**Translation techniques for distributed-shared memory programming models**

by

Douglas James Fuller

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Ricky A. Kendall, Co-major Professor
Gary T. Leavens, Co-major Professor
Mark S. Gordon

Iowa State University

Ames, Iowa

2005

Graduate College
Iowa State University

This is to certify that the master's thesis of

Douglas James Fuller

has met the thesis requirements of Iowa State University

Co-major Professor

Co-major Professor

For the Major Program

# TABLE OF CONTENTS

# LIST OF FIGURES

IS-T 2741

# CHAPTER 1 Introduction

## 1.1 The Overall Problem

The high performance computing community has experienced an explosive improvement in distributed-shared memory hardware. Driven by increasing real-world problem complexity, this explosion has ushered in vast numbers of new systems. Each new system presents new challenges to programmers and application developers.

Part of the challenge is adapting to new architectures with new performance characteristics. Different vendors release systems with widely varying architectures that perform differently in different situations. Furthermore, since vendors need only provide a single performance number (total MFLOPS, typically for a single benchmark), they only have strong incentive initally to optimize the API of their choice. Consequently, only a fraction of the available APIs are well optimized on most systems. This causes issues porting and writing maintainable software, let alone issues for programmers burdened with mastering each new API as it is released. Also, programmers wishing to use a certain machine must choose their API based on the underlying hardware instead of the application.

This thesis argues that a flexible, extensible translator for distributed-shared memory APIs can help address some of these issues. For example, a translator might take as input code in one API and output an equivalent program in another. Such a translator could provide instant porting for applications to new systems that do not support the application's library or language natively. While open-source APIs are abundant, they

do not perform optimally everywhere. A translator would also allow performance testing using a single base code translated to a number of different APIs. Most significantly, this type of translator frees programmers to select the most appropriate API for a given application based on the application (and developer) iteslf instead of the underlying hardware.

## 1.2 Background

The Partitioned Global Address Space (PGAS) model (§2.2.2) is a popular theoretical model implemented by many system vendors and open-source systems that admits a wide diversity of hardware types. Its rapid evolution has resulted in the availablity of many different APIs with differing performance (and availablity) characteristics. Our translator will target this model, given its widespread use for today's real world applications.

Our target here will be models based on the C programming language, as many currently available models are implemented in or derived from it in some form. Focusing on the implementation language allows us to readily compare characteristics in the various models considered by observing syntax as well as semantics. It also allows us to isolate performance characteristics imparted by the models themselves as opposed to any optimizations offered by individual compilers.

Specifically, our focus will be on the Global Arrays (GA) library (§3.1.2), and the Unified Parallel C (UPC) programming language. GA is an array-based, explicit model for handling distributed, multidimensional arrays. It provides data transfer routines, "convenience" routines for handing simple matrix and vector operations, and optimized collective routines for complex mathematics and linear algebra. GA also contains utility functions for directly controlling shared data layout.

Unified Parallel C (UPC) (§3.2.1) is a derivative of the C programming language

that features globally distributed and shared C data structures on parallel systems. It features implicit data transfer (through simple assignment) and adds minimally to the C syntax. Global data are stored using a static "blocking" mechanism. Supported by a consortium of laboratories and universities, UPC is increasing rapidly in popularity.

## 1.3 The Approach

GA and UPC were chosen here because they differ widely among the class of APIs for distributed-shared memory models. As such, a translator for these two models is strong evidence in itself that it is possible to add more APIs to the system. A more specific and detailed argument for this assertion is laid out in chapter 4.

In order to experiment with this type of translation, we present in this work a translation program that handles the GA and UPC models as both input and output and treat its implementation in detail. This program was implemented in C++ and derived from the `ctool` project (Ch. 5). `ctool` is a source-to-source translation program whose input and output languages are both standard C. This work extends the system to form a two-way translator between GA and UPC. `ctool` is built using the GNU `flex` and `bison` utilities, which are further utilized in this work.

Early results indicate that translation can have significant performance and programmability benefits. Several extensions to UPC are implemented using the translation system (§5.8) that correct systemic defects and significantly improve programmability, and preliminary performance numbers indicate that translation from UPC to GA can greatly improve performance in situations where sufficient amounts of local communication occurs. Performance gains are modest in other situations.

