\[
\text{void shmem\_ixput (void* target, void* source,}
\]

\[
\text{long* target\_index, int nlong, int pe);}\]

\[
\text{void shmem\_ixput32 (void* target, void* source,}
\]

\[
\text{short* target\_index, int nlong, int pe);}\]

The \textit{ixget} functions gather irregularly distributed data on the remote PE \textit{pe} and store the data in a contiguous memory block beginning at a local address \textit{target}. The \( k \)th data unit (of size \textit{size}) is fetched from address

\[ source + source\_index[k] * size \]

The \textit{ixput} functions scatter a contiguous data block starting at the local address \textit{source}
and stores the consecutive data units at

\[ \text{target} + \text{target.index}[k] \times \text{size} \]

for \( k = 0, \ldots, n_{\text{long}} - 1 \). The \textit{size} is assumed to be 64 bit for \texttt{shmem.ixget} and \texttt{shmem.ixput}, and 32 bit \texttt{shmem.ixget32} and \texttt{shmem.ixput32}.

### 4.2 Collective routines

All collective routines in SHMEM involve a set of processing elements called “active set” (see section 3.5). The stride between PE identifiers must be a power of 2 in SHMEM. GPSHMEM extends the concept of active set by allowing arbitrary stride between PE identifiers. The GPSHMEM functions that implement this extension have names that end with \_st. The parameter \texttt{PE.stride} of those functions specifies the stride. The collective interface of SHMEM makes use of workspace arrays (see section 3.6), however GPSHMEM does not use these arrays. GPSHMEM-specific code can pass values of \texttt{NULL} for the corresponding arguments.

#### 4.2.1 Barrier

```c
void gpbarrier (void);
void shmem_barrier.all (void);
void shmem_barrier (int PE.start, int logPE.stride,
                   int PE.size, long* pSync);
void shmem_barrier.st (int PE.start, int PE.stride,
                       int PE.size, long* pSync);
```

\textit{Remarks}: \texttt{gpbarrier} (\texttt{barrier}) is an intrinsic function on the Cray systems and is not a member of the SHMEM interface. Its functionality is equivalent to that of
These functions implement the barrier operation, involving either all PEs (in case of gpbarrier and shmem_barrier_all) or a particular subset of all PEs (shmem_barrier and shmem_barrier_st).

4.2.2 Broadcast

```c
void shmem_broadcast (void* target, void* source,
                     int nlong, int PE.root, int PE.start, int logPE.stride,
                     int PE.size, long* pSync);

void shmem_broadcast32 (void* target, void* source,
                         int nlong, int PE.root, int PE.start, int logPE.stride,
                         int PE.size, long* pSync);

void shmem_broadcast_st (void* target, void* source,
                         int nlong, int PE.root, int PE.start, int PE.stride,
                         int PE.size, long* pSync);

void shmem_broadcast32_st (void* target, void* source,
                           int nlong, int PE.root, int PE.start, int PE.stride,
                           int PE.size, long* pSync);
```

*nlong* data units are distributed from address *source* at the PE *PE.root* and stored on all other PEs in the active set beginning at address *target*. The source PE is the member of the active set with the zero-based index of *root* within the active set (assuming ascending ordering of the PE identifiers). The *PE.root* does not send any data to itself.

4.2.3 Collection

```c
void shmem_collect (void* target, void* source,
                    int nwords, int PE.start, int logPE.stride,
                    int PE.size, long* pSync);
```
int PE_size, long* pSync);
void shmem_collect32 (void* target, void* source,
  int nwords, int PE_start, int logPE_stride,
  int PE_size, long* pSync);
void shmem_fcollect (void* target, void* source,
  int nwords, int PE_start, int logPE_stride,
  int PE_size, long* pSync);
void shmem_fcollect32 (void* target, void* source,
  int nwords, int PE_start, int logPE_stride,
  int PE_size, long* pSync);
void shmem_collect_st (void* target, void* source,
  int nwords, int PE_start, int PE_stride,
  int PE_size, long* pSync);
void shmem_collect32_st (void* target, void* source,
  int nwords, int PE_start, int PE_stride,
  int PE_size, long* pSync);
void shmem_fcollect_st (void* target, void* source,
  int nwords, int PE_start, int PE_stride,
  int PE_size, long* pSync);
void shmem_fcollect32_st (void* target, void* source,
  int nwords, int PE_start, int PE_stride,
  int PE_size, long* pSync);

Each PE in the active set has a block of data that is considered to be a contiguous segment of length \textit{nwords} words of a large distributed array. The \textit{collection} routines gather all those pieces and consolidate them into a single array which is stored at the address given by \textit{target} on all participating PEs. The ordering of the segments is implied
by the ordering of the PE identifiers in the active set, i.e. the first segment of the array will come from the PE $PE_{start}$, the second one, from $PE_{start} + 2^{\log PE_{stride}}$ or $PE_{start} + PE_{stride}$, and so on, depending on the type of function used. The functions from the $fcollect$ family expect all segment sizes to be equal at all participating processors, the other functions allow the segments to be of different sizes. The word size is assumed to be 64 bits, except in case of $shmem\_collect32$, $shmem\_fcollect32$, $shmem\_collect32\_st$, and $shmem\_fcollect32\_st$, where it is 32 bits.

### 4.2.4 Reduction

```c
void shmem\_TYPE\_OPER\_to\_all (TYPE* target, TYPE* source,
                               int nreduce, int PE\_start, int logPE\_stride, int PE\_size,
                               int* pWrk, long* pSync);

void shmem\_TYPE\_OPER\_to\_all\_st (TYPE* target, TYPE* source,
                                    int nreduce, int PE\_start, int PE\_stride, int PE\_size,
                                    TYPE* pWrk, long* pSync);
```

**Remarks:** The $TYPE$ in SHMEM is one of short, int, double or float. GPSHMEM added two more types: long or long long. The type long long is only available when GPSHMEM is compiled with a compiler supporting that data type. The $OPER$ is one of and, or, xor, sum, prod, min or max. The first three operations are not available for types float or double.

The *reduction* functions perform an operation denoted by $OPER$ on consecutive elements of the arrays $source$ of type $TYPE$. The source array must be of the same length $nreduce$ on each PE in the active set. The $k^{th}$ element of the $target$ array is the result of application of the operation $OPER$ to all of the $k^{th}$ elements of the $source$ arrays.
4.3 Miscellaneous functions

4.3.1 Fence

```c
void shmem_fence (void);
void shmem_quiet (void);
```

Remarks: These functions are not part of the original Cray T3D interface. Currently both of these functions in GPSHMEM are implemented as calls to ARMCI_AllFence.

The remote put operations are not guaranteed to take effect in the same order in which they are issued. The shmem_fence acts as a order guarantor enforcing a particular order of completion of the remote write operations. More precisely, for any pair of PEs, say spe and tpe, and any two put operations issued by spe accessing tpe, if spe called shmem_fence between the two put operations, then they will complete at tpe in the order in which they were issued. In other words, remote put operations affecting some remote PE issued before the call to shmem_fence will complete before any such operations issued after that call.

shmem_quiet blocks until all remote put operations issued from the calling PEs complete and become visible for the remote PEs.

4.3.2 Wait

```c
void shmem_wait (long* ivar, long value);
void shmem_wait_until (long* ivar, int cmp, long value);
void shmem_int_wait (int* ivar, int value);
void shmem_int_wait_until (int* ivar, int cond, int value);
void shmem_long_wait (long* ivar, long value);
void shmem_long_wait_until (long* ivar, int cond, long value);
```
Remarks: All of these functions, except \texttt{shmem.wait} are not included in the SHMEM on the Cray T3D.

The \textit{wait-until} functions block until the local value pointed to by \textit{ivar} and the value given as \textit{value} satisfy the condition specified by \textit{cond}. The possible values of \textit{cond} and the corresponding conditions are:

- \texttt{SHMEM_CMP_EQ: *ivar = value}
- \texttt{SHMEM_CMP_NE: *ivar \neq value}
- \texttt{SHMEM_CMP_GT: *ivar > value}
- \texttt{SHMEM_CMP_LE: *ivar \leq value}
- \texttt{SHMEM_CMP_LT: *ivar < value}
- \texttt{SHMEM_CMP_GE: *ivar \geq value}

The other functions assume the condition to be equality. This function implicitly assumes that a \textit{put} or \textit{swap} operation has been or will be issued at a remote PE.

4.3.3 Time

\begin{verbatim}
double shmem_time (void);
\end{verbatim}

Remarks: The function \texttt{shmem.time} is an extension of GPSHMEM. \texttt{shmem.time} returns time, in seconds, elapsed from some fixed point in the past.

4.3.4 Error

\begin{verbatim}
void shmem_error (char* msg);
\end{verbatim}

Remarks: The function \texttt{shmem.error} is an extension of GPSHMEM. \texttt{shmem.error} displays the message \textit{msg} and aborts the execution of the current program.
4.3.5 PE queries

\begin{verbatim}
int gpnumpes (void);
int gpmype (void);
\end{verbatim}

Remarks: The PE queries are not a part of the official SHMEM interface. Some SHMEM implementations provide equivalent functions \texttt{num.pes} and \texttt{my.me}.

The function \texttt{gpnumpes} returns the number of processing elements executing a given program. The function \texttt{gpmype} returns the identifier of the calling PE.

4.3.6 Cache control functions

The Cray T3D SHMEM interface included functions \texttt{shmem.set.cache.inv}, \texttt{shmem.set.cache.line.inv}, \texttt{shmem.clear.cache.inv}, \texttt{shmem.udcflush} and \texttt{shmem.udcflush.line} for cache control. The remote memory accesses on the Cray T3E guaranteed cache coherency and thus those functions became obsolete. GPSHMEM does not implement any cache coherency functions.

4.3.7 \texttt{shmem.ptr}

The \texttt{shmem.ptr} function on the Cray T3D and T3E returned a pointer to a remote memory location. Such a pointer could later be used in the same manner as a local pointer. GPSHMEM does not implement this function and so all remote memory accesses must be done via explicit GPSHMEM calls.

4.4 Memory allocation

There are separate C and Fortran 77 interfaces for memory allocation. The C functions and the \texttt{gpshpalloc} functions can also be used with Cray pointers in Fortran 90/95. The C functions are:
The collective function `gpshmalloc` allocates at least the number of bytes given by `nbytes` on all PEs. The memory is aligned to the most recently specified boundary, or to the default boundary of 8 (actually, `sizeof (double)`). The memory can be freed by passing the pointer returned from `gpshmalloc` to `gpshfree`. The memory alignment boundary can be set by `gpptralign`, which returns the previous boundary.

The Fortran 77 memory management interface consists of:

- **FUNCTION** `GPSSHMALLOC (TYPE.ID, LENGTH)`
- **FUNCTION** `GPSSHINDEX (TYPE.ID, HANDLE)`
- **FUNCTION** `GPSSHMALLOCI (TYPE.ID, LENGTH, INDEX)`
- **SUBROUTINE** `GPSSHFREE.HANDLE (HANDLE)`
- **INTEGER** `GPSSHMALLOC, GPSSHINDEX, GPSSHMALLOCI`
- **INTEGER** `HANDLE, INDEX, TYPE.ID, LENGTH`

For details on using the Fortran 77 memory allocator, see section 5.4. The parameter `TYPE.ID` specifies the Fortran 77 data type associated with a given memory block. Currently, the supported types and their corresponding identifiers are:

- **INTEGER** `TYPE.ID = 0`
- **INTEGER*4** `TYPE.ID = 1`
- **INTEGER*8** `TYPE.ID = 2`
- **REAL** `TYPE.ID = 3`
- **DOUBLE PRECISION** `TYPE.ID = 4`
• COMPLEX \textit{TYPE.ID} = 5

• DOUBLE COMPLEX \textit{TYPE.ID} = 6

Functions specifically addressed for Fortran 90/95 are:

\textbf{SUBROUTINE GPSHPALLOC} (\textit{PTR, LENGTH, ERRCODE, ABORT})
\textbf{SUBROUTINE GPSHPALLOC32} (\textit{PTR, LENGTH, ERRCODE, ABORT})
\textbf{SUBROUTINE GPSHPALLOC64} (\textit{PTR, LENGTH, ERRCODE, ABORT})

\textbf{INTEGER} \textit{LENGTH, ERRCODE, ABORT}
\textbf{POINTER} \textit{PTR}

\textit{Remarks:} The arguments \textit{ERRCODE} and \textit{ABORT} are ignored by GPSHMEM.

The \textit{gpshalloc} functions allocate memory block of length \textit{LENGTH} words and associate pointer \textit{PTR} with it. The size of the word is assumed to be 64 bits, except in case of \textbf{GPSHPALLOC32}, which assumes the size to be 32 bits.

4.5 Initialization and finalization

\textbf{FUNCTION GPSHMEM.INIT ()}
\textbf{SUBROUTINE GPSHMEM.FINALIZE ()}
\textbf{INTEGER GPSHMEM.INIT}

\textit{Remarks:} These functions are GPSHMEM’s extensions.

These are initialization and finalization functions. No call to any GPSHMEM function can precede a call to the initialization function. Analogously, no call to any GPSHMEM function can follow a call to the finalization function. These functions also initialize and finalize the underlying message passing library.
5 Implementation

This chapter describes the structure and design of GPSHMEM. GPSHMEM builds its functionality on top of two major components: one of them is the ARMCI, the other is a message passing library. The structural relationship between all GPSHMEM’s components is presented. Implementation of different aspects of the GPSHMEM’s functionality is discussed. Section 5.4 presents the GPSHMEM’s memory allocator which is one of the central modules responsible for implementation of symmetric data objects in distributed environments.

5.1 ARMCI

ARMCI is one of the fundamental tools on which GPSHMEM is built. It implements a similar way to perform remote memory access operations to the way SHMEM did. Neither ARMCI nor SHMEM offer any coherency control. SHMEM is just a library allowing the program to read or write remote memory as if it was its own. The only difference between the SHMEM programming model and the true shared memory model was that SHMEM architecture was explicitly NUMA, whereas shared memory appears to be uniform. ARMCI is very similar to SHMEM in that respect. No coherency models are implemented, and the programmer is expected to maintain the program's consistency by using explicit synchronization mechanisms.

The ARMCI programming interface provides a relatively small but powerful interface. The ARMCI functionality can be divided into several groups:
1. Multi-level strided memory copy operations (ARMCI_GetS, ARMCI_PutS). These functions allow the transfer of sections of multidimensional arrays in one function call. The simple, `memcpy`-like memory copy functions, ARMCI_Get and ARMCI_Put are simple cases of ARMCI_GetS and ARMCI_PutS.

2. The generalized I/O vector operations (ARMCI_GetV, ARMCI_PutV). These functions follow the Unix `readv/writev` interface which allows the transfer of multiple scattered regions at once.

3. Accumulate operations (ARMCI_AccV, ARMCI_AccS). These functions allow the atomic update of remote memory locations according to the formula \( dest = dest + scale \times src \). These are not used by GPSHMEM.

4. Atomic read-modify-write (ARMCI_Rmw). This function performs an atomic swap or an atomic addition on a remote integer variable.

5. Completion operations (ARMCI_Fence, ARMCI_FenceAll). Block until outstanding memory access operations complete.

6. Mutex operations. These functions are not used by GPSHMEM.

7. Memory operations (ARMCI_Malloc, ARMCI_Free). All memory blocks that are to be used as remotely accessible locations in ARMCI calls must have been allocated with these routines. This limitation is inherited by GPSHMEM and thus the remotely accessible (or symmetric) data objects in GPSHMEM are those that were eventually allocated by ARMCI_Malloc. GPSHMEM provides its own function, `gpshmalloc` that invokes ARMCI_Malloc and registers the memory blocks with GPSHMEM. ARMCI_Malloc is a collective routine and it returns an array of pointers to the memory blocks allocated on each process.

8. Miscellaneous routines, such as initialization, cleanup, and abort functions.
5.2 Message passing libraries

The collective interface of GPSHMEM relies on the functionality provided by a message passing library. As opposed to the ARMCI, which is an integral part of GPSHMEM, the message passing component is intended to be replaceable. To maximize the variety of potential message passing libraries, GPSHMEM does not rely on any specific functionality provided by any particular interface. The only functions that are used by GPSHMEM to implement the collective interface are send_message, receive_message, and barrier.

GPSHMEM implements its own message passing functions: gp_send_to_one, gp_recv_from_one, gp_send_to_many and gp_recv_from_many that are nothing more than wrappers around the particular message passing function. Thus, adopting GPSHMEM to use an arbitrary message passing library would include providing the implementations of these functions, the implementation of a barrier, and a proper startup and shutdown code in GPSHMEM_INIT and GPSHMEM_FINALIZE. If a Fortran interface is desired for the new message passing library, appropriate initialization and cleanup must be provided via Fortran routines.

So far, the GPSHMEM has been built and thoroughly tested with two message passing libraries: MPI (i.e., MPICH [52] and other implementations) and TCGMSG [54]. Initial support has been added for PVM [48], although it has never been tested. Future plans include integration with MP_Lite [89] and other libraries, according to the demands.

5.3 GPSHMEM

GPSHMEM is a higher-level library than ARMCI or a message passing library. The implementation of GPSHMEM contains several logically separated blocks, as depicted in figure 5.1. The top two boxes represent the two major GPSHMEM interfaces: one-sided and collective. Each of these components is a collection of functions belonging to
the GPSHMEM application programming interface. The GPSHMEM functions operate on 32- or 64-bit words, in case of the one-sided interface, or on particular data types, i.e. int, double, etc., in case of the collective routines. Internally, all data transferred by GPSHMEM is treated as sequences of bytes. As a consequence, GPSHMEM will only operate properly in homogenous environments, where corresponding types have identical representations. This limitation is almost never truly restrictive in practice, but it significantly simplifies the GPSHMEM implementation. In the beginning, the first release of GPSHMEM had separate functions operating on all supported data types. The code was bloated and many functions were duplicated, with the only difference being in the actual data types used. This soon proved to be a maintenance headache. Each time an error was found in one routine, it was necessary to also fix it in all of the "clones" of that routine. Each time the code of such function was changed, such changes needed to be manually propagated to all other instances. We have decided that loss of support for heterogeneity was a small price to pay for an increase in code clarity and ease of maintenance. As an effect, most functions from the application programming interface are implemented as simple calls to the byte-driven functions from GPSHMEM's internals.
The **ARMCI adaptation layer** is a thin interface between the one-sided functions of the GPSHMEM and the ARMCI. For reasons explained in section 5.4, some feedback from the memory allocator must be obtained before the control can be passed to ARMCI. Since all of the one-sided functionality of GPSHMEM has its counterparts in ARMCI, all one-sided functions of GPSHMEM simply translate their arguments into the format expected by ARMCI.

The **collective interface implementation** consists of routines that perform all of the collective routines in terms of simple `send_message` and `receive_message` operations. As mentioned above, message passing is not performed directly by the message passing library, but by a few isolated functions embraced under the `primitive message functions` label in figure 5.1. The collective operations include `barrier`, `broadcast`, `collection` and `reduction`. Since all of these routines can involve active sets of arbitrary size, a communication scheme using only the four primitive functions needed to be developed. In GPSHMEM we use a binary tree. A support for a $k$-ary tree was considered, for arbitrary $k \geq 2$, but it was dropped since there appears to be no direct advantage in increasing the number of direct descendants of a tree node. In fact, a given node would have to synchronize not only with its parent and the two children, but with the parent and $k$ children. The use of $k$ greater than 2 could potentially reduce the depth of the tree, however we did not believe that the benefits of having shorter trees would outweigh the latencies introduced by synchronization with an increased number of neighbors.

**Broadcast** is implemented by a single top-down pass of the given message to the lower nodes in the tree. **Barrier** acts similarly, except that it waits for an acknowledgment of the dummy message broadcasted in the first stage. In other words, each node sends a message to all of its children and then waits for the children to return the message. This message is then passed to the parent. **Reduction** works in the opposite direction—first, partially computed results are received from the children, then the given node performs the reduction operation on its own data and the data obtained from its children, and
finally the outcome is handed to the parent node. After the calculations have been performed by the root node, the final result is known. It now has to be broadcasted down to all other nodes. The communication in reduction is thus bottom-up followed by a top-down broadcast. Since the code is modular, more optimal solutions could easily be developed once the functionality base has been finalized.

The collection routines consolidate an array whose pieces are distributed among multiple processors. The collection functions come in two flavors, one expects the sizes of the array segments to be identical, the other does not. In the first case, the communication very much resembles that used in the reduction. The array portions are passed up the tree and each node combines the results from its children into a single piece of data. After the root node has performed such consolidation, the entire array is obtained. It is then propagated down to all other nodes in the tree. The case with arbitrary segment sizes is more difficult to handle. To be able to put the various parts of the array obtained from the child nodes in proper places, the lengths of all of the array segments must be known. Therefore, in the first phase the sizes of all array portions are exchanged between processors in a way similar to the simple case of reduction. Then a similar routine is performed again, but this time the actual data is transferred.

5.4 Memory allocator

The memory allocator plays two major roles in GPSHMEM. Its first function is to provide memory address translation and the second is to enable memory allocation from a Fortran 77 code.

5.4.1 Memory address translation

The SHMEM programming model relies on the concept of symmetric data objects. To recap, symmetric data objects are objects whose addresses remain in a known rela-
tionship across processors. This “known relationship” is usually an equality, and so a symmetric data object appears to occupy the same memory location on all processors. The ARMCIMalloc (briefly described in section 5.1) is not guaranteed to allocate its remotely accessible memory at the same address on each node. That is why it returns an array of pointers instead of just one. To make the memory block allocated by ARMCI appear symmetric, we employed a simple translation trick. The gpsmalloc function, responsible for the memory allocation in GPSHMEM, invokes ARMCMalloc and returns the pointer that is local to the calling process. This way, each process receives the address at which the allocated memory block begins in its own memory. Since each process will assume that the memory block is located at the same place in all of the remote memories, it will pass its local pointer as the remote address in the GPSHMEM calls. For example,

```c
src_ptr = malloc (size);
remote_ptr = gpsmalloc (size);
/* ... */
shmem-put (remote_ptr, src_ptr, size/8, remote_pe);
```

Of course the remote_ptr will most likely be valid only at the calling processor. To work around this problem, before calling ARMCI_Put, GPSHMEM looks the given pointer up in the array previously returned by ARMCMalloc, finds the corresponding remote pointer and passes it to the ARMCI_Put. This technique is sometimes referred to as proxy pointer and is frequently used in the database community [40]. To make this scheme work, GPSHMEM’s memory allocator needs to keep some maintenance data. With each memory block allocated via gpsmalloc, there is associated a data structure, called memory block descriptor. Such descriptor holds the size of the block and the array of pointers returned by ARMCMalloc as well some other information. For each call to a GPSHMEM function, there is a local pointer “pretending” to be the remote address.
The local address is translated into the remote address in the following way:

1. A block into which a given local pointer on the local PE points is found. It is important to notice that the local pointer does not need to point to the beginning of the local block. It can point to anywhere inside of such a block. If such a block cannot be found, an error occurs. This situation indicates a serious programming error and no recovery is attempted.

2. An offset from the beginning of the found memory block is calculated.

3. The pointer to the beginning of the memory block on the remote PE is retrieved from the array returned by ARMCIMalloc.

4. The offset calculated in step 2 is added to the pointer looked up in step 3

In reality, the memory block descriptor holds more information. One major thing to keep in mind is the memory alignment. The memory allocation in Fortran 77 described below relies on proper data alignment. The memory allocator makes sure that enough memory is allocated to contain a contiguous area beginning from an address aligned to some predefined boundary. The memory block descriptor holds additional information about added offsets to properly calculate the translated memory addresses. The alignment boundary is, by default, set to sizeof (double), which is 8 on most systems. It can be changed to any value by using gpptralig function, but the results are only defined for alignments that are powers of 2 and that are not less than the default value. Use of this function is only encouraged in special situations and requires caution. The default value should be sufficient in most cases.

5.4.2 Fortran 77 memory allocation

Fortran 77 does not have a pointer type, nor does it provide any support for dynamic memory allocation. All objects in pure Fortran 77 code are allocated statically. The
GPSHMEM's memory allocator is a library extension to Fortran 77 that provides the `malloc` functionality wrapped in the Fortran 77 interface. It is implemented on top of the GPSHMEM's memory management for C, and so the memory blocks allocated by it are remotely accessible. Since Fortran 77 code cannot use pointers, some other data type must be used to identify memory blocks. The main idea here is the same as the one used in the Memory Allocator created as a part of Global Arrays [93].

The data blocks were identified by integer *handles*. The allocation function would return a *handle* that can later be used to access the memory block. An integer value, however, is not enough to access that memory block. The only way in which data can be stored in memory, directly from the Fortran 77 code, is via variables or arrays. Since variables offer very little flexibility, the solution had to use arrays somehow. An array element access can be thought of as a way to address some areas of the memory via variable displacement from a fixed memory location. That fixed memory location is, of course, the address of the first element of the array. An index of 1 gives us that the first element itself, index of 0 would address the element preceding the first one, etc.\(^1\) It is not hard to imagine that for every location in the address space, and for any array \(A\), there exists an index \(I\) such that \(A(I)\) falls into that location. The only issue with this solution is that all objects in Fortran have types, and so all elements accessed via references to some array \(A\) are assumed to have the data type of \(A\). To allow the programmer to use as many data types as possible, separate arrays need to be declared for every data type. Unfortunately, Fortran 77 has (theoretically) infinitely many data types. This is essentially caused by character strings that can have arbitrary lengths. Two strings of two different lengths are treated as different data types, at least as far as the programmer's convenience is involved. The GPSHMEM's memory allocator contains a small Fortran 77 module that declares a set of arrays, called *reference arrays*. Such arrays are declared for each data type, except CHARACTER. The memory allocation

\(^1\)Unless indicated otherwise, arrays in Fortran 77 are indexed from 1.
process in Fortran 77 follows this order:

1. Call GPSHMALLOC with two arguments: the *type identifier* and the desired number of elements of given type in the block. The type identifier is necessary for GPSHMEM to calculate the amount of memory in bytes, as Fortran does not have a counterpart of the `sizeof` operator from C or C++. The GPSHMALLOC will return a *handle* that can be used to identify the block later on. For example:

   \[
   \text{HANDLE} = \text{GPSHMALLOC} \left( \text{real\_type\_id}, \ 10 \right)
   \]

   The actual type identifier is only used to calculate the final block size, so an identifier of any type (of appropriate length) can be used.

2. Call GPSHINDEX to obtain the *index* into the proper reference array. GPSHINDEX takes two arguments: type identifier and the handle returned by GPSHMALLOC. Continuing the example, have

   \[
   \text{INDEX} = \text{GPSHINDEX} \left( \text{real\_type\_id}, \ \text{HANDLE} \right)
   \]

   In this case, the type identifier helps choose the right reference array for the index calculation, so the type identifier provided here must match the reference array used to access the block in the subsequent code.

3. Use the memory via accesses to the proper reference arrays

   \[
   \begin{align*}
   \text{DO} \ 10 & \ I = 0, 9 \\
   \text{GPS\_REAL} \left( \text{INDEX} + I \right) & = I \\
   10 \ & \text{CONTINUE}
   \end{align*}
   \]

4. Deallocate the memory block

   \[
   \text{GPSHFREE\_HANDLE} \left( \text{HANDLE} \right)
   \]
The whole memory allocation scheme employed in the Fortran 77 allocator does not seem to encourage good programming practices. Accessing memory locations well beyond the statically allocated storage for a given array is definitely not something that should be encouraged. Unfortunately, Fortran 77 does not provide any other means to dynamically allocate memory blocks. It is acknowledged that not all implementations have to support such tricks, but most of the known ones do. In such situation, the reduction of portability proves to be a good price to pay for adding an entirely new functionality.
6 Performance

6.1 The Splash-2 suite

As our test suite we have chosen the enhanced edition of the Stanford Parallel Applications for Shared Memory (Splash-2) [121]. The Splash-2 test suite consists of two parts: kernels and applications. The available kernels are Fast Fourier Transform, LU decomposition, Sparse Cholesky factorization and radix sort. The set of applications includes Barnes Hut n-body problem solver, ocean simulation, ray tracer and others. We decided to focus our attention on the kernels, since they contain the essence of the scientific computations and the results of those benchmarks can serve as an accurate prognostic for the GPSHMEM's performance in real applications. The calculations performed by these kernels are often the basis of the scientific applications, and the kernels are a well-known metrics of the computational throughput.

For the purpose of benchmarking GPSHMEM, we have chosen 3 kernels: FFT, LU decomposition and radix sort. These 3 provide a representation of the computational designs used in scientific applications.

The Splash-2 kernels had been already ported to SHMEM by Hongzhang Shan as a part of his research efforts at Princeton University. Mr. Shan has been kind enough to share the C source codes with us. Since the destination platform for these codes were the SGI Origin 2000, the Splash-2 codes have been adjusted to the requirements of the SHMEM implementation available on that platform. Since the SHMEM interface is not standardized, different implementations of it may differ slightly. GPSHMEM is not
an exception to that rule, but the necessary modifications were minor and we obtained working code in a very short time.

6.2 IBM-SP

The primary platform for our measurements was an IBM SP system, hosted by the National Energy Research Scientific Computing Center (NERSC). NERSC, a part of the Lawrence Berkeley National Laboratory of the U.S. Department of Energy provides high-performance computational resources to research institutions in the United States.

The IBM RS/6000 SP system is a collection of nodes connected with a high-bandwidth, low-latency network. Each node consists of 16 IBM POWER3 processors and has from 16 to 64 gigabytes of memory shared among the processors. Each node operates under its own instance of the AIX operating system.

6.3 The benchmarks

The job submission system on IBM SP (LoadLeveler) allows the user to request a particular number of nodes and specify the number of tasks per node. Due to the hardware configuration of the SP system, the number of tasks per node cannot exceed 16. To obtain a broad spectrum of results, we have performed a variety of time measurements. For one of the tests, we have assumed some specific configuration of the number of nodes and tasks per node and changed the problem size. The most important tests, however, were not related to the dependence of the execution time of the problem size; which were the scalability tests. The reason parallel computers are used is that increasing the number of computational resources should result in a decrease of the total execution time. Of course, a communication library cannot be expected to magically make a program run in half the time when the number of processors doubles, but a badly designed or a poorly implemented one can easily offset the benefits of running a program in parallel.
The Splash-2 codes already contained time-measurement components. After a given program terminated, it would print the measured times. We have adjusted those components to calculate and display the communication times. The times reported by each of the programs were later used to calculate average execution and communication times for each program with a fixed problem size and number of nodes and tasks per node. The averages were calculated using at least three different results. On systems like IBP-SP, where each node executes its own instance of the operating system, a large variance of the timing results is a common problem. When the results varied significantly, we submitted additional jobs to increase the size of the set of measurements. To better understand the nature of the communication in each of the problems, we have also calculated the ratio between the computation time and the communication time. Summarizing, for each test performed we have three different measurements:

1. total execution time, with and without initialization,
2. total communication time, including barriers, with and without initialization,
3. the ratio of the computation time to the communication time, with and without initialization.

We refer to a specified configuration of nodes and tasks per node as process grid.

To further increase the number of different measurements, GPSHMEM was compiled with two different message passing libraries: MPI and TCGMSG.

6.3.1 Scalability tests

For all of the problems, we have performed two different groups of the scalability tests:

1. Fixed number of nodes, varying number of tasks per node:
   (a) nodes = 1,  tasks_per_node = 2, 4, 8, 16,
2. Fixed number of tasks per node, varying number of nodes:

(a) $\text{nodes} = 1, 2, 4, 8, 16, \quad \text{tasks}\_\text{per}\_\text{node} = 1,$

(b) $\text{nodes} = 1, 2, 4, 8, \quad \text{tasks}\_\text{per}\_\text{node} = 8,$

All of these tests assumed some fixed problem size, dependent on the particular problem. The detailed descriptions will follow.

### 6.3.2 Problem size timings

These tests were designed to analyze the changes in the execution times as the problem size increases. The theoretical complexities of the algorithms used to solve the problems were known and, ideally, the observed increases in the execution time would reflect those complexities. For each problem, we have chosen three process grids ($\text{nodes} \times \text{tasks}\_\text{per}\_\text{node}$):

1. $4 \times 8,$

2. $4 \times 16,$ and

3. $8 \times 8.$

The ranges of the problem sizes depended on the particular problem. The details will be presented below.

### 6.4 FFT

The problem size for the scalability tests for FFT was set to $2^{24}$ (4M), except in case 2a from section 6.3.1, where, due to memory limitations, it was assumed to be $2^{20}$ (1M).

The FFT kernel timings were quite difficult to analyze and interpret. The test case with a single node (6.3.1, 1a) demonstrates a nice scaling behavior. The execution time
decreases as tasks_per_node increases (figure 7.1), the communication time slightly increases (figure 7.2), and obviously, the computation-to-communication ratio decreases (figure 7.3). In this situation all communication took place between processors on the same node. The test case with 4 nodes (1b, 6.3.1) introduces some inter-node communication and things begin to look different. First of all, substantially more time is spent on communication (figure 7.5), and the computation-to-communication ratios are now below 1 most of the time (figure 7.6). An interesting phenomenon occurs as the number of tasks per node grows from 8 to 16: the communication time drops significantly (figure 7.5). Most of the communication in the FFT kernel occurs, besides barriers, in calls to shmem_putmem. Inspection of the source code shows that the size of the data block transferred in that call is

\[ \frac{2 \text{sizeof(double)} \cdot N}{P^2} \]

or

\[ \frac{2^4 \cdot N}{P^2} \]

knowing that sizeof (double) is 8, where \( N \) is the problem size and \( P \) is the number of processes. With 4 nodes, the change of tasks_per_node from 8 to 16 increases \( P \) from 32 to 64. This causes the message size to decrease from

\[ \frac{2^4 \cdot 2^{24}}{2^{10}} = 2^{18} = 256k \]

to

\[ \frac{2^4 \cdot 2^{24}}{2^{12}} = 2^{16} = 64k \]

The abrupt change in the communication time, as the message size changes between 64k and 256k is consistent, as shown in figures 7.11 and 7.14. The low communication time for 1 \( \times \) 8 in figure 7.11 is due to the communication occurring entirely within a single node.
Because of memory limitations, the problem sizes in the tests showing the dependence of execution time on problem size were relatively small. Because of that, the total computation times were very short and could be significantly influenced by otherwise negligible factors such as rounding errors in time calculations, varying latency of the time measurement functions and the like.

In summary, when the execution times were large enough to eliminate the risk of having minor operating system fluctuations affect the time measurements, the scaling behavior meets our expectations. We consistently observed the drastic increase of inter-node communication times as the message sizes went from 64k to 256k. The exact cause of that behavior is not known, but it could be caused by a possible message fragmentation occurring at the SP switch.

6.5 LU decomposition

The LU factorization very clearly exhibits the desired scaling behavior. The execution time nearly halves as the number of processes doubles (figures 7.22, 7.25, 7.28 and 7.31). The computation-to-communication ratio behaves similarly (figures 7.24, 7.27, 7.30, 7.33), as the communication time remains nearly unchanged (figures 7.23, 7.26, 7.29, 7.32). The expected exception is the low communication time in case of a single node and a single task per node (figure 7.29). The observed time complexity closely follows the theoretical complexity of $O(n^3)$ (figures 7.34, 7.37, 7.40).

This benchmark clearly demonstrates that in computationally intensive applications, the scaling properties are preserved and the GPSHMEM library allows for an efficient use of an increase in computational resources. There are no noticeable overheads introduced that would interfere with either the scalability potential of this problem or its temporal complexity.
6.6 Radix sort

Similar to the LU decomposition, the radix sort kernel shows a clear dependency of the execution time on the number of processes executing it (figures 7.43, 7.46, 7.49 and 7.52). The cost of initialization is noticeably high, much higher than in any of the two other kernels. The computation-to-communication graphs (figures 7.45, 7.48, 7.51, 7.54) prove that the cost is mostly of a computational nature. An interesting fact is that, as in the case of FFT and LU decomposition, the differences in performance between MPI and TCGMSG\(^1\) were negligible, here the TCGMSG performs noticeably slower than MPI. This is clearly visible in figures 7.47, 7.53, and figures 7.56, 7.59, and especially in figure 7.62, where the performance penalty associated with TCGMSG reaches the factor of 10 or more. The exact cause of this difference is not known, however these results were persistent.

6.7 Remarks

6.7.1 Benchmark failures

During the testing process we observed some unexpected failures of the GPSHMEM-enabled Splash-2 kernels. These failures were sometimes caused by programs exceeding the time allocated for their execution. In a few cases it was our fault, but in several cases, the problem seemed to be caused by some runaway condition somewhere in the code. Other failures were caused by invalid operations performed by the programs that resulted in a premature termination. An important fact regarding those failures is that they only occurred when we used GPSHMEM linked against MPI. Since the ARMCI uses LAPI for one-sided communication, we suspect that there was some interference between LAPI and MPI on that system. Initially there were a few failures with TCGMSG, but,

\(^1\)as the underlying message passing library
in subsequent reruns of all of the benchmarks we have not observed any problems. The fact that the vast majority of these failures occurred with MPI and almost none with TCGMSG, and the fact that the only few failures associated with TCGMSG occurred in the initial stage of the testing efforts, leads us to believe that the cause of these problems is not associated with GPSHMEM itself, but lies deeper within the layers of communication libraries.

6.7.2 G4 cluster

The Scalable Computing Laboratory at the Ames Laboratory has several computing facilities, although none of them are of a computing power comparable to that of the IBM SP. These are mostly clusters made of low-cost Unix machines. Since GPSHMEM was targeted for such systems, we have run the Splash-2 kernel benchmarks on one of them. The system of our choice was the G4 cluster, consisting of 16 single processor nodes, each running at 400Mhz and 16 dual processor nodes running at 500Mhz. Each node gives 512MB of memory per processor. The interconnects are Fast Ethernet and Myrinet. The process counts were 1, 2, 4, 8, 16 and 32; the problem sizes were identical as on the IBM SP, except that the upper size for the radix sort was $2^{26}$ due to its memory limitations. The memory configuration also required the 2 processor runs to be performed on 2 nodes. The results obtained from these tests were similar to those on the SP system at NERSC. We have observed erratic behavior in the TCP/IP over Myrinet, but those variations may have been caused by the OS activity associated with executing other applications.
7 Performance graphs

This chapter contains the graphs demonstrating the results of the performance tests discussed in chapter 6.

7.1 FFT

7.1.1 Varying \texttt{tasks\_per\_node}

![Graph of FFT: Total execution time, nodes = 1, varying \texttt{tasks\_per\_node}, problem size = 2^{24}]

Figure 7.1 FFT: Total execution time, nodes = 1, varying \texttt{tasks\_per\_node}, problem size = 2^{24}
Figure 7.2 FFT: Communication time for $\text{nodes} = 1$, varying $\text{tasks per node}$, problem size $= 2^{24}$

Figure 7.3 FFT: Computation to communication ratio for $\text{node} = 1$, varying $\text{tasks per node}$, problem size $= 2^{24}$
Figure 7.4 FFT: Total execution time for nodes = 4, varying tasks_per_node, problem size = $2^{24}$

Figure 7.5 FFT: Communication time for nodes = 4, varying tasks_per_node, problem size = $2^{24}$
Figure 7.6  FFT: Computation to communication ratio for nodes = 4, varying tasks_per_node, problem size = $2^d$
7.1.2 Varying nodes

Figure 7.7 FFT: Total execution time for varying nodes, \( tasks\_per\_node = 1 \), problem size = \( 2^{20} \)
Figure 7.8 FFT: Communication time for varying nodes, $tasks\_per\_node = 1$, problem size $= 2^{20}$

Figure 7.9 FFT: Computation to communication ratio for varying nodes, $tasks\_per\_node = 1$, problem size $= 2^{20}$
Figure 7.10 FFT: Total execution time for varying nodes, tasks_per_node = 8, problem size = $2^{24}$

Figure 7.11 FFT: Communication time for varying nodes, tasks_per_node = 8, problem size = $2^{24}$
Figure 7.12 FFT: Computation to communication ratio for varying nodes, \( \text{tasks} \_\text{per} \_\text{node} = 8 \), problem size = \( 2^{24} \)
7.1.3 Varying problem size

Figure 7.13 FFT: Total execution time for process grid $4 \times 8$, varying problem size.
Figure 7.14  FFT: Communication time for process grid $4 \times 8$, varying problem size.

Figure 7.15  FFT: Computation to communication ratio for process grid $4 \times 8$, varying problem size.
Figure 7.16 FFT: Total execution time for process grid 4 x 16, varying problem size.

Figure 7.17 FFT: Communication time for process grid 4 x 16, varying problem size.
Figure 7.18  FFT: Computation to communication ratio for process grid 4 × 16, varying problem size.

Figure 7.19  FFT: Total execution time for process grid 8 × 8, varying problem size.
Figure 7.20 FFT: Communication time for process grid $8 \times 8$, varying problem size.

Figure 7.21 FFT: Computation to communication ratio for process grid $8 \times 8$, varying problem size.
7.2 LU decomposition

7.2.1 Varying tasks_per_node

Figure 7.22 LU: Total execution time, nodes = 1, varying tasks_per_node, problem size = $2^{24}$
Figure 7.23 LU: Communication time for $nodes = 1$, varying $tasks\_per\_node$, problem size $= 2^{24}$

Figure 7.24 LU: Computation to communication ratio for $node = 1$, varying $tasks\_per\_node$, problem size $= 2^{24}$
Figure 7.25 LU: Total execution time for nodes = 4, varying tasks_per_node, problem size = $2^{24}$

Figure 7.26 LU: Communication time for nodes = 4, varying tasks_per_node, problem size = $2^{24}$
Figure 7.27  LU: Computation to communication ratio for nodes = 4, varying tasks_per_node, problem size = $2^{24}$
7.2.2 Varying nodes

![Graph showing total execution time for varying nodes.](image)

Figure 7.28 LU: Total execution time for varying nodes, \( \text{tasks.per.node} = 1 \), problem size = \( 2^{20} \).
Figure 7.29 LU: Communication time for varying nodes, \( tasks\_per\_node = 1 \), problem size = \( 2^{20} \)

Figure 7.30 LU: Computation to communication ratio for varying nodes, \( tasks\_per\_node = 1 \), problem size = \( 2^{20} \)
Figure 7.31 LU: Total execution time for varying nodes, tasks\_per\_node = 8, problem size = $2^{24}$

Figure 7.32 LU: Communication time for varying nodes, tasks\_per\_node = 8, problem size = $2^{24}$
Figure 7.33  LU: Computation to communication ratio for varying nodes, $tasks_{\text{per\_node}} = 8$, problem size $= 2^{24}$
7.2.3 Varying problem size

![Graph showing execution time for process grid 4x8 with varying problem size.]

Figure 7.34  LU: Total execution time for process grid 4×8, varying problem size.
Figure 7.35  LU: Communication time for process grid $4 \times 8$, varying problem size.

Figure 7.36  LU: Computation to communication ratio for process grid $4 \times 8$, varying problem size.
Figure 7.37  LU: Total execution time for process grid 4 x 16, varying problem size.

Figure 7.38  LU: Communication time for process grid 4 x 16, varying problem size.
Figure 7.39  LU: Computation to communication ratio for process grid 4 × 16, varying problem size.

Figure 7.40  LU: Total execution time for process grid 8 × 8, varying problem size.
Figure 7.41  LU: Communication time for process grid 8×8, varying problem size.

Figure 7.42  LU: Computation to communication ratio for process grid 8×8, varying problem size.
7.3 Radix Sort

7.3.1 Varying tasks_per_node

![Graph showing execution time for different process grids and initialization options]

Figure 7.43 Radix sort: Total execution time, nodes = 1, varying tasks_per_node, problem size = $2^{24}$
Figure 7.44  Radix sort: Communication time for nodes = 1, varying tasks\_per\_node, problem size = $2^{24}$

Figure 7.45  Radix sort: Computation to communication ratio for node = 1, varying tasks\_per\_node, problem size = $2^{24}$
Figure 7.46  Radix sort: Total execution time for nodes = 4, varying tasks_per_node, problem size = $2^{24}$

Figure 7.47  Radix sort: Communication time for nodes = 4, varying tasks_per_node, problem size = $2^{24}$
Figure 7.48 Radix sort: Computation to communication ratio for nodes = 4, varying tasks_per_node, problem size = $2^{24}$
7.3.2 Varying nodes

Figure 7.49 Radix sort: Total execution time for varying nodes, tasks_per_node = 1, problem size = $2^{20}$
Figure 7.50 Radix sort: Communication time for varying nodes, tasks_per_node = 1, problem size = $2^{20}$

Figure 7.51 Radix sort: Computation to communication ratio for varying nodes, tasks_per_node = 1, problem size = $2^{20}$
Figure 7.52 Radix sort: Total execution time for varying nodes, tasks_per_node = 8, problem size = $2^{24}$

Figure 7.53 Radix sort: Communication time for varying nodes, tasks_per_node = 8, problem size = $2^{24}$
Figure 7.54  Radix sort: Computation to communication ratio for varying nodes, tasks_per_node = 8, problem size = $2^{24}$
7.3.3 Varying problem size

Figure 7.55 Radix sort: Total execution time for process grid $4 \times 8$, varying problem size.
Figure 7.56  Radix sort: Communication time for process grid $4 \times 8$, varying problem size.

Figure 7.57  Radix sort: Computation to communication ratio for process grid $4 \times 8$, varying problem size.
Figure 7.58  Radix sort: Total execution time for process grid $4 \times 16$, varying problem size.

Figure 7.59  Radix sort: Communication time for process grid $4 \times 16$, varying problem size.
Figure 7.60  Radix sort: Computation to communication ratio for process grid 4 x 16, varying problem size.

Figure 7.61  Radix sort: Total execution time for process grid 8 x 8, varying problem size.
Figure 7.62  Radix sort: Communication time for process grid $8 \times 8$, varying problem size.