## 1.4 Thesis Overview

Theoretical background information is presented in chapter 2. A brief treatment of existing distributed-shared memory APIs and their characteristics is presented in chapter 3. Chapter 4 details a formal argument for the existence of a modular translation system for these APIs. Specific implementation details for the translator produced for this work are given in chapter 5. Finally, chapter 6 is a summary of our findings with and about the translator and a discussion of possible future work on the system.

# CHAPTER 2 Models of Parallel Computation

Each programming paradigm available for parallel computation is based on some underlying theoretical model of parallel computation that is either implemented directly or can be simulated by the hardware. Various theoretical models have different implications for the types of APIs that may implement them. While a full treatment of theoretical models for parallel computing is beyond the scope of this thesis, a brief glimpse at those available provides motivation and background for this work.

## 2.1 Shared-memory models

### 2.1.1 Parallel Random Access Machine

The Parallel Random Access Machine (PRAM) model of parallel computing [8] defines a system in which multiple, synchronized processes complete operations on globally shared data in lock-step. It is very convenient model, often permitting parallel computations with the same time complexity as their serial counterparts. Programming PRAM applications is generally quite simple and parallel algorithms expressed in this model often appear very similar in form to their serial versions as well.

This simplicity is a result of the assumption that PRAM processes may all read and write to all memory locations at any time. Various subtypes of PRAMs are defined in terms of contention for shared resources, that is, whether memory reads and writes may each be performed concurrently. For example, a PRAM in which contention for reading is allowed, but contention for writing is prohibited is called an Concurrent-Read,

Exclusive Write (CREW) PRAM. The behavior of each of these PRAM types is similar, and algorithm expression is similarly straightforward.

PRAM and its variants present a functionally simple model of parallel computing. Unfortunately, this simplicity is difficult (if not, impossible) to realize with practical parallel computing resources. Such a system, if it were to exist, would be required to implement a fully data-parallel model with no necessary communication primitives – that is, any locally threaded programming model could effectively implement a PRAM. While it does seem theoretically possible that a useful PRAM could be constructed, the hardware required would certainly be prohibitively expensive.

## 2.2 Distributed-memory models

### 2.2.1 Bulk Synchronous Parallel

The Bulk Synchronous Parallel (BSP) model [16] defines a set of individual processes which perform operations synchronized at a more coarse-grained level. BSP processes operate in a cycle of independent local computation phases, followed by synchronized global communication phases in which they exchange data with their peers. The processes then enter a synchronization phase and the entire process (known as as "superstep") is repeated.

Unlike shared-memory models, the BSP model assumes individual nodes carry their own, private memory. Aside from the coarse-grained compulsory synchronization, BSP processes operate independently from each other. The result is a more fundamentally distributed system, which is programmed more as a set of collaborating processes than as a monolithic system.

BSP is much more adaptable to existing parallel computing hardware, as its entirely local computation model frees the hardware from having to manage shared data accesses. With its explicit communication model, it is also adaptable to existing serial APIs.

Additionally, it provides for simpler time complexity computation and verification since "supersteps" execute in lock-step and computation and communication may not be overlapped.

This last point is not to be overlooked. BSP lacks some intuitiveness for programmers because of the strict separation between computation and communication. Still, it is a model designed with both hardware support and programmer convenience in mind. This convergence of intentions is important for any parallel programming model to receive wide acceptance in the community.

### 2.2.2 Partitioned Global Address Space

The Partitioned Global Address Space (PGAS) model (fig. 2.1) attempts to strike an interesting balance between API and hardware concerns. It acknowledges that, while more fundamental models such as PRAM present a convenient interface to the programmer, they do not easily accommodate scalable hardware concerns. This prompts the development of a model that balances the interests of the programmer (a simple API) and the interests of the hardware developer (a scalable infrastructure).