Figure 7.63  Radix sort: Computation to communication ratio for process grid $8 \times 8$, varying problem size.
8 Correctness tests

As a candidate for a production-quality communication library, GPSHMEM must be thoroughly tested. At the beginning, when we developed the first, not yet fully functional, version of GPSHMEM, our first testing code was a simple program calling `shmem_get`, `shmem_put` and `shmem_barrier`. A very simple matrix multiplication program was developed to estimate communication overheads introduced by GPSHMEM. With time, our testing code has grown into a simple collection of some function calls covering the major representative elements of GPSHMEM. This code did not scale well and it only ran on two processors. The collective procedures were tested, but only to a very limited extent.

All of the above was not a sufficient testing suite. It was mostly an ad hoc code written to roughly check some particular functionality without a thorough coverage of the GPSHMEM interface. After the GPSHMEM code was rewritten and the second edition was ready, a more complete testing program was developed. The major design goals included:

1. covering the majority of the functions implemented by GPSHMEM;

2. maintaining an ability to run on an arbitrary number of processors; and

3. not relying on highly regular patterns of data for data transfers.

The last point might call for some explanation. Consider a possible testing code for `shmem_get`. The testing code would fetch some data from a remote processor and compare it with the expected outcome. The question is, what the expected outcome should
be. It cannot be data obtained from the system random number generator, since in general, there is no guarantee that the "random" numbers will be the same on each processor, even if it has been initialized with the same seed. The other obvious way is to use some constant numbers or some other regular pattern. Since this solution has the potential to mask some obscure errors, the test plan avoided it. In effect, we have used an almost trivial, linear congruence that generates sequences of various numbers that can be reproduced on each participating node.
9 C and Fortran interoperability

The SHMEM library on Cray T3D provided programming interface for both C and Fortran. The Fortran interface included both Fortran 77 and Fortran 90/95. GPSHMEM provided much the same support for programming languages as SHMEM did. Nearly all of the source code was written in C, and so the C interface emerged in the natural way. The remaining part was to provide the Fortran bindings.

Of course, writing a completely separate set of routines in Fortran should not be considered the first solution. C routines, under some circumstances can usually be called without trouble from Fortran in the same way Fortran subroutines are. There are three conditions that need to be met in order for that to work:

1. the argument passing conventions must agree;

2. the values must be returned in the same way; and

3. the function names must match.

Since none of the SHMEM functions returned any values, the second requirement was not very restrictive. At the time of initial implementation, the only GPSHMEM function returning a value was gpshmem_init, but since the initialization of GPSHMEM is handled in a special way, this was still not an issue. Later, an additional function gpshmem_time returning a value of type double, was added; but it is an extension of GPSHMEM and no essential functionality would be lost in case this function turned out not to return its value properly. Of course, the programmer would need not to rely on
the results of this function in such a situation. Although this seems like a real problem, it is mostly a theoretical possibility—we have not encountered a platform on which this behavior occurs.

The first and the last issue could potentially prevent GPSHMEM from exporting its interface for both languages in the same library file. Consider the following set of circumstances:

1. The Fortran compiler uses a different calling convention than the C compiler.

2. The external symbols are stored in object files in the same way by both the C and the Fortran compiler.

In such situations, a C function, say shmem_put, could be accessed directly from the Fortran code, but due to the difference in the argument passing, it could not be used. Moreover, no special Fortran replacement could be provided, since it would cause a collision with the existing C function. Fortunately for us, even though compilers where such scenario could occur exist, special options can be used to allow disambiguation between symbols generated by C and Fortran compilers.

The argument passing conventions in C and Fortran are almost always different. In Fortran 77 the changes made to the the function’s arguments by the callee are visible to the caller (that is, Fortran uses call by reference). Thus, in order to allow a C function to be used from Fortran code, its arguments must be declared as pointers to the expected values. It is assumed here that the value representation of C and Fortran data types are identical.

Satisfying the last requirement depends on the name mangling applied by the C and Fortran compilers. This name mangling in either case is nowhere near as complicated as the name mangling in C++; however, it can be different on different platforms. Most of the time, the C compiler does not mangle the symbol names in any way. A symbol foo declared in the C source code appears as foo in the object file. Fortran compilers
behave differently. We have identified three different ways in which symbol names are
treated in Fortran compilers:

1. An underscore character (\_\_) is appended to the name. Symbol foo appears as
   foo\_\_ in the object file.

2. An underscore is appended to symbols that do not already contain underscores,
   otherwise two underscores are added. For example, symbol foo, as previously is
   translated into foo\_, while foo\_bar becomes foo\_bar\__

3. The symbol name is capitalized. Symbol foo is stored in the object file as F00.

Assume the first option and consider the function shmem\_put declared as

```c
void shmem\_put (void* target, void* source, int nlong, int pe);
```

In order to make it available from Fortran, there needs to be a corresponding function
shmem\_put\_:\n
```c
void shmem\_put\_ (void* target, void* source, int* pnlong, int* ppe);
```

If, for every C function there was a corresponding function with an underscore added at
the end of its name, it would create a Fortran interface for GPSHMEM. This is exactly
what we have done. For each C function, we have added a function that conforms
to the Fortran naming and calling conventions, and calls either its C counterpart or
the GPSHMEM internal routines. Since a large part of the GPSHMEM functionality
is implemented by internal routines operating on sequences of bytes instead of words,
both the C and the Fortran interface functions usually consist of simple calls to these
byte-driven functions. In the first release of GPSHMEM, all functions were completely
separate and so the Fortran interface was a thin layer on top of the C interface.

The only remaining problem is ensuring that, in the presence of such variety of differ­
ent Fortran name mangling options, the names in the object files will match. All of the
function names exported by GPSHMEM, either as a part of the C or the Fortran interface are preprocessed before compilation. All of those names are macros, replaced with proper values by the C preprocessor. This additional step adds flexibility in mapping of the C and Fortran names to a common representation.

Thanks to the flexibility enabled by this preprocessing stage, we have decided to add an optional prefix to all functions exported by GPSHMEM. The user can set the prefix to any value, including empty string, at the compile time. By default, the prefix "gp" is added. This allows coexistence of GPSHMEM with other SHMEM implementations and provides additional an means of avoiding collisions with existing libraries.
10 Conclusions and future work

The development of GPSHMEM provides a generalized, portable version of SHMEM. This solves the problem of lacking a widely available SHMEM programming interface. GPSHMEM implements the majority of SHMEM interface, including one-sided communication—the most essential functionality of SHMEM. The lack of some less-often-used features is compensated with extensions that can be more applicable in a variety of different hardware configurations. GPSHMEM relies on well-known and well-established portable communication libraries. Continued support and development of these libraries allows for easy porting of GPSHMEM to a wide range of computer systems. The performance tests clearly demonstrate that the scaling potential of computational algorithms is not obstructed by GPSHMEM. We are convinced that GPSHMEM provides a functionable implementation of the distributed shared memory programming model without the performance penalty imposed by intricate memory coherency protocols.

The choice of ARMCI for the underlying data-movement engine left us with little choice regarding the selection of the programming language for implementing GPSHMEM: it was either C or C++. The main goal of GPSHMEM was efficiency and portability and while both of these languages are viable candidates, we decided that C was a better choice. C is still more popular and more available than C++ and is also far less complex. We have not encountered serious design challenges during implementation of GPSHMEM that would have significant influence on its functionality. The first version of GPSHMEM suffered from a code bloat caused by function implementations
differing only in minor details. The second version has been refactored and the common functionality has been extracted into a separate byte-driven layer. Besides reducing the size of the source code, it has increased its modularity and thus ease of maintenance.

The future work on GPSHMEM could include extending GPSHMEM's application programming interface to all functions from the T3E edition of SHMEM. In addition to that, various techniques of collective communication implementation could be explored (for example use of binomial trees [31] instead of binary trees). A case study in porting large scientific applications from Cray T3D or SGI Origin 2000 to GPSHMEM should be made in order to further evaluate GPSHMEM's applicability and detect possible transition problems.
PART II

The Fortran 77 preprocessor
11 Introduction

The SHMEM User's Manual for C specifies what objects can be used as remote objects in SHMEM calls. In some cases, a "symmetry" of a data object in Fortran 77 is achieved by using special compiler-specific directives. One way, for example, would be to use the Cray pointers, described briefly in section 3.7. The only two ways that do not require any vendor-specific extensions are the use of COMMON blocks and SAVE statements; thus, Fortran 77 code that was meant to run with implementations of SHMEM provided by different vendors was required to restrict its use of SHMEM calls to objects with symmetry achieved in one of these two ways.

Because GPSHMEM only allows the use of objects allocated via explicit calls to its memory allocation routines, neither of the two ways mentioned above are a valid means of obtaining remotely accessible data objects compatible with GPSHMEM. This severely limits the usefulness of GPSHMEM with existing Fortran 77 code and makes porting time-consuming and, although not very complicated, error-prone. To alleviate this problem, we have thought of a way of automatically converting data objects that would be symmetric in SHMEM into data objects symmetric under the requirements of GPSHMEM. Such automatic conversion would be performed by a tool taking a set of Fortran 77 source files and producing corresponding Fortran 77 code in which all objects being subject to conversion are replaced with objects allocated via explicit GPSHMEM memory allocation calls. Obviously, such conversion would not only have to replace all existing declarations of these objects, but also "correct" all uses of affected entities.

Symmetric data objects are remotely accessible objects in SHMEM. See chapter 3.
We had two kinds of objects to deal with—those in common blocks and those with a SAVE attribute. Common blocks in Fortran 77 are the way to declare global variables that are accessible to all functions and subroutines willing to have such access. Because of this, not all Fortran 77 objects can be declared as members of a common block. The SAVE statement prevents a local object from becoming undefined between function calls.

The contribution of this part of the work is a tool, called \textit{fgpp} that performs such transformations. \textit{fgpp} takes a set of Fortran 77 source files and generates an output Fortran 77 code prepared to be compiled and linked with GPSHMEM. \textit{fgpp} is a whole-program preprocessor—it generates its output based on the information gathered from a collection of source files instead of processing one file at a time. It functions in a similar manner to the way that a C preprocessor works, that is, it "executes" \texttt{INCLUDE} statements and produces compilable code that, as a consequence, is often not appropriate for a long term maintenance. The suggested sequence of compilation steps involving \textit{fgpp} would be the following:

1. Given any .F files, preprocess them with a C preprocessor to obtain a set of .f files.
2. Given a collection of .f files, that is, Fortran 77 source code without any additional non-Fortran directives, preprocess all of them at once with \textit{fgpp}.
3. Optionally perform any final preprocessing (for example, substituting names of intrinsic functions, etc.).
4. Compile.

The next chapters describe our findings and motivations for the decisions that evolved during the development process. Chapter 12 contains introduction to the Fortran 77 programming language intended to help readers unfamiliar with this language understand the issues discussed in the later parts of this report. Chapter 13 discusses the evolution
of the definition of our translation process. Later chapters concentrate on describing particular implementation issues associated with different aspects and constructs of Fortran 77.
12 Introduction to Fortran 77

In this chapter we will introduce the Fortran 77 language as a preparation for the discussion that will follow. As most people know, the Fortran programming language has a very long history. Actually, it is the oldest programming language that, compared to an assembly language, can be deemed high-level. The development of it started in the early 1950s and since then Fortran has come a long way. Many different revisions of the language have been created, the most prominent of which is the one known as Fortran 77. This edition of Fortran has been standardized in an ANSI document ANSI X3.9-1978, Programming Language FORTRAN [63]. Since then Fortran has evolved significantly. The version currently under development, known as Fortran 2000, hardly resembles its predecessors.

Despite its long history, Fortran is not the most popular programming language today. Some of its restrictions and limitations, as well as the growing popularity of C and later C++, have caused it to gradually lose its role as a primary language for general software development. Therefore, it seems reasonable not to assume any detailed knowledge of the Fortran 77 concepts not only among software developers, but also among computer science researchers. To allow everyone to understand the contents of this part of the report, we present some background information on Fortran 77 that is necessary to understand the translation issues. It may be worth mentioning that this is not intended to be a Fortran 77 tutorial or a reference. Readers seeking such material are encouraged to refer to other publications.

Since this part of the research work was devoted to processing and translating Fortran
12.1 Data types

Like almost any other programming language, Fortran data objects have types. To be more precise, the type is associated with a variable (like in C), and not with the value stored in it (as in the case of Scheme or Lisp). The Fortran data types are

1. INTEGER: signed integer data type;
2. REAL: single precision floating point;
3. DOUBLE PRECISION: double precision floating point;
4. COMPLEX: single precision complex value;
5. LOGICAL: a logical value, either .TRUE. or .FALSE.; and
6. CHARACTER: character data.

In addition to that, a type DOUBLE COMPLEX is often implemented as an extension. It represents complex values, where both, the real and the imaginary parts are double precision floating point values.

The Fortran 77 data types can be divided in two categories: numerical and character. The reason for such classification is the way in which the values of those types are stored in memory. Values of all data types are allocated in terms of storage units. There are two kinds of storage units: numeric and character. It is not difficult to guess that the CHARACTER data use the character storage units. For all other types, a single datum of that type occupies a single numeric storage unit, except for DOUBLE PRECISION and COMPLEX, where it will require two units. In particular, a single LOGICAL datum, although only allowed to contain one of the two truth values, still occupies a single numeric storage unit. That is the same amount of memory that a single INTEGER value takes.
One of the serious limitations of Fortran 77 is that it does not allow programmers to define their own data types. Therefore all data in a Fortran program have to be stored in a set of variables that may not appear to be logically connected at the first glance and may have data types that may not immediately reflect their intended use. Compound data structures (like records in Pascal or structs in C) were introduced in Fortran 90.

12.2 Variables and arrays

A Fortran variable is a symbolic name that refers to a single datum of some data type. Fortran's concept of a variable is not different in that respect from nearly all other programming languages. Fortran 77 also has the means to accommodate sets of data of the same type organized as sequences (vectors) or multi-dimensional arrays. Fortran's arrays can have up to 7 dimensions. Conceptually, arrays in Fortran are very similar to those in C and C++, although there are a few differences between the two. First of all, besides the limitations on the number of dimensions that are less strict in C,

\[ \text{int foo[3][4];} \]

declares a two dimensional array of elements of type \text{int}, of size \(3 \times 4\). Another, more formal way to describe it is to say that the declaration above declares an array of 3 elements. Each of these elements is a 4-element array of type \text{int}. In other words, multidimensional arrays in C (and in C++) are just arrays of arrays. In consequence, given the declaration above, an expression \text{foo[2]} is perfectly valid and denotes an element of type \text{int[4]}. Assume that we have an equivalent declaration in Fortran:

\[ \text{INTEGER BAR(3,4)} \]

Now, \text{BAR} is just a two-dimensional array and not an array of arrays. An array reference \text{BAR(2)}, although syntactically correct, is not valid in light of the above declaration.

\(^1\)The C standard recommends that an implementation allows at least 12 dimensions. In case of C++, the suggested minimum number is 256.
Another difference between arrays in C and Fortran is the storage sequence. Let us take a closer look at the C declaration. As stated above, it declares an array of 3 elements. Let's momentarily forget what the type of these elements is. This implies that each of these 3 elements must occupy a contiguous area of memory. Let us now recall that each of these elements is a 4-element array. This reasoning implies that the second index should vary faster than the first in the storage sequence. It can be easily extended to show that in case of C arrays, the last index always varies the fastest in the storage sequence. Fortran, on the other hand, does not assume anything about the logical structures of its arrays. Therefore, the layout cannot be deduced from the language definition describing the arrays. It has been decided that the first index will vary the fastest.

This difference in array layout does not seem to be really significant, since programmers should not really be concerned about how data is represented in memory. After all, the language should provide enough means to manipulate the data without resorting to direct memory accesses. If the algorithm does not specify it, the order of traversing a multidimensional array should not matter, at least as far as the result is concerned. An order of traversing array elements can, however, have a substantial impact on the implementation's performance. A code that follows the storage sequence will take advantage of the processor's cache and thus execute significantly faster than code that does not. Therefore, it is important to keep in mind how a given programming language organizes its compound data structures. The importance of such differences becomes even more obvious when language interoperability is required. In such a case, ignorance about the layout will not result in a code that works slower, but in a code that simply does not work at all.
12.3 Strings

The CHARACTER data by themselves represent single characters. On the other hand, often times there is a need to store various messages that the program will display to the user in different circumstances. Since strings of length 1 are often inadequate for most applications, a mechanism for creating sequences of characters was introduced. For character objects a length specification can be provided. Length specification is a positive integer and specifies the number of character storage units allocated in a sequence. This extends the CHARACTER data type to allow strings of arbitrary lengths, although lengths of strings must be specified at the compile time.

Some compiler vendors extended the notion of length specification from character strings to other data types. Such extensions, although common, are not a part of the official language standard and should be used with care, as their behavior is implementation dependent.

12.4 Expressions and constants

All arithmetic computation in Fortran is done via expressions. Fortran was created as a tool to simplify and unify programming of mathematical formulas (after all, Fortran is an abbreviation of "FORmula TRANslator") and thus the syntax (as well as precedence and associativity) of expressions mimics that used in mathematics. For example, 1+2, X*Y, and X+2*Y-Z/(1+Y) are all valid expressions. In addition to the four arithmetic operators (+, -, *, /), Fortran offers an exponentiation operator ** and a string concatenation operator //. Numbers (literals) in Fortran are normally represented in decimal notation. There is a common extension to Fortran known as "typeless constants" that allows the programmer to specify data in hexadecimal, octal or binary representation, although the exact semantics of them does not seem to be uniform across implementations. Assume that a variable A is of type REAL. We know of at least two compilers
where an assignment \( A = X'01234567' \) produces different results.\(^2\) In one case, the text, "01234567", is treated as an integer represented in a hexadecimal form, later converted to a value of type \( \text{REAL} \). In the other case, the "01234567" is taken as a bit pattern that is stored in the memory location associated with \( A \). This provides strong evidence that one should not rely on non-standardized features and that such constructs should be avoided whenever possible.

A function call is traditionally denoted by \( \text{FOO}(X) \) or \( \text{BAR}(X,Y,Z) \), etc. References to array elements have a syntax identical to function calls, so one can imagine that an array is a function given as a "black box" that returns whatever value was in memory at a location specified by the set of arguments.\(^3\)

Besides arithmetic expressions, Fortran provides conditional expressions. Basic arithmetic comparison operators are \( .LT. \) (less than), \( .LE. \) (less than or equal), \( .GT. \) (greater than), \( .GE. \) (greater than or equal), \( .EQ. \) (equal) and finally \( .NE. \) (not equal). Logical operators are \( .NOT. \) (negation), \( .AND. \) (conjunction), \( .OR. \) (disjunction), \( .EQV. \) (equivalence) and \( .NEQV. \) (nonequivalence).

12.5 Fixed form, comments

Fortran 77 is known for its almost infamous "fixed form." The fixed form has originated from the punch cards, invented in 1896 by Herman Hollerith [56]. A single card had 80 columns and so with one character per column, the maximum physical line length was 80 characters. In practice, all characters placed in column 73 and beyond were ignored. The first 5 columns were reserved for an optional statement label. A statement label consists of, at most, 5 digits, at least one of which must be nonzero. This leaves columns 7-72 for the actual text of a statement. To avoid problems with longer state-

\(^2\)The notation "\( X'\ldots' \)" denotes a hexadecimal format.

\(^3\)Of course, this is a purely conceptual unification, as arrays and functions are completely different objects from the point of view of the language definition.
ments, a line could be broken down into smaller segments: the initial line and, at most, 19 continuation lines. A continuation line was indicated by a character other than blank or '0' in the sixth column and blanks in the first five columns.

Any line that contains either only blank characters in all columns or a letter 'C' or an asterisk (*) in the first column was ignored. Such lines are used to provide comments in the source code. Although the original Fortran 77 did not support it, many compilers allowed comments that extend from an exclamation point to the end of the line (just like // - comments in C++ or Java, or # in shell scripts or Perl).

12.6 Fortran declarations, common blocks

12.6.1 Basic declarations

Knowing what data types and what basic data structures Fortran has to offer we can examine syntax used to declare those object in a source code. A basic form of a declaration is as follows:

\[ \text{type lengspec name lengspec dimensions} \]

where type is the name of the type or a keyword DIMENSION, lengspec is the length specification (officially valid only for CHARACTER data types), name is the name of the object being declared and dimensions is a parenthesized list of dimension specifications. Declaration of other entities of the same type can be specified with the same statement by adding a comma-separated list of the "name lengspec dimensions" parts of a declaration. If the lengspec immediately following the name is omitted, the length specification is assumed to be the same as the one immediately after type. If both are omitted, the length specification is assumed to be 1. All components of a declaration (except, of course, name) are optional. Before proceeding with more details, let us see some examples.
1. INTEGER FOO

The simplest form of an explicit declaration. Declares a single variable FOO of type INTEGER.

2. CHARACTER*10 BAR, BAZ(3)*20

Declares two variables: BAR, and BAZ. BAR is of type CHARACTER*10 (i.e. it is a string of length 10), and BAZ is of type CHARACTER*20.

3. REAL QUUX(10:20,30)

Declares a two-dimensional array QUUX of type REAL. The first index of this array varies from 10 to 20 (all inclusive) and the second index varies from 1 to 30. The total number of elements in this array is thus $11 \times 30 = 330$.

4. DIMENSION F00BAR(10)

Declares a one-dimensional array F00BAR with 10 elements. Unless otherwise specified in another place in the program unit, the type of the array is assumed to be REAL.

As stated above, most of the components of a declaration can be omitted. In fact, the parts not specified in previous declaration statements can be added later. This means that the full information about the object being declared can span across several statements. Consider the following example:

```
INTEGER FOO

DIMENSION FOO(3,4)
```

The first statement defines the type of FOO to be INTEGER. The second statement makes FOO an array of size $3 \times 4$. The keyword DIMENSION is used to declare an array without explicitly specifying the type of the array. If the type is not defined elsewhere, it is assumed according to the implicit typing rules (see section 12.6.2 for more information on implicit typing).
Array dimensions are declared by a list of dimension specifications. Each dimension is defined by a pair of values: the lower and the upper bound. The lower bound is optional and it is assumed to be 1. The declarations

\begin{verbatim}
REAL FOO(10)
COMPLEX BAR(-3:3)
\end{verbatim}

specify an array FOO with 10 elements indexed by numbers 1...10 and an array BAR with 7 elements with indices in the range $-3, \ldots, 0, \ldots, 3$.

12.6.2 Implicit declarations

Fortran 77 does not require that a type is explicitly associated with every symbol used in a program. In such a case, the type of the variable is deduced automatically, based on the first letter in the name of the symbol. By default, the rules for implicit typing are as follows:

1. If the first letter of the name is A-H or O-Z, the type is REAL,
2. otherwise it is INTEGER.

The programmer can change the default setting with the \texttt{IMPLICIT} statement. It takes a list of type specifications, each of which is a name of the type followed by a list of letter groups. For example:

1. \texttt{IMPLICIT REAL(A-Z)} makes all symbols assume the type REAL by default,
2. \texttt{IMPLICIT COMPLEX(C-D,Z), LOGICAL(O)} makes all symbols beginning with letters C, D and Z implicitly of type COMPLEX, and all symbols beginning with O to type LOGICAL.

\footnote{This is another difference between arrays in Fortran and in C.}
Implicit declarations are regarded by many programmers as a misfeature rather than as an advantage. They can cause typographical errors to pass the compilation without being detected and cause erroneous behavior that is nearly impossible to track down. There are famous urban legends about an alleged crash of a space rocket caused by a Fortran statement similar to

\[ \text{DO 100 } I=1.50 \]

It was intended to be \text{DO 100 } I=1,50 (with a comma instead of a dot), but due to ignoring blank characters it was interpreted as

\[ \text{DO100I} = 1.50 \]

that is, as an assignment of a value 1.50 to an implicitly declared variable DO100I. Various versions of this story circulate in the computer science community, but there is no evidence of any tragic accidents ever having been caused by such an error. In any case, this does not seem unlikely and everyone who has written significant amount of code in C, C++ or Fortran has made errors similar to this one. To provide a way to avoid such problems, most, if not all compilers allow an implicit statement of form

\text{IMPLICIT NONE}

that turns off all implicit typing. It causes an error on an occurrence of a symbol that has not been explicitly declared. Use of \text{IMPLICIT NONE}, although it is not a part of the standard, has, nevertheless been regarded as good programming practice for a long time.

12.6.3 Parameters

A parameter in Fortran terminology is a symbolic name for a numeric or a character constant. It is conceptually the closest to \text{const} variables in C++. It is slightly different
from C macros, although the differences are not significant and bear no consequences to the understanding of Fortran's parameters in the scope of our treatment.

The syntax used to define a parameter is as follows:

```
PARAMETER (name=value1, name2=value2, ...)
```

There must be at least one pair `name=value` on the list following the `PARAMETER` keyword. Each of the names appearing on the list becomes a symbolic name given to the corresponding value. The values must be *arithmetic constant expressions*, that is, they must be expressions involving literal constants, arithmetic operators and previously defined parameters. No function calls are allowed. Parameters have types, either given explicitly or obtained by applying implicit typing rules. Parameters can be used to specify index ranges in array declarations. For example, the declarations

```
PARAMETER (N=5, M=2**N)
REAL A(N:M, 3)
```

are equivalent with

```
REAL A(5:32, 3)
```

### 12.6.4 Statement functions

A statement function is often seen as a short macro provided to avoid repetition of some frequently used expression. Whether it really is expanded as if it was a macro\(^5\) or not depends on the implementation. A statement function is defined by a statement function statement:

```
name (arg1, arg2, ...) = expression
```

For example,

\(^5\)Fortran 77 does not have any support for macros in the similar way that C does. The term "macro" is used here informally.
\[ \text{FOO}(X, Y) = X^2 + Y^2 \]

Now consider the following scenario:

\begin{verbatim}
REAL FOO(10,10)
INTEGER I, J
C ...
FOO(I,J) = I+J
\end{verbatim}

The last line definitely meets the syntax of a statement function statement, although, obviously, it is an assignment to an array element. This demonstrates that statement function statements and assignments may be indistinguishable with respect to their syntax. Context-sensitive rules, such as placement of the statement and previous declarations, can be used to disambiguate these two cases.

12.6.5 Common blocks, SAVE

In languages like C, C++ or Pascal, a programmer can declare a variable outside of any program unit. Such a variable is then a global variable, that is, it is in the global scope, visible and accessible from every program unit. In Fortran 77, however, all declarations must lie within a scope of some subprogram, possibly the main program. Since having global variables is often desired, there is another way to achieve a similar effect: a common block. In section 15.9.4 of the standard [63], we read “A common block provides a means of communication between external procedures or between a main program and external procedures. The variables and arrays in a common block may be defined and referenced in all subprograms that contain a declaration of that common block.”

A common block is an optionally named, ordered sequence of variables and arrays. Suppose we have declared a common block CB in a function FOO. If FOO calls BAR and BAR also contains a declaration of CB, the contents of the common block CB are preserved
after the execution of BAR terminates. However, if the function FOO was called by a subprogram that did not declare CB, the variables in CB can become undefined after FOO returns. Thus, the common blocks are not the exact counterparts of global variables. A programmer can, however, prevent the contents of a common block, or any local variable or array, from becoming undefined by specifying the name of the common block in a SAVE statement.\(^6\)

A common block is declared as follows:

\[
\text{COMMON } /\text{name}/ \text{ declarator-list}
\]

The name may be empty, in such case the pair of slashes (//) is optional. A common block declared with an empty name is called a blank common block and it has some special properties that differentiate it from a named common block. Each element of the declarator-list is either a variable name or an array declarator of form \(\text{name}(\text{dimensions})\). For example,

\[
\text{COMMON } // \ A, B \\
\text{COMMON } /\text{BAR}/ \ X(5,5), Y, Z(-1:1)
\]

Several common blocks can be defined in a single COMMON statement:

\[
\text{COMMON } /\text{FOO}/ \ A, B /\text{BAR}/ \ X, Y(10)
\]

The SAVE statement takes a list of variables or names of common blocks (those should be enclosed in slashes), as in the example below:

\[
\text{SAVE } /\text{FOO}/, \ \text{BAR}
\]

On Cray T3D and T3E, objects in common blocks and variables and arrays specified in a SAVE statement are symmetric data objects. Symmetric data objects can be used as remote locations in SHMEM calls. These special properties of common blocks and SAVE\(^6\)
objects are the center of our attention and are the main motivation for the translation efforts.

12.7 Program structure, program units

A Fortran 77 program consists of program units. A program unit is a subroutine, a function, a block defined as BLOCK DATA or a main program. Each program unit (possibly except for a main program) begins with a declaration using an appropriate keyword and ends with an END statement. The keywords are SUBROUTINE, FUNCTION, BLOCK DATA and PROGRAM and their meanings are self explanatory. Subroutines, functions and a main program are all forms of subprograms, that is, they are groups of executable statements. Functions differ from the other subprograms in that they return a value. Functions and subroutines can accept arguments. The set of variables that assumes the values passed to the program unit, in modern terminology called formal arguments or parameters, are called dummy arguments in the Fortran nomenclature. Before proceeding any further, let us see the syntax of function and subroutine declarations. A subroutine declaration takes the form of

\[
\text{SUBROUTINE name (arg}_1, \text{arg}_2, \ldots)\]

If the dummy argument list is empty, providing the empty pair of parentheses () is optional. A function declaration has one of the following forms:

\[
\text{FUNCTION name (arg}_1, \text{arg}_2, \ldots)\]

\[
\text{type FUNCTION name (arg}_1, \text{arg}_2, \ldots)\]

Unlike in the case of a subroutine, if the function takes no arguments, the empty pair of parentheses must still occur in the function declaration. If the type of the function is not given, it will be either deduced implicitly or it can be explicitly provided later as in the case of variables. The rules for implicit declarations were described in section 12.6.2.
All variables, arrays and array elements provided as actual arguments to a function or a subroutine are passed by reference. It is required that the type of the actual argument matches the type of the corresponding dummy argument. If a whole array is given as an argument then the corresponding dummy argument should also be an array and the corresponding elements of these two arrays are associated with each other. Let us have a look at the following example:

```
SUBROUTINE FOO(A)
    REAL A(3)
C
END

REAL B(3)
CALL FOO(B)
```

In this case, the element A(1) is associated with B(1), A(2) with B(2), and A(3) with B(3).

If an array element is passed as an argument and the corresponding dummy argument is an array, the association still occurs and begins with the element passed as the argument [63] (15.9.3.3). In the example above, the association would be identical if the call to the subroutine FOO was replaced with CALL FOO(B(1)). Consider the following snippet of code:

```
SUBROUTINE FOO(A)
    INTEGER A(5)
C
END

INTEGER B(10)
```
In this example, the element B(2) is associated with A(1), B(3) is associated with A(2) and so on.

This association is very important—it is the foundation on which the whole translation idea relies.

A function or a subroutine can have additional entry points, that is a point at which the execution of the subprogram can start. Each entry point is defined by an ENTRY statement. An entry statement can specify its own list of dummy arguments and can provide its own return type. Detailed rules governing the use entry points can be found in the language standard or in a Fortran 77 textbook.

The main program is the subprogram from which the program execution begins. It may optionally be started with a keyword PROGRAM followed by a symbolic name. The main program does not take any arguments, although some implementations allow it. If a statement other than a subprogram declaration occurs outside of a scope of a program unit, it is assumed to be a part of the main program. In such a case, all following statements, up to the END statement are considered to be the definition of the main program.

Program units declared with the BLOCK DATA keyword are used to provide initial values to objects in common blocks. More detailed explanation of what they are can be found in section 12.8 of this chapter.

For completeness, it is worth adding that besides variables, arrays and array elements, two more kinds of objects can be passed to a subprogram: other functions (or subroutines), or statement labels. In the latter case the corresponding dummy arguments must be denoted by asterisks and the subprogram can contain an alternate return statement terminating the execution of the current subprogram and indicating at which
of the given statements the execution should continue.

Functions and subroutines in Fortran 77 cannot be called recursively, either directly or indirectly.

12.8 DATA and BLOCK DATA statements

12.8.1 DATA and BLOCK DATA

The DATA statement is used to assign an initial value to a data object. For example,

```
DATA X /1.0/
DATA A,B,C /3*-1.0/
```

The values are enclosed in // and can contain repetition operators num*value. Only local variables and arrays can be given as arguments to DATA, and dummy arguments, in particular, cannot. The important restriction is that data objects declared as members of a common blocks cannot be initialized this way in a function, a subroutine or a main program. Such objects, however, except members of the blank common block, can be initialized in DATA statements in BLOCK DATA subprograms. A BLOCK DATA is a non-executable program unit whose only role is to specify such initialization. Common blocks are often allocated statically and, in such cases, their contents can be provided at the compile time.

12.8.2 Implied-DO lists

An array or array section can be initialized in a DATA statement, and, obviously, it cannot be done by specifying just the name of the array. The only way is to provide the list of array elements. In case of large arrays this might be cumbersome; and, if the array's dimensions are expressed by using parameters, it may be simply impossible. A
simpler way, that syntactically mimics a DO loop exists, hence it’s name—*implied-DO*
list. Let us begin with an example. The code

```fortran
REAL FOO(10)
DATA (FOO(I),I=1,10) /10*0.0/
```

initializes the array `FOO` with all zeroes, just as a loop with the variable `I` varying from
1 to 10 would do. The implied-DO list can take more complicated forms. In the snippet
below

```fortran
REAL FOO(10)
DATA (FOO(I),I=1,10,2) /5*-1.0/
```

only the odd-numbered elements of `FOO` are assigned an initial value of $-1.0$ (the variable
$I$ changes from 1 to 10 with an increment of 2.) Finally, implied-DO lists can be nested.
A lower triangular section of an array can be specified as follows:

```fortran
REAL FOO(10,10)
DATA ((FOO(I,J),J=1,I),I=1,10) /55*123.0/
```

All limits and increments of the “implied loop” control variables must be known at the
compile time. They must be either constant integer expressions or can involve control
variables of the enclosing implied-DO lists.

### 12.9 Other specification statements

#### 12.9.1 Function declarations

Besides those already described above there are two other significant kinds of spec-
ification statements, the function declarations: INTRINSIC and EXTERNAL. The syntax
for both of them is the same: appropriate keyword followed by a list of names. Intrinsic
functions are those provided by the implementation, like ABS for an absolute value, or
INT for conversion to the type INTEGER. External functions are all other functions that can be accessed in a given program unit. In addition to the function declaration, a type of its value can be explicitly specified:

```fortran
EXTERNAL FOO
INTEGER FOO
```

### 12.9.2 FORMAT statement

The `FORMAT` plays a role similar as the format string in the `printf` function in C: it defines the representation of data in the input/output statements. The `FORMAT` statement does not have any influence on any data objects and is almost insignificant in the translation process.

### 12.10 EQUIVALENCE statement

Fortran 77 does not have pointers, so there are no obvious ways to have two different objects refer to the same memory location. Such association takes place during function calls, where the dummy arguments are associated with the actual arguments. A programmer can cause two different dummy arguments to be associated by passing the same variable as two actual arguments. The `EQUIVALENCE` statement allows a more direct association between two entities. It takes a list of groups of items to be associated:

```fortran
EQUIVALENCE (A,B,C), (X(2),Y)
```

Note that arrays can also be associated by providing association of one of their elements. In addition to that, dummy arguments, symbols not referring to data objects (e.g. function names), cannot appear in an `EQUIVALENCE` statement. Special provisions apply to elements of common blocks: two elements of a common block cannot be associated, either directly or indirectly. An item associated with a common block member becomes,
implicitly, a member of that common block. *It is therefore possible to extend a previously declared common block:*

```plaintext
REAL X, A(10)
COMMON /FOO/ X
EQUIVALENCE (X, A(1))
```

Such extensions are only allowed if no data is prepended to the common block in question.

The `EQUIVALENCE` statement is considered by many programmers to be a can of worms and use of it is often regarded as bad programming practice.

### 12.11 Executable statements

As the name suggests, the executable statements are mostly those for which there is an associated action performed during the execution of the program. These are conditional statements (`IF, ELSE, etc.`), input/output statements (`READ, WRITE, etc.`), jump statements (`GOTO, CALL, RETURN`) and others. These statements were of much less concern for us from the point of view of the translation, since they do not provide any additional information that can influence the properties of the data objects used in the program. Because of that, we will not describe those statements in detail. Such information can be easily found in other, more specialized sources.

The only additional fact to keep in mind about input/output statements is that some of them can accept entire arrays as their arguments. In such cases, the array is treated as if all of its elements were specified in the order of the storage sequence. Such statements can also be provided with an implied-DO list specifying all elements in a correct order.
13 Designing the translator

We will now present the development of the requirements and expectation of the translation process. We will describe various approaches and possible issues associated with them. When we started thinking about the translator, we only had a very rough idea of what it should do. It could be summarized as "take Fortran 77 code, replace common blocks with objects allocated via GPSHMALLOC and generate new Fortran 77 code with these changes in it." Let us explain why we thought this was at all possible. After all, our idea could be "write code that will solve the halting problem for Turing Machines" and everybody knows that any attempt to actually implement it would simply be a waste of time.

The outline presented above has 3 components:

1. converting a Fortran 77 source code to some internal representation,

2. performing changes to the code,

3. generating a Fortran 77 source code that corresponds to the internal data structures after the changes have been made.

Depending on the depth of knowledge about Fortran 77, all or none of these steps can seem difficult. Even though lexical and syntactical analysis is no longer considered to be a research problem, and tools for automatic lexer and parser generation have been created (for example [81], [72]), Fortran seems to have been largely unaffected by these developments. Because of Fortran's legacy (described later in chapter 20), especially
because it ignores blank characters and allows Hollerith constants, lexical analyzers for Fortran 77 are usually hand-written. Once such a lexer is ready, the syntactical analysis can be left to an automatically generated parser. At one point we considered writing our own lexer and parser, but since it would require a significant amount of work we decided to find an open-source project with a license that allowed us to modify its code and use the syntax-related module contained therein. Our choices were the *G77* from the GNU Compiler Collection [47] and *f2c* from AT&T [44]. Since *f2c*'s license seemed more permissive, it became our choice.

The least number of problems was always associated with the last step. After all, the only thing that needs to be done in that step is traversing the data structures and generating the Fortran 77 source code that corresponds to the internal representation. There were just a few things that we needed to keep in mind. One of these things that could not possibly be relaxed was preserving the proper fixed format layout. The responsibility to store and keep enough information to properly unpars the internal representation was therefore assigned to the first two stages.

Making the actual changes was where the real problems were. Unless the changes could be made based purely on information that can be statically collected by an analysis of the source code, such translation would be impossible. There is a strong resemblance between common blocks and global variables in languages like C and C++. Such global variables are usually allocated (and initialized) by the compiler. They remain in memory from the beginning of execution of the program until the end of it. At various points in the program they may not be visible, but they still exist. A block of memory, dynamically allocated right at the beginning of the program execution and released immediately before its termination, has exactly the same properties, at least as far as the lifetime and access in languages like C is concerned. It may be placed in a different area of memory than the global variables, but it places no restrictions on the use of it. The other kind of objects that were symmetric on Cray T3D and T3E, the objects specified
in a SAVE statement, were not initially to be supported. Later, the support was added and the basis for it was the similarity of SAVEd objects and local variables declared as static in C. Such variables in C exist at the latest from the first time the function that contains them is called and until the program terminates. Once again, there is nothing that would significantly enough differentiate between the storage for SAVEd objects and a dynamically allocated memory block.

We did not yet fully realize the actual level of complexity of the second step of the translation process, but we did not see any immediate reasons for which such a conversion could not be performed. During the development of the GPSHMEM library we manually converted a HALO code [120] from its original SHMEM version to GPSHMEM. The conversion was not very difficult; it was actually rather straightforward. At that time, it was a very positive finding, since it helped us see that the GPSHMEM library was close enough to the original to be a good candidate for wider use. Later, it provided additional inspiration and motivation for developing the automatic converter. The changes made to the HALO code were:

1. adjusting the initialization and cleanup code to call appropriate GPSHMEM functions;

2. removing all common blocks and adding corresponding calls to GPSHMALLOC;

3. replacing references to all objects previously in common blocks with appropriate references to the allocated memory blocks; and

4. changing some intrinsic functions used in the code to their functional counterparts available on our machines.

The last step was not related to SHMEM or GPSHMEM in any way, and so the need for it was not a concern. The first three steps, although somewhat tedious, were rather simple and even when performed manually were still done "mechanically." Now, let us
discuss the second and the third in more detail. Imagine that we have the following piece of Fortran code:

```fortran
REAL X
COMMON /FOO/ X
X = 1.23
```

In accordance with the requirements of GPSHmalloc, the code after conversion should look approximately like this:

```fortran
INTEGER HANDLE, INDEX
C allocate 1 element of type REAL
HANDLE = GPSHmalloc (real_type_id, 1)
C obtain the index
INDEX = GPSHIndex (real_type_id, HANDLE)
C replace X = 1.23
GPS_REAL(INDEX) = 1.23
```

That was a really simple example. Suppose we have two objects in a common block:

```fortran
REAL A, B
COMMON /FOO/ A, B
A = 1.0
B = -1.0
```

We now have several possible solutions. One of them, the simplest one, would be to allocate a separate block of memory for each of the elements of the common block:

```fortran
C initialize all
INTEGER H1, I1
INTEGER H2, I2
```
HI = GPSMALLOC (real type id, 1)
I1 = GPSHINDEX (real type id, HI)
H2 = GPSMALLOC (real type id, 1)
I2 = GPSHINDEX (real type id, H2)

C replace A and B in the assignments

GPS_REAL(I1) = 1.0
GPS_REAL(I2) = -1.0

This could work, although it does not seem right. First of all, it can cause a massive
memory fragmentation. A multitude of tiny blocks will waste a lot of memory, as there is
housekeeping information associated with each block. In the above scenario, the length of
the allocated blocks is 4 bytes (at least at most platforms). The "hidden" information
stored with each block is usually longer than that, so actually most of the allocated
memory would be used for holding maintenance data. To implement this, all common
block definitions would have to be collected. Then, for each common block member an
allocation code would have to be generated. Also, all occurrences of that symbol would
have to be appropriately adjusted to refer to the new location. This approach, despite
its simplicity was not considered a viable option. We had several other ideas in our
minds:

1. Allocate a single block, large enough to hold all members of all common blocks.

2. Allocate a single block for each common block.

3. Allocate a single block for each data type.

The first of the above is obviously the opposite of the "single common block member,
single memory block" approach. Again, once the information about all of the common
blocks has been collected, the sizes of the common blocks would have to be calculated.
Then, a routine would need to be generated to allocate a single memory area. Since each
common block would occupy its own segment of that area, a location of each common block within this area would have to be calculated. Having laid the common blocks out, we could finally calculate the final locations of the members of the common blocks and put them in place of the references to them in the original code. Compared to the very first idea, several more steps seem to be involved; but, in fact, the implementation would most likely not be much more involved.

The most natural approach to the common block resolution is to take each common block separately, prepare a replacement and make all necessary changes to the references to its original members. That is what the second idea above is all about. It seemed reasonable; we did not see any immediate problems with it and it did not seem any worse than any of the other proposed solutions. That is the way we chose for our implementation.

Because of the way the memory allocation for Fortran 77 works in GPShMEM, all data needs to be properly aligned. In general, all data of type $T$ needs to be aligned to a memory boundary that is a multiple of the size of type $T$. For example, if the size of INTEGER is 4, then all data objects of type INTEGER must be aligned to a multiple of 4. Therefore, if an object of type CHARACTER is surrounded by objects of type INTEGER, there must be some padding inserted between these objects. This did not seem to be a big problem, and we were not concerned about it at all. After our initial design and implementation was ready, we thought of some possible ways to avoid padding insertion. The first would be to rearrange the objects in the common block. For example,

```
REAL A, B

COMPLEX Z

COMMON /FOO/ A, Z, B
```

would be laid out in memory as Z A B. Let's quickly recap Fortran's storage principles: all data objects occupy a multiple of a storage unit associated with their type. Nearly all
types have a numerical storage unit associated with them, except for the type CHARACTER whose storage unit is, not surprisingly, a character storage unit. All non-character types, required by the Fortran 77 standard, occupy either 1 or 2 numerical storage units. There are some additional data types, not a part of the language, but often present as a language extension, namely INTEGER*8 and DOUBLE COMPLEX. They require 2 and 4 storage units, respectively. There is an important observation to be made. Take two different types. If they both need the same number of numerical storage units, it does not make any difference which one of them is placed in memory first, since there will be no need for any padding between them anyway. Assume then, that one of them takes \( n_1 \) numerical storage units and the other takes \( n_2 \), and that \( n_1 > n_2 \). Then, regardless of what types we have, \( n_1 \) will always be a multiple of \( n_2 \). It is not anything that the language definition requires, it just happens to be true for all types that we decided to support. Let's now assume that we need \( k \) items of the first type and that the size of the storage unit is \( s_U \) bytes. The total size needed, in bytes, would be

\[
S = k \ n_1 \ s_U.
\]

Assume now that we obtained a memory block beginning at address \( A \), aligned according to the requirements of the first type. This means that \( A \) is a multiple of \( n_1 \ s_U \), that is,

\[
A = n_1 \ s_U \ A'
\]

for some \( A' \). The memory area used by the first object would thus end at

\[
A + S = n_1 \ s_U \ A' + k \ n_1 \ s_U = n_1 \ s_U \ (A' + k) = n_2 \ n' \ s_U \ (A' + k) \quad \text{since} \ n_1 \ \text{is a multiple of} \ n_2
\]
for some $A''$. In other words, if an object of type with size $n_1$ is placed in a memory, beginning at a properly aligned address, the object will end at a memory address properly aligned for placement of the object of type with size $n_2$. Therefore, to find an arrangement that avoids all padding, we would only need to sort the common block members according to the sizes of their types in a descending order.

There may be other ways to eliminate padding. One more would be to gather all objects of the same types and allocate a single memory block for each such type. In such a case, all variables and arrays of type INTEGER would be placed, one after another, in a memory block devoted entirely to the type INTEGER. But this approach has no real advantages over the previous one.

The decision about how the “substitute” common blocks are allocated was not the last one to be made before we could proceed with the implementation. A memory block allocated via GPSHIMALLOC must be accessed via a reference to one of the GPSHMEM’s reference arrays. At least one index per an allocated block would be necessary. The question was whether or not one was enough. If there are two or more elements in a given common block, an approach with a single index would have to use a constant displacement to access all of these elements. For example, a common block FOO defined as below

```latex
REAL A,B
COMM /FOO/ A,B
A = 1.0
B = 2.0
```

would become

```latex
INTEGER HANDLE, INDEX
```
HANDLE = GPSHALLOC (real_type_id, 2)
INDEX = GPSHINDEX (real_type_id, HANDLE)

C  access A, no displacement necessary:
GPS_REAL(INDEX) = 1.0

C  access B, displacement (+1) needed:
GPS_REAL(INDEX+1) = 2.0

The advantage of this approach is that there are only two additional variables associated
with each memory block. Most memory accesses, however, require a runtime calculation
of the "effective address." Let us have a look at another example:

REAL A
INTEGER I
COMMON /FOO/ A, I
A = 1.0
I = 5

The idea of having a single index falls apart, as the same index cannot be used to
access objects of different types. Objects have to be accessed via reference arrays of
corresponding types and the same index cannot be used. There are two direct reasons
for that: first, the displacement cannot be computed from within the Fortran 77 program
and second, the reference arrays preserve the types of objects.

An alternative to this approach is to associate an index with each element of the
common block. It definitely solves the problem of different data types—each object is
accessed individually. Consider the piece of code presented above. After translation it
would look similar to the following:

INTEGER HANDLE
INTEGER INDEX_A, INDEX_I
HANDLE = GPshmAlloc (real_type_id, 2)
INDEX_A = GPshIndex (real_type_id, HANDLE)
INDEX_I = GPshIndex (integer_type_id, HANDLE) + 1
GPS_REAL(INDEX_A) = 1.0
GPS_REAL(INDEX_I) = 5

In addition to solving the problems of the previous idea, at the price of some initial setup the runtime penalty is eliminated. A minor detail calls for an explanation here. The memory allocation in line 3 above allocates a memory for 2 REALS. This does not cause any problems, as the sizes of types INTEGER and REAL are identical. The only thing that needs to be taken care of is to make sure that enough memory is allocated. Also, to keep control of the alignment, the actual implementation uses a multiples of type DOUBLE COMPLEX\(^1\) to allocate all memory blocks. Apart from meeting the requirements of the GPshMEM's reference arrays, the alignment of the common blocks can also speed up the sequential program execution. On many architectures, memory accesses at aligned boundaries are faster than at unaligned ones. It is known that lack of such alignment in the case of common blocks can result in performance degradation [111].

The actual allocation and initialization of the indices will take place in a specially generated routine. This routine will be called at the beginning of the execution of the program. Now, the values of the handle and the indices must be propagated to all other subprograms that used the originating common blocks. Fortran 77 already has a feature meant exactly for this purpose: common blocks. It may seem a little ironic that we use this mechanism to eliminate the uses of it elsewhere, but in this case the data contained in those common blocks do not need to be remotely accessible. Being shared among routines within the same process is completely sufficient. The following example should illustrate the idea. Assume this is the original code:

\(^1\)That is, the largest type available.
SUBROUTINE S
REAL A
INTEGER I
COMMON /FOO/ A, I
A = 1.0
I = 5
END

After transformation we should obtain something like this:

SUBROUTINE ALLOC
INTEGER HANDLE_FOO
INTEGER INDEX_FOO_A, INDEX_FOO_I
COMMON /GPSHMEM_FOO/ HANDLE_FOO, INDEX_FOO_A, INDEX_FOO_I
HANDLE_FOO = GPSHALLOC (real_type_id, 2)
INDEX_FOO_A = GPSHINDEX (real_type_id, HANDLE_FOO)
INDEX_FOO_I = GPSHINDEX (real_type_id, HANDLE_FOO)+1
END

C

SUBROUTINE S
COMMON /GPSHMEM_FOO/ HANDLE_FOO, INDEX_FOO_A, INDEX_FOO_I
GPS_REAL(INDEX_FOO_A) = 1.0
GPS_INT(INDEX_FOO_I) = 5
END

The outline of our translation idea is nearly complete. The major problems were identified and solved. We now have enough confidence that the substitution is indeed possible and that it can be done automatically. There is still one problem left: strings. In Fortran 77, a string of each specified length is almost like a separate data type—it
is not equivalent with an array of characters. If a string is placed in a common block, all of the references to it would have to be replaced with expressions of the same type. In other words, we would need to have reference arrays that would allow us to refer to strings of all possible lengths. Since GPSHMEM does not even have a reference array for a type `CHARACTER`, we have encountered the first problem that does not have an exact solution. Our first compromise was to disallow the use of strings (and `CHARACTER` objects in general) in common blocks. Later, we decided that the user will be able to specify which common blocks would be subjects to translation, so a program with common blocks that contain character objects would still have a chance to be processed.

One more question to be asked is what the generated code should look like. This may not seem like a real issue—it should look like Fortran 77 code after all. The real question is whether it should be human-readable or not. We have seen examples of automatically generated code\(^2\), and, in many cases user readability apparently was not on the top of the priority list. Initially, we planned to generate user-maintainable code and we had two reasons for that:

1. Having user-maintainable code after conversion would allow the translation to only happen once. Further maintenance efforts could therefore be carried out on the translated code.

2. We have already encountered problems that do not have accurate solutions. Some compromises were necessary and we had reasons to believe that our problems would not end there. Having a user-maintainable output would provide us with a safety exit: a possibility of user intervention in cases that would not otherwise be possible to handle.

This requirement was eventually dropped. It turned out that generating a code that humans would consider readable was much more difficult than we previously expected.

\(^2\)Parsers, lexers, etc.
We hope that the later sections will provide enough evidence to convince the readers that this decision was justified.

The following chapters of this part will provide details on the treatment of all Fortran 77 features that were of significance for us, along with the rationales for the decisions we had to make that have affected the functionality and usability of the translator.
14 Declarations

Declarations are the tools programmers use to describe the shape and type of data objects used in Fortran 77 code. Since the translation efforts are closely associated with data objects, the declaration statements are the focus of our attention. The remaining part of this chapter describes various issues that we encountered during the design and implementation process and our approaches to solve them.

14.1 Constants

As stated in the previous chapter, we initially tried to make the output code as easy to maintain by humans as possible. Given that decision we would have to comply with the POLA principle. In particular, this has significant consequences on the use of symbolic constants, i.e. parameters. Imagine that a programmer has written the following code:

```
PARAMETER (N=10)
REAL A(N)
COMMON /FOO/ A
C ...
DO 10 I=1,N
   A(I) = BAR(I)
10 CONTINUE
```

1 Principle Of the Least Astonishment.
If, after conversion, the PARAMETER statement disappears, each occurrence of \( N \) would have to be replaced with its assigned value of 10. Everybody would agree that this would defeat the purpose of using symbolic constants and that it would make the code much harder to maintain. Thus, we decided to keep all PARAMETER statements intact and preserve the use of them in all expressions. The principle we kept in mind was that the set of parameters used to manipulate the program's behavior before the translation should be the same. Imagine that the programmer created a constant, called \( \text{NITEMS} \) that would hold the number of items of some kind that the program would later work with. The program could then use the symbol \( \text{NITEMS} \) in array declarations, loop limits or in expressions. If, for any reason a square of \( \text{NITEMS} \) was necessary, the program could also have a definition of \( \text{NITEMS}^2 \) as \( \text{NITEMS} \times \text{NITEMS} \). In this case, it is still sufficient to change only the value of \( \text{NITEMS} \) to adjust the program's behavior to accommodate the new value. Our intent was then two-fold:

1. do not add or remove any places in which such changes would be necessary, and

2. do not change the number of parameters that need manual adjustment.

These assumptions did not appear to cause any problems. After all, all that we would need was a dictionary with names of constants and references to their values, given as some internal representation of an expression. Since we should not make any decisions depending on the actual values of parameters, such representation of PARAMETERS seemed sufficient.

14.2 Common blocks

All common blocks with the same name share their storage areas. More precisely, they share their first storage unit. By providing different definitions of a given com-
mon block in different subprograms, a programmer can achieve an association between different objects. Consider the following example:

```
SUBROUTINE S
  COMPLEX C
  COMMON /FOO/ C
END

SUBROUTINE T
  REAL R(2)
  COMMON /FOO/ R
END
```

The elements $R(1)$ and $R(2)$ in subroutine $T$ are associated with the real and the imaginary parts of the variable $C$ in the subroutine $S$. Although such an association may have legitimate uses, taking advantage of it is considered bad programming style. Nevertheless, we thought we might have to accept it and implement our translator accordingly.

The first approach considered consisted of two passes performed by two separate tools:

- **Pass 1**: Collect complete information about all common blocks and save this information to a file.
- **Pass 2**: Read the file created in pass 1 and process all source files, replacing common blocks according to the information from pass 1.

Having two separate tools had some advantages. First of all, the first one would give us information about all common blocks. Should that information turn out to be needed later, we would already have it. We also thought that it may be necessary to calculate the layouts of all instances of a given common block. For example, suppose that a
common block F00 was defined twice in two different ways? We believed that there may be situations in which, in order to preserve the original associations, we would need information about both of these common blocks before we could proceed with any changes. All of this information could be obtained by a single run of the program on a set of Fortran source files. After the first pass has completed, the second tool would iterate over a set of Fortran source files and apply the changes using the data collected in the first stage. A noteworthy observation to be made here is that the second-stage processing could be done one file at the time.

More detailed analysis has revealed some problems. Remember that we did not evaluate any of the constant arithmetic expressions in declarations. In other words, given

```fortran
PARAMETER (N=10)
REAL A(N)
```

we decided "not to know" that the size of A was 10. The reasons for that were already given, but, essentially, we must be able to allow the programmer to change the value of N to something else without affecting the meaning of the program. Therefore any decision that was made knowing the current value of N, and that would not have been made otherwise, would not be legitimate. Consider the following example:

```fortran
SUBROUTINE S
PARAMETER (N=10)
REAL A(N)
COMMON /F00/ A
END

C

SUBROUTINE T
PARAMETER (M=10)
```
There are two problems with this code. Assume it is time to generate the allocation function. Obviously, the GPSHALLOC function needs to know the size of the block to be allocated, but what exactly is the size in this case? A human reader can see that it is 10, but the parameters $N$ and $M$ are independent and the programmer should be allowed to manipulate them separately (even if both were called "N"). In case the values are different, we may choose to calculate the maximum of the two and assume it to be the size of the memory block. Here is where the second problem shows up. Fortran 77 standard requires all declarations of a given named common block to have the same size, regardless of whether the subsequent definitions are identical or not. Violation of this constraint is usually not diagnosed by a compiler, but our solution would prevent it from ever being diagnosed, even if the compiler would do this in the original code.

Redeclaring the same common block without preserving its size is a programmer's error. In addition it is a situation that most compilers would not detect. Because of that, the approach described above may seem acceptable. The problems, however, do not end there. Suppose we have the following source code:

```fortran
SUBROUTINE S
    REAL R
    COMPLEX C
    COMMON /FOO/ R,C
END

SUBROUTINE T
    REAL X,Y,Z
```
Because of the nature of GPSHMEM's memory allocator, all data of type T has to be aligned to a boundary that is a multiple of the size of T. In this particular example, assume that the size of REAL is 4. Then, the size of COMPLEX would be 2*4 = 8, according to the language definition. This implies that R must be aligned to a 4-byte boundary, and C must be aligned to an 8-byte boundary. Therefore, if R happens to be aligned to a 8-byte boundary as well, there would need to be 4-byte padding between R and C. We could make sure that FOO begins at an address that is an odd multiple of 4 and eliminate the need for padding. Unfortunately it is not a universal solution: if FOO is redeclared as

```
REAL Q,R
COMPLEX C,D
COMMON /FOO/ R,C,Q,D
```

then padding would be required somewhere, regardless of the initial placement of FOO. In the original code, the programmer might have assumed that Y and Z will be associated with the real and imaginary parts of C. Therefore, if padding was inserted somewhere, we could also accordingly separate the members X, Y and Z with padding to keep the association. As usual, there is an example illustrating the defects of this proposed solution. Suppose the common block FOO was also declared as

```
REAL X(3)
COMMON /FOO/ X
```

Now, X is an array and it must be contiguous. No padding is allowed between elements of an array.
We were unable to find a solution that would solve this problem in a satisfactory manner. We did not want to allow our code to just insert padding arbitrarily, since it could possibly change the meaning of the program. Therefore we decided that a compromise may be necessary: *we have chosen not to allow different definitions of the same common block.* As a result, we required all common blocks with a given name to have the same structure, that is, for a list of common block members, the corresponding list of their types would have to be the same for all given definitions. This requirement could also be rephrased in the following way: all definitions of a given common block must be identical with respect to the types of their members. The only differences could possibly be the names of variables and arrays contained in such common blocks.

Given this constraint, the translator would still need to maintain a list of definitions of all common blocks with the same name. It must do so, because the symbol names are important in the substitution process. Imagine we have two definitions of a common block BAR, both consisting of a single REAL variable. Suppose now, that this member is called X in one definition and Y in the other. Now, in the subprogram containing the first declaration, all occurrences of X would have to be replaced, whereas in the other subprogram, symbol Y would be the subject of translation. We realized that the requirement of an identical structure would already restrict the set of candidate common blocks to a certain subset of “nice” common blocks. Given that, an additional requirement that all corresponding members’ names be identical did not appear to add much to the restriction. It was not mandated by any translation issues, but it would significantly simplify the implementation of our translator. From now on, for each name of a common block, there would be only one corresponding definition of that block. This definition would not be associated with any particular subprogram; therefore, we could process all subprograms, collect the definitions of all common blocks and store them in a dictionary. If a common block declared in the source code was already present in the dictionary, the two declarations would be compared against each other. Should they be
different, an error would be signaled. Of course, a common block occurring for the first
time would be entered into the dictionary without further checks.

When handling common block declarations various erroneous situations need to be
recognized properly. The less obvious cases would include one common block being a
prefix of another, as in the following example:

\begin{verbatim}
COMMON /FOO/ X,Y,Z
\end{verbatim}

\begin{verbatim}
C and, in a different subprogram:
COMMON /FOO/ X,Y
\end{verbatim}

Both cases, corresponding to the two different orderings of these declarations, must be
detected. More common errors would include:

1. \textit{Type mismatch}: occurs when the corresponding members in the first and the
second declaration have different types, for example:

\begin{verbatim}
REAL X
COMMON /FOO/ X
\end{verbatim}

and

\begin{verbatim}
INTEGER I
COMMON /FOO/ I
\end{verbatim}

2. \textit{Name mismatch}: when corresponding entities have the same type, but differ in
names:

\begin{verbatim}
REAL X
COMMON /FOO/ X
\end{verbatim}

and
REAL Y
COMMON /FOO/ Y

3. Dimension mismatch: when the corresponding members of the common blocks have different dimension specification. Assume that the common block FOO was initially defined as follows:

REAL X(0:9,2)
COMMON /FOO/ X

Then, any of the following redeclarations do not match it:

- Same number of elements, but different array shape

  REAL X(20)
  COMMON /FOO/ X

- Same array shape, but different bounds

  REAL X(10,2)
  COMMON /FOO/ X

Also, because the constant arithmetic expressions were not evaluated, the following cases would be recognized as different:

- given two parameters N and M

  COMMON /FOO/ A(N)

  and

  COMMON /FOO/ A(M)

  regardless of the actual values of N and M.

- even without using parameters:
COMMON /FOO/ A(2+3)

and

COMMON /FOO/ A(3+2)

The last two cases seemed to be unfortunate consequences of our decisions, but we did not think they were serious enough to reject our current approach. When we eventually decided to change the translation idea from generating human-readable code to just preprocessing, we were no longer required to preserve any symbolic constants and thus arithmetic constant integral expressions were evaluated. Thanks to that, the problem went away, and 2 + 3 was again equal to 3 + 2 and N was recognized as equal to M, if their values were indeed equal. This was not the ultimate reason we changed the design of the translator to a preprocessor, but it was one of the arguments in favor of it, when we were considering that choice.

Finally, let us present some of the issues that need to be addressed when collecting Fortran's declarations. As we already showed, declarations in Fortran can span across several statements. In particular, any declared entity can be inserted later into a common block. Therefore, we have to treat each declaration as a possible declaration of a common block member. Another “curiosity” that needs to be mentioned is that even a declaration of a single common block in a single subprogram does not have to be contained within a single statement. Subsequent common block specifications in the same subprogram are merged. This means that the common block declaration

```
REAL X,Y,Z
COMMON /FOO/ X,Y,Z
```

can be rewritten as

```
REAL X,Y,Z
COMMON /FOO/ X
```
COMMON /FOO/ Y
COMMON /FOO/ Z

We have chosen to implement a common block declaration as a list of declarations of its members. If an existing common block is encountered in the source file, the list is “opened” and the subsequent members of the common block are compared, or an error is signaled if we “run out of members” on the list. When a subprogram ends, the list is “closed” and, if the last member has not been reached, an error is generated.

14.3 Declaration in the output

Not all declarations that appear in the source code will also be in the output. The declarations of common block members will have to be removed, since they will no longer be necessary. This means that the output cannot be generated line by line, as we read the source. First, we need to have all of the specification statements before we can generate any output. Conceptually, we would have to collect all declarations and separate them in two groups: those that describe common block members and those that do not. After that all declarations from the latter group would have to be regenerated to appear in the output. Since we had already processed all declarations at once, we initially planned to do the programmer a little favor. It is known that implicit declarations can cause problems. A common extension, namely an IMPLICIT NONE statement has been officially introduced in Fortran 90 [65]. During the regeneration stage we wanted to provide full declarations of all encountered entities. For example,

\[
\text{DIMENSION A(10)}
\]
\[
A(1) = 1.0
\]

would become

\[
\text{REAL A}
\]
DIMENSION A(10)
A(1) = 1.0

or simply

REAL A(10)
A(1) = 1.0

Since all type information would be explicitly provided, this would eliminate the need for any IMPLICIT statements. It turns out that not only the specification statements are needed to carry this conversion out. A Fortran 77 program

```
IMPLICIT INTEGER(X)
X = 1
END
```

has only an IMPLICIT specification statement, but it is still lacking a declaration that, according to our plan, should be provided for X. Therefore we should process an entire body of a subprogram before any output can be generated.

Not all declarations in Fortran are independent from each other. Obviously, all PARAMETER statements should precede all other declarations, since constants defined in PARAMETER statements can be used in array declarations. It seems like the following, approximate statement order in the output may be assumed:

1. PARAMETER statements,
2. variable and array declarations, in no particular order,
3. common block statements (we need those to pass the object indices around)
4. executable statements, in the order in which they appear in the source
However, this turns out to be too simplistic and there are situations in which this outline would not produce valid code. First of all, PARAMETER statements can depend on each other. A statement \texttt{PARAMETER (M=N)} is perfectly valid, as long as it is preceded by a \texttt{PARAMETER} statement for \texttt{N}. Since initially all constant expressions were to be preserved, we would have to keep track of the dependencies between parameter values. The second problem is that parameters also have types. Therefore, if the implied type of a parameter does not match its value, an explicit type statement has to be provided and it has to precede the \texttt{PARAMETER} statement. This already violates our ordering. The third problem was a consequence of Fortran's \textit{adjustable arrays}. An adjustable array is an array passed as a argument to a subroutine or a function, size of which is not statically declared within that subroutine or function, but passed as another argument. For example:

\begin{verbatim}
SUBROUTINE S(A,N)
  INTEGER N
  REAL A(N)
C ...
END
\end{verbatim}

In the code above, \texttt{A} is an adjustable array and its size must be passed as the second argument \texttt{N}. Although the standard does not seem to precisely state that, some compilers require that the declaration of the argument specifying the length come before the declaration of the array, unless the implied type of it is \texttt{INTEGER}. In the example above, the implied type of \texttt{N} is \texttt{INTEGER}, so it could either follow the declaration of \texttt{A} or not occur at all. This particular case did not turn out to be a big problem, though. Only dummy argument arrays can be adjustable, and dummy arguments cannot be in a common block. Hence, neither the array nor the argument specifying its length can be legally placed in a common block. This means that they will never be removed from the source and thus they can be reproduced in the same order in which they were placed in
the source code. This did not immediately solve all problems, but it at least eliminated
the necessity for analyzing inter-declaration dependencies in that particular case.

14.4 Special declarations in the input

There were some other “interesting” cases of declarations to be analyzed. One of
these was ENTRY statements and the declarations associated with them. ENTRY state­
ments can specify additional entry points into a subprogram. They can have a set of
dummy arguments and they can share names with the dummy arguments of the en­
closing subprogram. They can also specify return values of a different type than the
enclosing function. An ENTRY statement can appear almost anywhere within the body
of the program unit. It must maintain its relative order with respect to executable
statements, but type statements for its dummy arguments can precede it. For example

```fortran
SUBROUTINE S
  INTEGER N
  REAL A(N)
  ENTRY(A,N)
  C ...
END
```

is valid.\(^2\) This shows one more problem: we cannot always tell that a given type state­
ment refers to a dummy argument when we see it. Therefore, the declaration REAL(N)
cannot be rejected at the point at which it occurs. Assume that the programmer has
added a COMMON statement right after the declaration of N:

```fortran
SUBROUTINE S
  INTEGER N
```

\(^2\)Actually, the Fortran 77 standard does not address this situation, but, since it does not forbid it,
it is assumed that it is indeed allowed.
COMMON /FOO/ N
REAL A(N)
ENTRY(A,N)
C   ...
END

In this case, since N was declared a common block member, it cannot be a dummy argument. Thus, the following statement REAL A(N) was invalidated.

Fortran 77 provides a special feature to ease the use of short but frequent expressions: statement functions. Statement functions are defined via statement function statements, whose syntax is at least a bit confusing from the point of view of parsing. Consider the following code:

```
SUBROUTINE S
REAL FOO(3,3)
INTEGER I,J
FOO(I,J) = REAL(I+J)
END
```

The statement in line 3 is definitely an assignment to the array FOO. It does not matter that neither I and J are undefined — it is an erroneous program with a semantically incorrect assignment. Suppose we remove the first line with the declaration of the array FOO. Now, given just

```
SUBROUTINE S
INTEGER I,J
FOO(I,J) = REAL(I+J)
END
```

the last line has changed its meaning without changing its form at all. There are some other minor issues in the treatment of ill-formed statement functions:
SUBROUTINE S

FOO(I,I) = I+1

END

Is this an assignment to an undeclared array or a malformed statement function statement? Such questions are important to be answered in order to generate meaningful error messages. We have decided to treat such statements as potential statement function statements, unless the outermost symbol on the right hand side had been declared as an array.

Dummy arguments in statement function statements also have their own trail of issues: a dummy argument in a function statement can only be a variable—arrays are not allowed. Their scope is limited to the expression defining the statement function and there is no good way to declare their types. The name of a dummy argument in a statement function statement can appear in a separate type statement but, according to the standard, such type statement would declare a local variable which would be a distinct object. Since IMPLICIT NONE is not a part of the language standard, the question remains what consequences it would have on the types of the dummy arguments in a statement function. Most compilers seem to require an explicit type specification for dummy arguments in such cases after all, which raises questions about the relationship between the local variables and dummy arguments that those compilers assume.

Some issues seem to be either vaguely worded in the standard or not mentioned at all. Several implementations, most likely due to some legacy issues, do not fully conform to the standard, there are also some situations to which different compilers react differently. Since we had no foundation for some decisions that would have to be made, especially regarding language issues not common enough to have uniform treatment, we decided to only diagnose situations that directly interfere with the translation process. Many problems are detected at the later stages, when they would prevent the current part of
the processing scheme from completing successfully.
15 BLOCK DATA

The parts of a Fortran 77 program designated by BLOCK DATA statements require special processing. This chapter describes how the BLOCK DATA statements are handled by the translator to allow them to undergo the final object substitution, described later in chapter 18.

15.1 Common block initialization

A BLOCK DATA is a separate program unit. The purpose of its existence is to provide initial values to the members of named common blocks. Members of such common blocks cannot be initialized anywhere else. Members of the blank (unnamed) common block cannot be given "compile-time" initial values at all. A BLOCK DATA is not an executable subprogram and therefore it cannot contain any executable statements. Therefore no assignments are allowed in a BLOCK DATA — all initialization has to occur via a DATA statement. Since the replacement objects for the common block members will be allocated at the beginning of the program execution, the initialization must also occur during the run time. Therefore the BLOCK DATA subprograms must be converted into subroutines that perform the same initialization which would otherwise take place as a result of DATA statements previously placed therein.

15.2 Formal syntax of DATA statements

The precise syntax of a DATA statement can be expressed as follows:
DATA datavarlist / vallist /

Both, datavarlist\(^1\) and vallist are comma-separated lists. The datavarlist is a list of objects to be initialized and vallist is a list of the initial values. Each item on that list is either a single constant value or a constant value with a repetition specification. The repetition specification is of form count*val, where count specifies number of times that the value val is to be repeated. The repetition specification is nothing more than just a shorthand notation.

The more complicated part of the syntax is the datavarlist. As mentioned above, it is a comma-separated list of certain items, that we will call datavars. The formal syntax of a datavar is what makes processing of DATA statements interesting.

\[
\begin{align*}
\text{datavar} & \rightarrow \text{lhs} \\
\text{datavar} & \rightarrow ( \text{datavarlist} , \text{dospec} ) \\
\text{dospec} & \rightarrow \text{name} = \text{expr} , \text{expr} \\
\text{dospec} & \rightarrow \text{name} = \text{expr} , \text{expr} , \text{expr}
\end{align*}
\]

The symbol \text{lhs} stands for “left hand side” and originally referred to a term that can appear on the left hand side of an assignment. Essentially, it is a reference to a memory location, i.e. a variable, an array element\(^2\) or an array itself. The “\text{DO} specification”, mentioned above as dospec, plays the same role as an analogous construction in DO loops. It specifies an iteration variable, its beginning, ending values, and an optional stride, in that order. The second alternative for datavar is nothing other than an implied-DO list and it shows the recursive nature of the datavarlist. It can be easily seen that the implied-DO lists can be nested. The beginning, ending and the stride in the nested implied-DO lists can refer to the iteration variables of the enclosing implied-DO lists. Obviously,

\(^1\)The names datavarlist, vallist, etc. are taken from the initial Yacc grammar file supplied with f2c.
\(^2\)Both possibly with an optional subscript specification.
the iteration variables must not be the same as those in the enclosing implied-DO lists. Potential expressions occurring in an \textit{lhs}, such as array indices or subscript limits must be constant, optionally involving the iteration variables of the enclosing implied-DO lists. Let us see some examples:

1. Have two variables $X$ and $Y$ of type \texttt{REAL}. Initialize $X$ with $-1.06$ and $Y$ with $2.34$:

   \begin{verbatim}
   DATA X,Y /-1.06/
   \end{verbatim}

2. Assume a declaration \texttt{REAL X,Y,Z,A(5)}. Initialize elements $X$, $Y$, $Z$, and the first two elements of $A$ with $17.1$. The remaining three elements should be initialized with $51.4$, $51.5$ and $51.6$ respectively:

   \begin{verbatim}
   DATA X,Y,Z,(A(I),I=1,5) /17.1, 51.4, 51.5, 51.6/
   \end{verbatim}

3. Initialize the upper triangular section of the matrix $A$ with $1$:

   \begin{verbatim}
   PARAMETER (N=10,N2=N*(N+1)/2)
   INTEGER A(N,N)
   DATA ((A(J,I),I=J,N),J=1,N) /N2*1/
   \end{verbatim}

\section{15.3 First attempts}

As we have already stated, we need to convert this compile-time initialization into an executable code. Before we can successfully tackle the most general case, let's consider smaller examples. First of all, it is obvious that the main culprits here are the implied-DO lists. Assume temporarily that they are not allowed here, so that our restricted \texttt{datavarlist} can only consist of \textit{lhs}s. Let's have a look at the example 1 above. There are only two objects and two values. The simplest way to deal with it would be to replace them with two assignments. Actually, assignments are about the only means that we
have, the real question is how they should be generated. The simple example can be solved with just two separate assignment statements, but is it sufficient to work in other cases too? Consider a slightly more complex example:

```
PARAMETER (N=3)
REAL X,Y,Z
DATA X,Y,Z /N*0.0/
```

The problem here is that in our initial approach, the value of $N$ was unknown. Therefore we could not determine the length of the `vallist` at the time of translation. Of course, one could argue that there are three variables on the first list and so it can be deduced that the value of $N$ should be 3. Therefore the output code would contain

```
X = 0.0
Y = 0.0
Z = 0.0
```

This is wrong for two reasons. First of all, it is an error for the two lists to be of different lengths. If $N$ was indeed 4, we would hide the error by just using the first 3 values. If, on the other hand, $N$ was 1, then the last 2 assignments would not be warranted in any way. Actually, things can get much worse:

```
PARAMETER (N=1,M=2)
REAL A(2),X
DATA A(1),A(2),X /N*-1.0, M*1.0/
```

How many elements are assigned $-1.0$ and how many are assigned $1.0$? This question cannot be answered without knowing the actual values of $N$ and $M$; and the information that there are exactly 5 elements to be initialized does not help at all. We needed something other than just a fixed number of assignments, something that would allow us to incorporate the symbolic parameters into the process. This is exactly the reason
for which loops were invented, so a DO loop seemed to be our way out of trouble. We had two choices as to what the loop should iterate over: either the \textit{datavars} or the \textit{vals}. To see if one of these options was better than the other, we had to apply them to different cases and compare the results. In the example above the second alternative seems to work and it appears to be much simpler to implement, if we ever want to consider it in the general situation. The output could look like this:

\begin{verbatim}
C initialize using first N values
  DO 10 I=1,N
    IF (I.EQ.1) THEN
      A(1) = -1.0
    ELSE IF (I.EQ.2) THEN
      A(2) = -1.0
    ELSE IF (I.EQ.3) THEN
      X = -1.0
    ENDIF
  10 CONTINUE
C initialize using next M values
  DO 11 I=N+1,N+M
    IF (I.EQ.1) THEN
      A(1) = 1.0
    ELSE IF (I.EQ.2) THEN
      A(2) = 1.0
    ELSE IF (I.EQ.3) THEN
      X = 1.0
    ENDIF
  11 CONTINUE
\end{verbatim}
The massive conditional statement in the body of these loops is the unfortunate consequence of not knowing the values of N and M. The variable I varies from 1 to N+M and depending on its actual value we have to choose the right element to be initialized. Since we are considering the case in which implied-DO lists are not allowed, the number of initialized objects is known and the I\textsuperscript{th} object can be statically determined. Instead of rewriting the same code with the loops iterating over the data\textit{varlist}, let's proceed to another example. This time, let us slowly introduce implied-DO lists back:

```fortran
PARAMETER (N=10)
INTEGER A(N)
DATA (A(I),I=1,N) /N*0/
```

In the previous case, our loop iterated over the list of values. It worked fine, since we were able to statically determine the I\textsuperscript{th} element on the list of objects. In case of implied-DO lists this is no longer possible and so, even in this simple example we have to resort to the other alternative: iteration over the data\textit{varlist}:

```fortran
PARAMETER (N=10)
INTEGER A(10)
DO 10 I=1,N
   IF (I.GE.1 .AND. I.LT.N) THEN
      A(I) = 0
   ENDIF
10 CONTINUE
```

We have only considered a case with only an implied-DO list, but, since it looks promising, we can start thinking about how to generalize it to encompass both individual elements and implied-DO lists. Consider a more complicated case:

```fortran
PARAMETER (N=10, M=5, NM=N+M)
```
PARAMETER (K=7,L=NM+3-K)
INTEGER A(N+M)
DATA X,Y,(A(I),I=1,NM),Z/K*-1,L*1/

The values of K and L do not have any deep meaning besides being chosen to add up to the length of the array A, plus 3 additional elements: X, Y and Z. They were introduced to eliminate any direct correspondence between the structures of the list of objects and the list of values. The way to handle a case like this is to break down the datavarlist into individual items, either variables, or array references, or implied-DO lists and treat them separately. The \( n^{\text{th}} \) item would then correspond to some \( k^{\text{th}} \) element from the list of values (where \( k \geq n \)). We would need to have a convenient way to refer to the \( k^{\text{th}} \) value. The simplest way seems to be to have a counter, initialized at the beginning and incremented each time an assignment is generated. The following expected result of the transformation of the example above should illustrate the idea:

C \hspace{1em} \textit{Let IV be the counter used to pick values}

IV = 0

C \hspace{1em} \textit{initialize the first two objects: X and Y}

C

IV = IV+1

C \hspace{1em} \textit{determine the value from the list}

IF (IV.GE.1 .AND. IV.LE.K) THEN
  X = -1
ELSE IF (IV.GT.K .AND. IV.LE.(K+L)) THEN
  X = 1
ENDIF

C
IV = IV+1

IF (IV.GE.1 .AND. IV.LE.K) THEN
    Y = -1
ELSE IF (IV.GT.K .AND. IV.LE.(K+L)) THEN
    Y = 1
ENDIF

C process the implied-DO loop

C DO 10 I=1,N
IV = IV+1
IF (IV.GE.1 .AND. IV.LE.N) THEN
    A(I) = -1
ELSE IF (IV.GT.K .AND. IV.LE.(K+L)) THEN
    A(I) = 1
ENDIF
10 CONTINUE

C process the remaining element Z

C IV = IV+1
IF (IV.GE.1 .AND. IV.LE.K) THEN
    Z = -1
ELSE IF (IV.GT.K .AND. IV.LE.(K+L)) THEN
    Z = 1
ENDIF
Notice, that the value selection looks almost identical in all the places in which it occurs. The only difference is the data object being assigned a value. Assume that the value list is specified as follows:

\[
/ N_1*V_1, N_2*V_2, \ldots, N_k*V_k / 
\]

for some \( k \). Given that, let’s define a SELECT-ASSIGN template:

\[
\text{SELECT-ASSIGN}(\text{var}, \ iv) = \\
iv = iv + 1 \\
\text{IF (iv.GE.1 .AND. iv.LE.N_1) THEN} \\
\quad \text{var} = V_1 \\
\text{ELSE IF (iv.GT.N_1 .AND. iv.LE.(N_1+N_2)) THEN} \\
\quad \text{var} = V_2 \\
\text{ELSE IF (iv.GT.(N_1+N_2) .AND. iv.LE.(N_1+N_2+N_3)) THEN} \\
\quad \text{...} \\
\text{ELSE IF (iv.GT.(N_1+...+N_{k-1}) .AND. iv.LE.(N_1+...+N_k)) THEN} \\
\quad \text{var} = V_k \\
\text{ENDIF} 
\]

The code above is now more concise and has a more regular shape:

\begin{verbatim}
C Let IV be the counter used to pick values
IV = 0
C initialize the first two objects: X and Y
SELECT-ASSIGN (X, IV)
SELECT-ASSIGN (Y, IV)
C process the implied-DO loop
\end{verbatim}
DO 10 I=1,N
    SELECT-ASSIGN (A(I), IV)
10 CONTINUE

C
C process the remaining element Z
SELECT-ASSIGN (Z, IV)

Armed with this tool, we can try to attack more complicated cases, in particular, nested implied-DO lists. Assume we have the following datavar:

\(((A(I_1,I_2), B(I_1,I_2), I_1=b_1,e_1,s_1), C(I_2), I_2=b_2,e_2,s_2)\)

Once again, our approach is to iterate over the datavarlist and generate the output for each of the datavar on the list. In case of an implied-DO list, we would recursively process the datavarlist contained therein:

IV = 0

C process the outermost implied-DO list
DO 12 I2=b_2,e_2,s_2

C (a) process the nested implied-DO list
DO 11 I1=b_1,e_1,s_1

C descend into the datavarlist and process the two objects there
SELECT-ASSIGN (A(I1,I2), IV)
SELECT-ASSIGN (B(I1,I2), IV)

11 CONTINUE

(b) process the lhs C(I2)
SELECT-ASSIGN (C(I2), IV)

12 CONTINUE

It should be intuitively clear how this algorithm works. It still did not solve the problem of the invalid number of initializing values. We could not find satisfactory solution to
this problem. One unsatisfactory solution was to leave the original DATA statement and the declarations of the objects being initialized. These objects would not be used in the rest of the code, since all references to them would be replaced later in the process and so any initialization of them would cause no harm. In case of the erroneous situation we are discussing, the compiler would be able to diagnose it in those "leftover" DATA statements. Because this increases memory usage and definitely makes the code more confusing, we have decided to abandon this idea. As far as the length mismatch problem was concerned, we have agreed that not diagnosing it is a small price to pay; it has become one more limitation of our translator. At this point, the cost of verification of this constraint would exceed the benefits of improved functionality.

15.4 The final answer

Since none of the algorithms described above were actually implemented before we decided to change our translation scheme, we have decided to take advantage of some opportunities that were now open. In this particular situation, the greatest gain for us was that we could evaluate all expressions involved in the implied-DO lists, including control variable limits and array indices. This could potentially simplify the translation process. The significant difference was, not surprisingly, the treatment of the implied-DO lists. Let us take yet another look at an implied-DO list. What it actually does is specify some elements in a particular order. Most often those elements are elements of some array, usually they are in the same order in which they are stored in memory. If all of the expressions used to specify an implied-DO list are of known values, it should be possible to recreate the sequence of the elements that a given implied-DO list describes. In other words, given an implied-DO list, it would be possible to expand it into the corresponding sequence of variables and array elements. Once such an expansion has been made, we could simply iterate over the sequence and, for each element on it,
generate an appropriate assignment. This was the idea that was finally implemented in the translator.

The expansion algorithm is not very complicated. It converts the `datavarlist` contained in a DATA statement into a list of `lhs` by applying an expansion routine to each of the elements of the original `datavarlist` and merging all resulting lists into one.

```scheme
EXPAND-DATAVARLIST (dvs):
    return join [ EXPAND-DATAVAR (dv) | dv ∈ dvs ]
```

We now have to define EXPAND-DATAVAR. As we have seen, the argument it takes, a `datavar`, can have two different forms: an `lhs` and an `impdo`. Since an `lhs` does not need any further expansion, the EXPAND-DATAVAR does not do much:

```scheme
EXPAND-DATAVAR (lhs):
    return [ lhs ]
```

The last case left is the definition of EXPAND-DATAVAR for an `impdo`. Assume the `impdo` has the following form:

```plaintext
impdo = (dvs, name = beg, end, str)
```

Then,

```scheme
EXPAND-DATAVAR (impdo):
    for i = beg to end step str
        dvs' ← [ SUBST-NAME (dv, name, i) | dv ∈ dvs ]
    return EXPAND-DATAVARLIST (dvs')
```

The function `SUBST-NAME` substitutes a given `name` with the value `i` in a given `datavar`. The substitution is "deep," that is, all occurrences of `name` are replaced, including those in the nested `datavars`, if there are any. The expressions `beg`, `end` and `str` in `impdos` are also subject to substitution.
EQUIVALENCE statements can have a significant influence on the data objects that a given program defines. It is thus necessary for the translator to properly recognize those statements and subject them to an appropriate treatment. The rest of this chapter describes the way in which EQUIVALENCE statements are handled.

16.1 Formal introduction

As described in section 12.10, EQUIVALENCE statements are used to associate a group of data objects. Such an association is specified by the programmer by providing lists of elements to be associated. We will call those lists equivalence lists. Suppose that we have three REAL variables, X, Y and Z and we want to make them all refer to the same object. The simplest way would be:

\[
\text{EQUIVALENCE (X,Y,Z)}
\]

Equivalence lists in subsequent EQUIVALENCE statements do not have to be disjoint. The example above could be rewritten as

\[
\text{EQUIVALENCE (X,Y), (X,Z)}
\]

or

\[
\begin{align*}
\text{EQUIVALENCE (X,Y) } \\
\text{EQUIVALENCE (Y,Z)}
\end{align*}
\]
One way to look at the EQUIVALENCE statement is that it establishes an equivalence relation $E$ between elements of some set. The equivalence lists provided by the programmer define a reflexive and symmetric relation $T$:

$$x T y \quad \text{if } x \text{ and } y \text{ belong to the same equivalence list}$$

The relation $E$ can now be defined as the reflexive-transitive closure of $T$.

16.2 The idea

In the translation process, members of common blocks will be replaced by completely different objects, provided by GPSHMEM. The EQUIVALENCE statement can provide other synonyms by which the objects previously declared as members of common blocks can be accessed. Thus, it is necessary to take the equivalence classes into account during the object substitution phase. Obviously, it is only required if a common block member is listed on an equivalence list. As usual, let's begin with a simple example:

```fortran
REAL X, Y
COMMON /FOO/ X
EQUIVALENCE (X,Y)
Y = 1.0
```

Thanks to the EQUIVALENCE statement, the assignment on the last line is functionally identical with an assignment $X = 1.0$. Every time the variable $Y$ is used, the variable $X$ can be placed instead. If all occurrences of $Y$ were rewritten in such way, we would no longer need the EQUIVALENCE statement. The first idea would be then to take all elements in the equivalence class of some common block member $T$ and replace their occurrences in the source code with $T$. The following will illustrate the idea.

```fortran
REAL X
```
COMMON /FOO/ X

X = 1.0

Notice that the variable Y has been removed completely. There is no Y in the declaration and the EQUIVALENCE statement is gone. Everything went well in this case, but there is one small problem. Consider a slightly modified version of the above example:

COMPLEX X
REAL Y
COMMON /FOO/ X
EQUIVALENCE (X,Y)
Y = 1.0

Despite the difference in the types of associated elements, this is still a legal Fortran 77 code. The language standard defines the storage layout of type COMPLEX as two consecutive units of type REAL: the real and the imaginary part respectively. The assignment Y = 1.0 is thus an assignment to the real part of X. This short example is enough to demonstrate some shortcomings of the initial approach. If just a straightforward substitution of Y with X was performed, we would end up with an assignment X = 1.0. While this assignment is still valid (the assigned value is actually CMPLX(1.0), as required by section 10.1 of the Fortran 77 standard), it becomes visible that the association of objects of different types should be handled with care. Ignoring the type differences can lead to problems:

COMPLEX X
REAL R(2)
COMMON /FOO/ X
EQUIVALENCE (X,R)
R(2) = 2.0
Because of the EQUIVALENCE statement, the elements of \( R \) are the real and imaginary parts of \( X \). The last line assignment is thus an assignment to the imaginary part of \( X \).

First of all, the textual replacement fails miserably, since neither \( X = 2.0 \), nor \( X(2) = 2.0 \) make any sense. Some other code would have to be generated, for example, \( X = (\text{REAL}(X), 2.0) \). Obviously there are situations that cannot be handled at all. This happens if an EQUIVALENCE statement associates objects of unrelated types, for example:

\[
\text{DOUBLE PRECISION A}
\]
\[
\text{COMPLEX C}
\]
\[
\text{EQUIVALENCE (A,C)}
\]
\[
\text{COMMON /FOO/ C}
\]
\[
A = 1.0\text{DO}
\]

In such situations, no arithmetic conversion is made when the assignment in the last line takes place. The variable \( C \) contains whatever happens to be the representation of the number 1 of type DOUBLE PRECISION. It may even happen not to be a valid COMPLEX number at all. There are only a few circumstances in which explicit association between elements of different types is guaranteed to produce well-defined results, and even those situations require special attention. Thus, we have decided to disallow the use of the EQUIVALENCE statement with elements of different types.

16.3 "Potentially equivalent" elements

It should be clear that not all associations are valid. In particular, the following cases are excluded:

\[
\text{EQUIVALENCE (A(1), A(2))}
\]

or

\[
\text{EQUIVALENCE (A(2), B(1)), (B(2), C(1)), (A(1), C(2))}
\]
The first example looks obvious, since it is trying to directly associate two elements of the same array. The last example shows that it is not necessary for an array to be referenced twice in one equivalence list to actually cause a conflict. The situation becomes much more complicated when we cannot determine the exact relationship between two given elements. That will happen if symbolic constants are used:

\[ \text{EQUIVALENCE (A(N), B(M)), (A(K), B(L))} \]

Some valuations of \( K, L, M \) and \( N \) will be valid, whereas some others will not. Without knowing the exact values of these parameters, we have no way to establish validity of such statements. We had two ways out of this problem: either reject all \text{EQUIVALENCE} statements, or try to impose some additional restrictions on the associations that would allow only those associations that we would be able to deal with. We have decided to take the second approach. First, we had to establish what associations we would deem acceptable. When we were trying to generate maintainable code, the foundation of our approach was to allow the programmer to manipulate all the symbolic constants in the generated code in exactly the same way as it would be done in the original code. Keeping that in mind, we wanted to allow those associations that would remain valid regardless of the actual values of any parameters. Therefore, we have chosen to \textit{accept only those \text{EQUIVALENCE} statements that, for all valuations of the parameters used in them, yield valid associations.}

Section 8.2.5 of the standard provides restrictions that any \text{EQUIVALENCE} statement must meet:

1. it must not specify that the same storage unit is to occur more than once in a storage sequence,

2. it must not specify that consecutive storage units are to be nonconsecutive.

Let us analyze ways in which those constraints can be violated. How can the same storage unit be requested to appear in two different locations? If only variables and
elements of different arrays are given on the equivalence lists, all of the storage units they specify belong to different data objects (that is to the variables themselves or to the arrays, in the case of array elements). Since all of those objects are distinct, there is no way that the first condition can be violated. The only situation that can possibly cause a conflict with it is when elements of the same array appear more than once in an equivalence list. The second condition is a bit clearer. The only kind of object that can have its different storage units referred to directly is an array (other objects with more than one storage unit are variables of type DOUBLE PRECISION, COMPLEX or INTEGER*8, but in all of these cases we can only refer to the first storage unit so the following ones are always implicitly contiguous.) In other words, the second condition can only be violated in the same circumstances as the first one—by having multiple elements of the same array (including repetitions of possibly the same element) appear on an equivalence list. To better capture the idea of elements of the same array we have introduced a concept of potentially equivalent elements. Intuitively, two elements are potentially equivalent if they are elements of the same array, whose indices are given as expressions that involve symbolic constants. The term “potentially equivalent” comes from the fact that, depending on the values of the parameters, they will or will not denote the same element of a given array.\footnote{The name “potentially identical” or “potentially equal” would probably be more accurate.} Since we did not evaluate arithmetic expressions at the translation time at all, even if they did not involve symbolic constants, in the course of our analysis efforts we have extended our definition of potentially equivalent elements to elements of the same array whose indices were given by different expressions. The following examples should illustrate the concept of potentially equivalent elements:

1. Obvious case, both elements are parameterized: \( A(N) \), \( A(M) \).

2. Only one element is parameterized: \( A(1) \), \( A(N) \).

3. No elements use symbolic constants, and both denote exactly the same array el-
ement, but they are expressed by different terms: \( A(1+2) \), \( A(2+1) \). A simpler example is \( A(1) \) and \( A(1+0) \).

The last example illustrates some of the consequences of not evaluating constant expressions. From our point of view, expressions \( 1+2 \) and \( 2+1 \) are different, since they have different structures. Expressions in our translator were always represented by syntax trees and we compared expressions by comparing their trees. To simplify the code we did not take advantage of any special properties of arithmetic operators, such as commutativity. We can now present a formal definition of potentially equivalent elements. To capture the properties of tree comparison, first we need to define equivalent expressions:

**Definition 16.3.1 (equivalent expressions).** Define the equivalence between two expression terms as follows:

1. Any two identical literals are equivalent. Any two identical symbolic constants are equivalent.

2. \( E \) is equivalent to \( (E) \).

3. If \( E_i \) and \( F_i \) are equivalent for each \( 1 \leq i \leq n \) and \( f \) is a name of an \( n \)-ary function or operator, then \( f(E_1, \ldots, E_n) \) and \( f(F_1, \ldots, F_n) \) are equivalent. This also applies in infix notation, i.e. \( E_1 \ f \ F_1 \) and \( E_2 \ f \ F_2 \) are equivalent, if \( E_i \) and \( F_i \) for \( i = 1, 2 \) are equivalent.

The above definition will allow us to precisely describe what terms we consider identical.

**Definition 16.3.2.** Let \( x \) and \( y \) be terms denoting variables or array elements. We will consider \( x \) and \( y \) to be identical if either

1. they both denote the same variable, or
2. there exists an n-dimensional array A such that \( x = A(x_1, \ldots, x_n) \) and \( y = A(y_1, \ldots, y_n) \) for some expression terms \( x_i \) and \( y_i \) for \( 1 \leq i \leq n \), and \( x_i \) is equivalent to \( y_i \) for each \( 1 \leq i \leq n \).

Given that, let us now define a relation \( \mathcal{P} \mathcal{E} \) that would be the expression of the potential equivalence in mathematical terms:

**Definition 16.3.3 (\( \mathcal{P} \mathcal{E} \) relation).** Let \( x \) and \( y \) be terms denoting data objects (variables or array elements). We say that \( x \mathcal{P} \mathcal{E} y \) if either

1. both \( x \) and \( y \) are identical with respect to definition 16.3.2, or

2. there exists an n-dimensional array \( A \) such that \( x = A(x_1, \ldots, x_n) \) and \( y = A(y_1, \ldots, y_n) \) for some expression terms \( x_i \) and \( y_i \) for \( 1 \leq i \leq n \).

The definition of the relation \( \mathcal{P} \mathcal{E} \) has a small problem: it classifies elements \( A(1) \) and \( A(2) \) as potentially equivalent, even though they are obviously different. We have tried to overcome this issue, but we have finally accepted it as a limitation of our approach. Actually, it has a somewhat convenient consequence:

**Lemma 16.3.1.** The relation \( \mathcal{P} \mathcal{E} \) is an equivalence relation.

For the set of all EQUIVALENCE statements specified in a given program unit, create a graph \( \mathcal{G} \). The nodes of the graph are all of the objects appearing on the equivalence lists, objects identical with respect to definition 16.3.2 are represented by the same node. There are two kinds of edges in graph \( \mathcal{G} \), both of them can connect two given nodes at the same time: \( \mathcal{E} \)-edges and \( \mathcal{P} \mathcal{E} \)-edges. For nodes \( x \) and \( y \),

1. there is an \( \mathcal{E} \)-edge between \( x \) and \( y \) if \( x \) and \( y \) are not identical and \( x \mathcal{E} y \),

2. there is a \( \mathcal{P} \mathcal{E} \)-edge between \( x \) and \( y \) if \( x \) and \( y \) are not identical and \( x \mathcal{P} \mathcal{E} y \).

Armed with all those definitions, we can now express the condition that we required the EQUIVALENCE statements to meet in order to be accepted.
Definition 16.3.4. The set of EQUIVALENCE statements is acceptable, if the corresponding graph $G$ contains no cycles consisting of nodes of different kinds.

In other words, we require the graph to either contain no cycles at all, or to only have cycles consisting of nodes of the same type. We assume that all cycles considered are proper, that is, the same node does not appear on the cycle more than once. It is worth pointing out that two nodes with both kinds of edges between them would constitute a cycle. Existence of such a cycle would, of course, render $G$ unacceptable.

Suppose, once again, that a given set of EQUIVALENCE statements violates the constraints imposed by the standard. The first condition is that the association "must not specify that the same storage unit is to occur more than once in a storage sequence". This can only happen if there is an object $t$ such that there are paths, possibly empty, consisting of $E$-edges from $t$ to two distinct nodes $x$ and $y$ connected by a $PE$-edge. In other words, in order to have a node $t$ "appear" in two different places at the same time, we need to associate it with two different elements of the same array. The association may be indirect. It needs to be noted that the paths from $t$ to $x$ and $y$ cannot both be empty, since $x$ and $y$ are different. In these circumstances we can construct a cycle that contains different kinds of nodes, so such a situation will be rejected.

The second condition requires that the association "must not specify that consecutive storage units are to be nonconsecutive." This happens when two different elements of the same array are associated with the same object, or with two elements of another array whose relative distance is different than the relative distance between the elements of the original array. Let $x$ and $y$ be nodes corresponding to the original elements. There is a $PE$-edge between them. Now, there must be two paths, consisting of $E$-edges, that connect $x$ and $y$ with two other elements $x'$ and $y'$. There are two possibilities:

1. $x'$ and $y'$ are the same node, or

2. there is a $PE$-edge between $x'$ and $y'$.
In both cases, it is possible to construct a cycle that contains both kinds of edges, thus violating our acceptance criterion.

This reasoning shows that our condition is strong enough to eliminate all invalid cases of \texttt{EQUIVALENCE} statements. One could ask if it is weak enough not to reject all allowed situations. The answer is no. A simple example of a valid \texttt{EQUIVALENCE} that would get rejected can be

\begin{verbatim}
REAL A(10), B(10)
EQUIVALENCE (A(1),B(1)), (A(2),B(2))
\end{verbatim}

There are \(P\varepsilon\)-edges between \(A(1)\) and \(A(2)\) and also between \(B(1)\) and \(B(2)\). There are also explicit \(\varepsilon\)-edges between \(A(1)\) and \(B(1)\), and between \(A(2)\) and \(B(2)\).

We believe that this would still allow enough legitimate \texttt{EQUIVALENCE} statements. It is definitely better than disallowing them all. Our planned implementation had several other limitations:

1. If \(A\) is an array indexed from \(n\), then \(A\) and \(A(n)\) are synonymous on an equivalence list. Initially, we did not perform conversion from the name of the array to the first element of it in such circumstances. Therefore \texttt{EQUIVALENCE (A(1),A)} would get rejected.

2. If \(A\) and \(B\) are both arrays of the same type, but with different dimensions, a substitution usually cannot be performed.

\begin{verbatim}
REAL A(2,3), B(5)
EQUIVALENCE (A(2,1),B)
COMMON /FOO/ A
WRITE (*,*) (B(I),I=1,5)
\end{verbatim}

A substitution of \(B(1)\) with an appropriate term of \(A(i,j)\) is possible, but is much more involved.
3. Common block members were not allowed to be associated with elements that had a neighbor in the graph $G$ connected with them with a $PE$-edge. If such a situation was detected, a rather meaningless error message was displayed: “EQUIVALENCE class with a COMMON element has a PE-neighbor.”

When our approach to the whole translation process changed, our equivalence validation algorithm also changed. Exactly how the problems listed above, except for the last one, are solved will be explained in chapter 18.

Let us now present the final algorithm implemented in the translator. It assumes the existence of a list $eqll$ of all equivalence lists specified in the source code:

```
SUBST-EQUIV (eqll):
    search eqll for eql containing some common block member $c$
    if not found, STOP
    if not VALID (eql), ERROR
    remove eql from eqll
    for each lhs $\in$ eqll
        if lhs $\neq$ c, SUBST-EQUIV-LHS (lhs, c)
```

The function VALID verifies a given equivalence list. It is defined as follows:

```
VALID (eql):
    if eql contains more than one common block member, or
    eql contains elements of different types, or
    eql contains two or more elements in the relation $PE$,
    return NO
    otherwise
    return YES
```

Equivalence lists are lists of objects. A term that refers to an object that appears on these lists is called an $lhs$ in the grammar. A more detailed description of $lhs$ can be
found in section 15.2. The last function, \texttt{SUBST-EQUIV-LHS}, takes two \texttt{lhs} objects, one of them referring to the object being replaced and the other referring to the object that will be put in the place of the first one. In our application, the first object is an object associated with some common block member and the second object is the common block member itself. The function \texttt{SUBST-EQUIV-LHS} will then search the source code for references to the first object and perform appropriate substitutions. The function \texttt{REWRITE-LHS} will be defined later, in chapter 18.

\begin{verbatim}
SUBST-EQUIV-LHS (lhs, c):
    for each lhs' in the source code
        if lhs and lhs' refer to the same object,
            replace lhs' with REWRITE-LHS (lhs', lhs, c)
\end{verbatim}
We will now describe how the translator processes Fortran's INCLUDE statements. The INCLUDE statement is not a part of the official Fortran 77 language, that is, it is not defined by the standard. It is, however, implemented by most Fortran compilers, since it provides a very convenient facility to extract a commonly used set of definitions and store it in a single file and it has become an official part of the Fortran 90 language. It works very similar to the #include directive in a C or C++ program: it reads a file, whose name was provided as an argument and inserts the contents of that file at the current position in the file containing the INCLUDE statement.¹

The including relation between source files can represented as a directed graph. Nodes of such graphs would be the source files and there would be an edge from file foo to file bar, if bar includes foo. If we were serious about generating user maintainable code, preserving the structure of this graph would be an absolute requirement. This actually proved to be the hardest part of the translation process. There were many problems we had to address; many of these problems did not have fully satisfactory solutions and there is no doubt that many problems were left undiscovered by the time we had finally abandoned the idea of generating readable code.

Let us begin with a small example. Suppose file inc.fh contains

```
COMMON /FOO/ A
```

file src1.f contains:

¹The INCLUDE is not formally a statement; it is more like a directive.
INCLUDE "inc.fh"
REAL A

and file src2.f contains:

INCLUDE "inc.fh"
DOUBLE PRECISION A

The common block FOO, has now two different definitions. According to one of them, A is of type REAL, according to the other one, of type DOUBLE PRECISION. This demonstration of bad programming style would luckily be detected, since we require all instances of a given common block to have identical definitions. This should not lull anyone into thinking that the complications caused by the use of INCLUDE can be easily solved by already existing constraints. Consider another example:

File "inc.fh"
INTEGER A

File "src1.f"
INCLUDE "inc.fh"
COMMON /FOO/ A

File "src2.f"
INCLUDE "inc.fh"

C just use A

This all seems not to cause direct conflicts, until we try to imagine what the output code would look like. Since A is a common block member, it would be removed and references to it would be replaced with appropriate references to GPHMEM's objects. This would badly break the code in file src2.f, since it relies in the non-default type declaration of the variable A. The declaration could be left untouched by the translator; however, in
general, it would not be possible for it to determine if it is necessary at the point of processing the file src1.f. Even if we decided to apply this approach, we could not be sure if there were other cases not covered by this solution. The translator was intended to take declarations out and replace them with other declarations. Such a replacement was declaration-for-declaration, rather than line-by-line, so having declarations that span across different source files seemed awfully wrong. Therefore, we have decided that all declarations would have to be fully contained in a single source file. We did not care if it was an included file or not, we just did not want to face all the potential troubles that split declarations could cause. Similar problems could occur if the definition of a common block was distributed across files:

File "inc.fh"
   REAL A
   COMMON /FOO/ A

File "src1.f"
   SUBROUTINE S
   INCLUDE "inc.fh"
   REAL B
   COMMON /FOO/ B
   C ...
   END

File "src2.f"
   SUBROUTINE T
   INCLUDE "inc.fh"
   REAL B
   COMMON /FOO/ B
   C ...

In both, `src1.f` and `src2.f` the common block `FOO` would be defined as having two REAL members: `A` and `B`. No mismatch would be detected at this stage. The output code would contain a common block with appropriate indices\(^2\), call them `INDEX_A` and `INDEX_B`. But where would that common declaration be placed? To preserve the structure of the declarations, the `INDEX_A` would have to be declared in `inc.fh` and the declaration of `INDEX_B` in both, `src1.f` and `src2.f`. In other words, we would need to split the definition of the resulting common block to reflect the placement of the original declarations. Again, this was definitely an ad hoc solution meant to address this problem only. There was no guarantee, or even evidence that we would not encounter other issues not addressed by it. Once more, an additional requirement was created: *all common blocks should be entirely defined in the same source file.*

The `EQUIVALENCE` statement is evil by itself, but in conjunction with an `INCLUDE` statement it shows its true destructive power. Consider the following:

File "inc.fh"

```plaintext
EQUIVALENCE (A,B), (C,D)
```

File "src1.f"

```plaintext
INCLUDE "inc.fh"
COMMON /FOO/ A
REAL B,C,D
```

File "src2.f"

```plaintext
INCLUDE "inc.fh"
COMMON /BAR/ C
REAL A,B,D
```

\(^2\)as described in chapter 13
Let us analyze this problem closely. The **EQUIVALENCE** statement in file *inc.fh* establishes an association between A and B and between C and D. From section 16.2 we know that the equivalence lists containing members of common blocks will be removed from their original **EQUIVALENCE** statements (or, if the **EQUIVALENCE** statement would otherwise be left with no equivalence lists, it would be removed altogether). Let's have a look at the file *srcl.f*. It puts A in a common block and thus forces removal of the equivalence list (A,B). The other association is still in effect, though, and the equivalence list (C,D) cannot be deleted. The **COMMON** statement in the file *src2.f*, in turn, makes C a common block member. This time, the equivalence list (C,D) is removed and the (A,B) should be retained. The two cases are not only conflicting, they are exact opposites of each other. Clearly, we cannot make both of the required translations, but, at the same time, there is no reason to favor one over the other. Therefore another restriction had to be added to our list: *included files cannot contain EQUIVALENCE statements*. To prove that running away from the problem was probably the best idea, we present a truly devilish example for reader's enjoyment:

File "inc1.fh"

    EQUIVALENCE (A,C1)

File "inc2.fh"

    EQUIVALENCE (B,C2)

File "inc12.fh"

    INCLUDE "inc1.fh"
    INCLUDE "inc2.fh"
    EQUIVALENCE (A,B)

File "srcl.f"

    INCLUDE "inc12.fh"
COMMON /FOO/ C1
B = 1.0

File "src2.f"
INCLUDED "inc12.fh"
COMMON /BAR/ C2
A = 2.0

The following diagram should make it easier to visualize the inclusion relation.

```
EQUIVALENCE (A,CI)  EQUIVALENCE (B,C2)
    
    EQUIVALENCE (A,B)
    
COMMON /FOO/ C1  COMMON /BAR/ C2
B = 1.0  A = 2.0
```

Finally, there are issues with generating the allocation routine. The function that will call GPSHMALLOCC needs to know the size of the common blocks. It does not matter that the translator is not aware of what the value would really be, it only needs to generate calls to GPSHMALLOCC with proper arguments. If the arguments contained symbolic constants, we would have to make their values known for the compiler. In other words, the allocation routine would have to have PARAMETER statements for all constants used in the calls to GPSHMALLOCC. There are several problems with that:

1. If the PARAMETER statements are contained in a separate header file, we would or would not be able to generate an INCLUDE statement to bring them in. After all, we would not know, whether or not these files contain anything that would collide with what is already present. If multiple files were to be included, we could not guarantee that they would not contain conflicting declarations.
2. If the `PARAMETER` statements are an explicit part of the program unit that uses them, we would not be able to include the files in which they are defined.

   In both cases, we could "remember" the values of all parameters and just generate a set of fresh `PARAMETER` statements in the allocation routine, but this would violate our principle that there should not be any new definition points introduced. A programmer who later changed the value of one of the parameters, would also have to change the value in the allocation routine.

   Since the new approach was to only preprocess the source, without caring too much for the readability of the output, we could simply paste the contents of the included files into the generated code. The entire set of problems associated with the `INCLUDE` statement has thus gone away.
There are different kinds of manipulations that the translator performs. What the previous chapters discussed, was how the source code is parsed and prepared for the processing that we will now present. This chapter describes the actual essence of the translation process—the substitution of the common block members with their dynamically allocated counterparts.

18.1 Top-level array (TLA) expansion

The essence of the translation is replacing objects of one kind with objects of another. Existing common blocks are replaced with common blocks holding information needed by GPSHMEM, references to members of former common blocks are replaced with references to the GPSHMEM's reference arrays. For example, assuming that \( x \) is a REAL variable and a member of some common block, an assignment

\[
x = y
\]

would become something similar to

\[
\text{GPS\_REAL(INDEX\_X)} = y
\]

Given a REAL array \( A \), again a common block member, an assignment

\[
A(E) = y
\]

would become something like
GPS_REAL(INDEX_A+E) = Y

In other words, a reference to a common block member is always replaced with a reference to an element of an appropriate reference array from GPSHMEM. There is one problem with this: every reference to an object subject to translation becomes a reference to a single array element, but not all of the original references must have been variables or array elements. In some circumstances Fortran allows using array names as ways to specify the entire array. If a whole array is somehow translated into a single element, the meaning of the program could change. We have investigated the cases in which such situations can occur. References to entire arrays are allowed in several different situations, besides declarations, of course:

1. In function and subroutine calls, for example

   REAL A(10)
   CALL FOO(A)

2. In EQUIVALENCE statements:

   REAL A(10), B
   EQUIVALENCE (A,B)

3. In certain I/O statements in the list of objects to be read or written:

   REAL A(10)
   READ (*,*) A
   WRITE (*,*) A
   PRINT *,A

4. In DATA statements:
REAL A(10)

DATA A /10*2.71/

5. In SAVE statements.

6. In other situations, like in input/output statements, but as either unit identifier or format identifier. In such cases, however the array must be a CHARACTER array. Since our restrictions prevent CHARACTER objects from being common block members, this case would never be subject to translation.

Let us examine the meaning of the array reference in all these cases so we can appropriately address this issue. The first two cases are conceptually very similar. There is an association that takes place between the array and some other object. The other object is either a dummy argument provided in the subprogram definition or another object in the relation $\varepsilon$ established by the set of EQUIVALENCE statements. The association is always between the first storage elements and so exactly the same effect would be achieved by specifying the first element of the given array explicitly, instead of referring to the whole array. For example,

    SUBROUTINE S(A)
    REAL A(10)
    C
    ...
    END

    REAL A(10)
    CALL S(A)
    CALL S(A)
    CALL S(A(1))

\footnote{See chapter 16, section 16.1 in particular for more details on the relation $\varepsilon$.}
The two last calls to subroutine S are functionally identical. Now suppose we have the following case:

```fortran
REAL A(0:9)
REAL B
EQUIVALENCE (A,B)
```

The `EQUIVALENCE` statement can be replaced with

```fortran
EQUIVALENCE (A(0),B)
```

without affecting the semantics of the program. It is important not to forget that the first element of an array is not always the element with an index of 1.

Providing a general recipe for solving the problem in these two cases should now be easy. Assume we have an n-dimensional array A with the following dimensions:

```fortran
A (L₁:U₁,...,Lₙ:Uₙ)
```

The appearance of the symbol A without subscripts in subprogram calls and in `EQUIVALENCE` statements should therefore be replaced with `A(L₁:U₁).

The `SAVE` statements are, just like in the last case, very simple to deal with. `SAVE` statements are treated just like `COMMON` statements and require no changes of any kind.

The only two cases left are the I/O statements and `DATA` statements. The shared objects can only be initialized via `DATA` statements placed in `BLOCK DATA` program units. It is an error for a shared element to appear in a `DATA` statement outside of `BLOCK DATA`. This way the set of circumstances of concern to us has been narrowed down to two cases: array names in `DATA` statements inside `BLOCK DATA` and array names in input/output statements denoting objects to be transferred. In both cases, an array name has exactly the same meaning: it specifies all elements of the array in their storage order. Fortunately, Fortran provides programmers with facilities allowing them to express that explicitly: both, `DATA` statements and input/output statements allow the use
of implied-DO lists. Therefore, to convert an array name to equivalent form referring to the array only in terms of its elements, we should replace the array name with an appropriate implied-DO list. Given the dimension specification above, a reference to an array A would be translated into

\[ A \ldots ((A(I_1, \ldots, I_n), I_1=L_1, U_1) \ldots), I_n=L_n, U_n) \]

It should be noted that the conversion of array names to implied-DO lists in DATA statements will not be visible in the final output. The reasons for that are that

1. DATA statements not contained in BLOCK DATA are not converted, and

2. DATA statements contained in BLOCK DATA are eventually rewritten as a sequence of direct assignments.

In the early stages of the design, we informally thought of an array name as an expression that evaluated to “an array.” Obviously, in all circumstances in which an array name was allowed, it would not be permitted to be a subexpression of another expression. In a way, an array name would then be a “top-level” expression with respect to the containment relation. It was often referred to as a “top-level array” in our notes, or TLA for short. The process of replacing references to arrays with references to their elements was thus called “TLA expansion.” The name is not really formally justified, but it was good enough for our internal purposes. As time went by the name stuck and we decided to leave it.

### 18.2 Introducing GPSHMEM elements

The TLA expansion makes sure that for any shared array, the array is not referred to just by its name. After the TLA expansion takes place, all references to shared objects are either references to variables or array elements. At this point, we are ready to replace
the shared objects with references to the GPSHMEM's reference arrays. The process seems rather straightforward at this point. If $X$ is a variable of type REAL, an occurrence of $X$ in an expression would be replaced with a reference to $\text{GPS}_{\text{REAL}}(\text{INDEX}_X)$. For an array, the situation is just slightly more complicated. In

```plaintext
REAL A(L:U)
C ...
A(K) = B
```

the assignment would be replaced with

$$\text{GPS}_{\text{REAL}}(\text{INDEX}_A + (K-L))$$

For any shared object $Z$ of type $T$, the corresponding index, say $\text{INDEX}_Z$, is such that $\text{GPS}_T(\text{INDEX}_Z)$ refers to the first storage unit allocated for $Z$. Hence, all references to array elements need to be adjusted to reflect the actual distance from the element being referred to and the first storage unit of the array. Distances between elements of an array are used frequently and so we found it useful to define a function that calculates a distance between two given elements of an $n$-dimensional array:

**Definition 18.2.1.** Suppose that $A$ is an $n$-dimensional array with dimensions $A(L_1:U_1, \ldots, L_n:U_n)$. For $i = 1, \ldots, n$, define

$$s_i = U_i - L_i + 1.$$ 

Let $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$. Define a function $OF_A$ as follows:

$$OF_A(x, y) = (y_n - x_n) s_{n-1} + \cdots + (y_2 - x_2) s_1 + y_1 - x_1$$

With this definition in hand, we can thus formalize part of the translation process. Given a variable $X$ of type $T$, all occurrences of $X$ become $\text{GPS}_T(\text{INDEX}_X)$. For an array
A and x as in definition 18.2.1 of type T, a reference to array element A(x) becomes
\text{GPS}_T\text{(INDEX}_A + OF_A(l,x)), where l = (L_1, \ldots, L_n)}.

The introduction of the GPSHMEM objects occurs in all places where the shared objects are referenced, including equivalence lists. This will allow a smooth execution of the next stage.

18.3 Equivalence substitution

The only thing left to do at this point is to take care of the pending \texttt{EQUIVALENCE} statements. The process was described in chapter 16, so we will not repeat it here. We will just provide the definition of the function \texttt{REWRITE-LHS}, as promised in chapter 16. The function \texttt{REWRITE-LHS} takes three \texttt{lhs} terms, say \texttt{s}, \texttt{p} and \texttt{c}. The argument \texttt{c} is the shared object as it appeared on the equivalence list. The argument \texttt{p} is an object from the same equivalence list\footnote{and same equivalence class} and finally, \texttt{s} is an \texttt{lhs} term occurring in the source code and pending for rewriting. At this point, \texttt{c} will already be a reference to an element of one of the GPSHMEM's reference arrays, so it will have a form of \text{GPS}_T(Z). There are still two possibilities for \texttt{s}: it can be a variable or an array element. In the first case, \texttt{s} can simply be textually replaced with \texttt{c}. Note that \texttt{p} was not needed this time. The next case is slightly more complex—\texttt{s} is an array element. Let's see an example of such a situation before we proceed:

\begin{verbatim}
REAL A(10), B(10)
EQUIVALENCE (A(6),B(1))
B(5) = 2.0
\end{verbatim}

If A was a shared object, the last assignment would have to be rewritten in terms of A. The element B(5) is associated with the element A(10), so the assignment would be A(10) = 2.0. It is clear that in order to properly calculate the final reference, we
need to know exactly what elements are associated and the relative positions between all involved entities. In this example, c would be A(6), p would be B(1) and s would be B(5). Because of the EQUIVALENCE statement we know that A(6) and B(1) refer to the same object. We need to find out how to express B(5) in terms of A. For convenience, let's assume that the function Addr returns the number of the first storage unit of its argument. Then

\[ \text{Addr } B(5) = \text{Addr } B(1) + 4 \]

Since \( \text{Addr } B(1) = \text{Addr } A(6) \), we get that

\[ \text{Addr } B(5) = \text{Addr } A(6) + 4 \]

Since we know that both arrays, A and B have the same underlying type,

\[ \text{Addr } B(5) = \text{Addr } A(10) \]

We have shown that elements B(5) and A(10) share the same first (and only) storage unit. In other words, they are synonyms for the same element. Let us now recreate this reasoning in a general case. Assume A(z) and B(x) are associated by an EQUIVALENCE statement and we need to rewrite B(y) in terms of A, where x and y are vectors of coordinates of the same dimension and z is an expression. Then

\[ \text{Addr } B(y) = \text{Addr } B(x) + OF_{B}(x,y) \]

Thanks to the EQUIVALENCE statement, we have

\[ \text{Addr } B(x) = \text{Addr } A(z) \]

Therefore,

\[ \text{Addr } B(y) = \text{Addr } A(z) + OF_{B}(x,y) \]

We are now ready to present the complete definition of REWRITE-LHS.
Rewrite-lhs \((s, p, c) = c\)

if \(s\) is a variable, and

\[
\text{Rewrite-lhs} \ (s, p, \text{GPS_T(Z)}) = \text{GPS_T}(Z + OF_A(x, y))
\]

if \(s\) is an element of array \(A\).

It is important to notice that the GPSHMEM's reference arrays are all 1-dimensional. All other objects are thus "flattened" into a vector, regardless of their original dimensions. If the "destination dimension" was not 1, we could have more trouble with this process as well as issues crossing the C-Fortran boundaries.

Let's briefly summarize the whole rewriting procedure:

1. Expansion of all array names into references to their elements by replacing the array name either with a reference to its first element or with an appropriate implied-D0 loop.

2. Replacement of all shared objects with appropriate references to the GPSHMEM reference arrays.

3. Substitution of the elements associated with the shared elements (by now expressed by GPSHMEM's reference arrays) with appropriate references to those shared elements (reference arrays, that is).

18.4 Introducing new symbols

On several occasions there is a need for us to introduce our own symbols into the scope of a given program unit. This happens when

1. a common block with indexes and handles is declared, or

2. an array is expanded into a implied-D0 list in an input/output statement.
As noted before, when an array name is expanded into an implied-DO list in a DATA statement (recall that this only happens in BLOCK DATA subprograms) it is immediately rewritten as a sequence of assignments, so the variables otherwise introduced by the implied-DO list are eliminated. When a new symbol is inserted in some scope, there is always a problem of a name clash. One of the unfortunate features of the Fortran 77 language is that it does not require variables to be declared. Therefore, some symbols “spring into existence” when they are first used. We do not examine the whole code in search of all existing symbols—we are only concerned about shared symbols and those must be explicitly declared as such. Because of that, we only know a certain subset of names used by a given program unit. To minimize the risk of name clashes, each name generated by the translator starts with a certain prefix. This prefix can be set by the programmer using a command line option or the default value can be used.

18.5 The big picture

It is time to summarize the whole translation process and put together all the details presented so far. When the translation begins, the given source file is opened and read. The translation occurs one program unit at a time. When a subprogram or a BLOCK DATA program unit is read, all declarations contained therein are stored in special data structures. All statements, including declarations, are also stored in the parsed form in an internal list. Each time an END statement is encountered, the substitutions described above are performed. The next step is to traverse the statement list, eliminate declarations of replaced variables and add declarations of new symbols and common blocks. Finally, the preprocessed code is unparsed and printed to the standard output. After all files have been preprocessed, the translation process concludes with the generation of the new startup and finalization code.
19 Final stage

The final part of the translation is to wrap things up and prepare the code to run. Among the things that need to be performed by the translated code is proper initialization and finalization of GPSHMEM, actual shared memory allocation and deallocation, and finally, setting up the indices for all shared objects and the initialization previously performed by BLOCK DATA subprograms. All these tasks are carried out by a special main subprogram. The subprogram does the following:

1. Call GPSHMEM_INIT.

2. Call allocation routine.

3. Call all subroutines created out of previous BLOCK DATA subprograms.

4. Call the subroutine containing the previous main program.

5. Call the deallocation routine.

6. Call GPSHMEM_FINALIZE.

The allocation routine calls GPSHMALLOC for each existing common block or a set of statements listed in a SAVE statement. The indices of all members of the block of shared data are obtained by calling GPSHINDEX and adding a precomputed constant offset.
20 Fortran 77 source code analysis

The source code analysis described in this chapter refers to the two initial stages of all language translations: lexical analysis and syntactical analysis. The fgpp's lexer a part of AT&T's Fortran-to-C translator f2c [44]. The lexical analysis of most modern programming languages is usually the easiest step in the compilation (or compilation-related) process. The lexical tokens can usually be easily described in terms of regular expressions. Once such a description is prepared, a lexical analyzer can be automatically generated by one of many freely available tools, like lex [81]. Unfortunately, the lexical tokens of the Fortran 77 language cannot be expressed as matching any set of regular expressions. There are several reasons for that. First of all, Fortran 77 ignores all blanks. Therefore both SUBROUTINE and SUBROUTINE are valid keywords and have the same meaning if they occur in the same context.

Another reason, sufficient to make the set of lexical tokens a non-regular language, is the use of Hollerith constants. A Hollerith constant has a form

\[ nHc_1c_2...c_n, \]

where \( n \) is a positive integer and each of \( c_i \) for \( i = 1, \ldots, n \) is a character. It can be proved that there is no regular expression that describes Hollerith constants. Of course, having a non-regular subset does not prove that the set itself is non-regular, but in the case of lexical tokens in Fortran 77 it can be argued that it indeed is not.

One more problem with the lexical analysis of Fortran 77 is that it does not have
reserved words. Words that correspond to keywords can be used as identifiers and their meaning must be deduced from the context. For example, the code below is valid:

```fortran
SUBROUTINE F00(*)
RETURN = 1
RETURN 1
END
```

The first occurrence of the symbol `RETURN` is a part of an assignment of number 1 to a variable with name "RETURN." The second one is an alternate `RETURN` statement with 1 as its argument.

The `f2c` uses a hand-written lexer that generates a stream of lexical tokens in a form that can be used with Yacc [72]. In `fgpp` we have borrowed the lexer with a few changes. The entire lexer has been cut down to three C source files: `lex.c`, `lexsupp.c`, `lexerr.c` and several header files. The files `lexsupp.c` and `lexerr.c` are not a part of the original `f2c`: they contain the routines necessary for the functions in `lex.c` to work. Most of the changes that have been made eliminated the unused code and declarations.

Currently the lexer is a self-contained module within the `fgpp` and is loosely tied with the rest of the code. Its functions are only called from the parser, except for the initialization and cleanup routines which are called from the function `main`.

The parser used in `fgpp` was, similar to the lexer, based on the Yacc specification from `f2c`. The difference here is that while the lexer was adopted from `f2c` with only a few changes, the parser has undergone a massive overhaul. Nearly all of the semantic actions associated with the grammar productions were removed. Some of the unnecessary productions have also been eliminated and some new grammar rules have been added. All of these changes were necessary, since the parser in our translator played a slightly different role than in `f2c`, and needed to be able to operate on completely different data structures. The parser was intended to perform two major functions:
• translate the source code into the internal representation, and

• create data structures, such as variable dictionaries, needed by later processing routines.

The second task only applies to the specification statements; the executable statements are stored in memory without affecting the state of the translator.
21 The \textit{fgpp}'s internals

This chapter sheds some light on the inner workings of the translator. It is not intended to be a complete reference to all of the implementation details of \textit{fgpp}. Instead, it describes some of the \textit{fgpp}'s internal mechanisms. First, we present a brief introduction to the internal representation that \textit{fgpp} uses to store Fortran 77 code. Then, some practical aspects of manipulating the data structures are presented with, in particular, a discussion of a common problem of sharing pointer values and our approach to it.

21.1 The syntax tree

During the syntactical analysis phase, the source code must be stored in memory for further processing. The choice of internal representation is important, because it needs to accommodate all necessary translations. It also must allow for unparsing—generating Fortran 77 code of equivalent functionality. The internal representation must, therefore, store more information than a Fortran 77 compiler would do. The most natural choice for such representation is a tree reflecting the hierarchical structure of source code statements. The source code of a Fortran 77 program differs from that of most newer programming languages. In C++, for example, a program is a list of declarations, and a declaration can contain statements (e.g. function body) or nested declarations. Statements correspond to logical program elements and are not related to the way in which a program is represented in the source. Fortran 77 places some restrictions on what the source code can look like. Apart from the infamous fixed format, Fortran
forces certain language elements to appear on separate lines. Consider, for example, the following C++ piece of code:

```cpp
if (num > 0) {
    foo ();
} else {
    bar ();
}
```

The entire example is a single statement and can be represented as a single tree node with three fields: the condition, and the two execution alternatives. The Fortran counterpart of that example, on the other hand, would look like this:

```fortran
IF (NUM.GT.0) THEN
    CALL FOO
ELSE
    CALL BAR
ENDIF
```

It is not a single conditional statement anymore. These are five different statements (one statement per line) that are logically connected only by their relative positions in the source code. In general, a Fortran 77 program is a list of statements, and the concept of a statement may differ slightly from the more common view of it inherited from modern languages. These differences influence the shape of the internal code representation. While the above code could be represented by a single tree node, generation of it could be unnecessarily complex and would add no advantages. The internal representation chosen for fgpp was, therefore, a list of statements. Despite that, we will still refer to the internal representation as "syntax tree." Each statement node would contain pointers to data structures describing its parameters. The further analysis and translation procedures
would operate on these structures and, finally, the unparsing stage would convert them back into the form of a Fortran program.

21.2 Syntax tree manipulation

All of the translation efforts take effect via operations made to the internal representation. The statement list is traversed, its elements are examined, moved, copied and, finally, deleted. Initially, the programming language used for the implementation of the translator was C. To avoid creating multiple structurally identical types, the syntax tree nodes were modeled after their contents without fully reflecting their meaning and usage. For example, all tree nodes containing two children and a pointer to the next node on the list, would be represented by a data structure `p2n_t` defined as

```c
typedef struct tag_p2n_t {
    struct tag_p2n_t *next;
    void *p1, *p2;
} p2n_t;
```

The actual meaning of the data contained in such a structure could only be deduced from its placement in the tree. The top-most node was a statement: specification, executable or other. Functions operating directly on such objects would call other functions to process their contents, represented in the above example by the data pointed to by `p1` and `p2`. Thus, all of the functions needed to be passed a correct data structure. Since data types did not reflect the meaning, and objects of the same type were used for different purposes, we could not rely on type checking to warn us about improper use of data structures. To make things even worse, even in the case of a single function expecting a particular kind of object, it was not always able to deduce the real meaning of its argument. Therefore many of the internal structures had a field indicating their exact meaning. Many functions consisted of a big switch statement with often significantly
different actions associated with different values of the meaning field. As the program grew in size it was becoming more and more apparent that this approach would not lead to an elegant solution. The reduction of apparently unnecessary data structures was resulting in massive code bloat and obfuscation. Imagine a simple operation on the entire internal source code representation, let it be as simple as deleting it from memory. In order to facilitate this simple task, we would need a function that can deallocate an element of the statement list. This function would then invoke an appropriate deallocation routine, depending on the actual statement being deleted. Since all statements would need to be represented by the same data type at the top level\(^1\), decisions about the choice of the proper deallocator for the data pointed to by member pointers would need to be based upon the value of the meaning field. These “second-level” functions would share similar problems, and, effectively, there would need to be an individual deallocation function for each tree node. If each operation was to be as complicated as this, we were facing some serious trouble.

One of the major problems seemed to be the fact that we could not just store the semantic information carried by the source code; but needed the syntactical elements as well. Without it, regenerating the source code could turn out to be impossible. Technically, all conditional statements could be unified into one “if-then-else” abstraction, but first, it would complicate the parser, since the creation of such an abstraction would have to span across several productions, and second, the generated Fortran code could look significantly different from the original one. Since, at that time, one of the requirements was to generate human-readable code\(^2\), we decided against such unifications.

After our first attempt to implement the translator purely in C exposed several important issues, we decided to switch from C to C++. What C++ offered that C did not have were classes and virtual functions. The problems with the tree deallocation

---

\(^1\)Since they will need to be processed by the same function, with a fixed parameter types.

\(^2\)By, among the others, preserving as much of the original program structure, as we could.
described above could be solved in one, or at most a few, lines of code, with the help of object destructors. Of course, each class would need to have a destructor defined for it, but there would be no risk of calling a wrong one since the compiler would take care of it for us.

C++ does not solve problems automatically, it only provides some features that make dealing with certain issues easier than in C. The next chapter describes our approach to sharing data objects between different structures.

21.3 Pointee sharing

The experience with the initial versions of the translator written in C showed two things. The first was that the usage of pointers is inevitable and that the number of them will explode. The second was that once the delicate network of pointers is created, any manipulations of this network requires extreme care and any, even the simplest, mistake can and will have disastrous effects. There seemed to be two major problems: ability to properly destroy parts of or the whole data structures, and ability to make changes to them without causing any damage. The first problem was overcome by switching to C++ and using constructors and destructors. The second problem did not have such an elegant solution. The hardest problem to deal with was duplicating the data structures. Often there was no need to allocate memory and keep separate copies of those objects. Copying of objects would imply the copying of all of their subobjects as well. A “deep-copy” function would need to be implemented for most, if not all data types. The creation of separate copies was, most of the time, unnecessary since the values remained unchanged and could be shared with no collisions. The problem, however, would, again, be deleting shared structures. Obviously, a data object should not be destroyed while it is still in use, so some protection mechanisms needed to be employed. Thus, a facility allowing safe sharing of pointers was created. This facility
is a simple class that implements a reference counter and some functions allowing the proper use of it. It is important to note that this is not an attempt to create a garbage collector for C++. The use of the reference counter requires full cooperation from the programmer and can be easily "defeated" by a nonconforming user. However, its main purpose was to enable the writing of working code and not to thwart the programmer's attempts to break it.

The implementation of the reference counter is provided by a class template deathctl.t, whose name originated as an abbreviation of "death control." It contains a member function template copy that returns the pointer to the object on behalf of which it is called, increasing the reference counter at the same time.

The member function destroy decreases the reference counter and deletes itself after the counter has reached 0.

Some older compilers do not implement member function templates and so they will not be able to compile the foo.p. Let us elaborate on how the decision to use member function templates was made.

The most intuitive way to implement some functionality that is to be shared among multiple classes is to encapsulate it in a separate class and have other classes inherit its properties. This is how the deathctl.t was designed. Let us consider a candidate class deathctl1.t that is not a template, but an ordinary class. A typical usage of it would be

```cpp
class foo.t : public deathctl1.t {
    /* implementation */
};
```

The deathctl1.t would provide a member function to create a shared copy of the pointer to the object on behalf of which it was invoked. Now, we face a question: what is the return type of such a function? If it is just a pointer to deathctl1.t, every
time it is called, the return value would have to be converted to an appropriate type (downcasted). Following the C++ standard and coding guidelines, such downcasting would be performed using the `dynamic_cast` facility. This solution, although valid, sounds like a big headache. Let us try something else that would not require an explicit type conversion every time it is used. A virtual function immediately comes to mind.

Let us consider another candidate for the proper "death controller":

```cpp
class deathctl2_t { 
    /* have the reference counter here */

public: 
    virtual deathctl2_t* copy () = 0;
};

deathctl2_t *deathctl2_t::copy () { 
    /* adjust the reference counter */
    return this;
}
```

Now, every class derived from `deathctl2_t` would be forced to provide its own implementation of `copy` that, thanks to "covariant return types," would return a pointer value of appropriate type. An explicit conversion would still be necessary, but it would not appear at the point of invocation of `copy`:

```cpp
class foo_t : public deathctl2_t { 
    /* ... */

public: 
    virtual foo_t *copy ();
};
```
foo_t *foo_t::copy () {
    return dynamic_cast<foo_t*>(deathctl2_t::copy());
}

The conversion has been simply moved to the body of copy. This requires the programmer to implement the function copy for each class derived from deathctl2_t. Again, while valid, this solution also seems to demand a lot of additional work, although the work has been completely shifted from the point of use to the points of implementation. An ideal solution would be contained in a single point of implementation and require no additional actions at the points of use, a hybrid merging the features of both of the approaches described above. The function copy implemented as a member function template seems to be the closest to this. The following is a fragment of the template class deathctl_t:

    template <class T>
    class deathctl_t {
    public:
        template <class U>
        U *copy () {
            ref_cnt++;
            return dynamic_cast<U*>(this);
        }
    private:
        int ref_cnt;
    };  

Assume that a class foo_t was derived from deathctl_t. No additional member functions have to be added to foo_t. The use of copy now looks like this:

    foo_t *p, *q;
It is not ideal, an explicit type specification is still required, although it does not appear to be so cumbersome as in the case of `dynamic_cast`.

The convenience of use may not seem a valid reason to abandon the possibility of using compilers that do not support member function templates. However, member function templates are a part of C++, no worse than virtual functions or plain templates, and there is no reason to base the design decisions on some compiler's limitations.³

In C++ it is valid to `delete` null pointers. Such an attempt is ignored and causes no harm. Since the functions described above are member functions, they cannot be invoked on behalf of a nonexistent objects. To allow safe “destruction” of null pointers an external function `kill` has been provided. To avoid name collisions it has been placed in the namespace `deathctl`. All it does is call the `destroy` function for non-null arguments.

When a pointer to an object is passed to the `deathctl::kill` function, the member function `destroy` will be invoked. If the reference counter after decrementing drops to 0, the destructor will be called. Thus, the destructor will only be invoked when an object is no longer in use. To properly destroy objects pointed to by the member pointers, the destructor should call `deathctl::kill` for all pointer members of class types derived from `deathctl_t`.

---

³This is of course assuming that we are not tied to any particular compiler, but it was not the case here.
22 Conclusions

The intent for \textit{fgpp}, the Fortran 77 preprocessor, was to enable the legacy Fortran 77 code to be easily adopted for use with GPSHMEM. The major objective was to replace all statically declared objects that would be remotely accessible on systems with native SHMEM implementation with dynamically allocated memory blocks. Throughout the design process we have discovered a wide spectrum of problems, which have all had some impact on the final shape of the \textit{fgpp}. Some of them resulted in a set of restrictions that we placed on the source code subject to translation, others were significant enough to force us to change some of the design goals. Translation that preserves the structure of the source files and maintains human readability has proven to be far more difficult than we previously expected. At the end, we believe that the major design goals have been met and that the restrictions resulting from the unexpected issues will not have a significant impact of the translator's usability.

The implementation of a translator for a language like Fortran 77 involves some unique issues that are not often encountered in other programming languages. Fortran's legacy makes the initial translation steps, the lexical and the syntactical analysis, more difficult than, for example, that of Pascal. To avoid spending significant amounts of time on aspects not directly related to the essence of the translation, we have decided to base our front-end on the Fortran-to-C translator, \textit{f2c}. The choice of the programming language was mainly dictated by portability issues and thus it was C at the beginning. The low-level character of C, and especially the need for manual memory management, has convinced us to switch from C to C++. While C++ does not solve these problems,
it provides mechanisms that help alleviate some of them. Use of class inheritance and virtual functions allows to specialize objects' behaviors and factor out common functionality. Virtual destructors have proven to be very helpful in dealing with memory deallocation in heterogenous data structures. However, even with virtual destructors or virtual functions, the highly data-oriented character of language translation demonstrated the inadequacy of C++ for such tasks. Unfortunately, the choice of the programming language depends on more factors than just its applicable features. Things like compiler availability, library support or cost of maintenance must also be taken into account.

Future work would concentrate on making fgpp more user-friendly—in particular on enabling incremental processing. Such an approach could substantially reduce the translation time during application development process, since unchanged source files would not need to be re-translated. Other work could possibly try to remove some of the requirements we currently place on the source code, or even attempt do to that what we tried in the beginning—generating user-readable code.
BIBLIOGRAPHY


[100] Private communication with Robert Numrich.