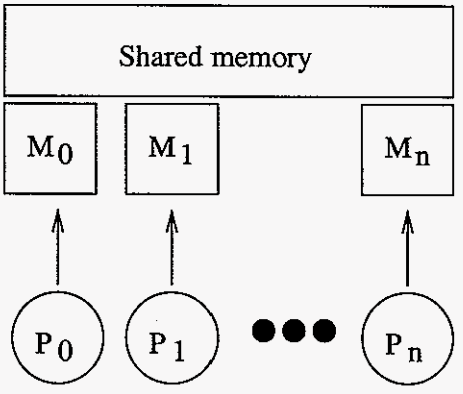


Figure 2.1 Conceptual diagram of a PGAS system

The PGAS model provides for a set of loosely coupled processors (nodes), each with

its own local execution stack, memory, and secondary storage. Additionally, a single, globally addressable memory is provided from a portion of the local memory on each node. While accessing this global memory is more computationally expensive than local memory access, it does not require the cooperation of any other node. Implementations may elect to expose locality information to the nodes in the system to save some computation.

Of course, contention for access to global memory creates synchronization issues. A PGAS system must provide shared, atomic locking primitives to provide synchronization features. These locking primitives may include simple binary semaphores or be extended to include waiting and signalling operations.

While they may be provided, explicit pairwise cooperative and collective communication primitives (traditional "send" and "receive" operations) are unnecessary under the PGAS model since nodes may read and write shared data independently. This feature may be used to provide the functionality that would ordinarily be provided by a message passing layer. Explicit message passing primitives do provide a level of convenience to the programmer, however, and enhances development and readability by remaining syntactically similar to other available APIs. This also provides a layer of software interoperability with scientific libraries that an application may require that are implemented using a message passing model.

The PGAS model assumes:

1. The system is composed of a set of interconnected nodes.

2. Each node contains some amount of local memory.

3. From a portion of memory on each node, a globally addressable, shared memory is formed.

4. Each node supports the following operations *using function calls and/or language*

*constructs:*

(a) Atomic copying from local memory to shared memory.

(b) Atomic copying from shared memory to local memory.

(c) Obtaining an identifier unique to each node.

(d) Global barriers.

(e) Basic mutex operations.

Thus, the PGAS model implements a `distributed-shared memory (DSM)` model.

## 2.3 Usefulness of the PGAS Model

The PGAS model lends itself well to currently available system hardware configurations. It is supported trivially by SMPs, and provides a great deal of flexibility running on such systems. Additionally, machines supporting non-uniform memory access (NUMA), support the PGAS model by definition, as they allow remote processes to access local memory, albeit with an increased cost. Since this cost is accommodated by the PGAS model, NUMA supports PGAS trivially.

Nearly all vendor-produced MPPs support PGAS, as they are either designed to execute serially authored code or provide remote memory access primitives. Many do both. As such, they are able to carve from their local memory spaces a globally addressable memory.

Recently, the PGAS implementations have been available on cluster-based systems. Obviously, this is far more complex than the architectures discussed above, requiring integration with underlying networking systems. This often requires support from the individual nodes' operating system kernels – either directly or indirectly in the form of drivers for attached network devices.

Since all these types of hardware support the PGAS model, APIs that implement it can often be ported to different systems of different types. Where possible, this is an enormous convenience for the high-performance software programmer. It provides source compatibility across a variety of platforms and system types. Implementations of such APIs are becoming available at a rapid pace as research into them progresses.

These implementations have manifested themselves in the forms of new programming languages (such as UPC [5], Co-Array Fortran [13], Titanium [6], and Split-C [3]), and new libraries for existing programming languages (such as Global Arrays, SHMEM, DDI, and ARMCI). Both of these methods provide PGAS features to high-performance software programmers.

There are several important concerns with respect to how such APIs are presented to the programmer:

- Familiarity: Similarity to existing models provides important conveniences to high-performance software programmers as they adopt new languages and APIs. It also simplifies the porting of existing applications to new platforms and systems.

- Paradigmatic simplicity: APIs and languages that maintain correspondence to other paradigms, especially serial programming paradigms, are easier to learn. They are more accessible to developers and can be adopted more quickly. They also produce programs which are easier to read for new or traditional programmers.

- Transparency: Since PGAS systems provide access to local and global memory, APIs and languages that present syntactic (or library) constructs for global memory access that are similar to those for local access will provide for programs with greater readability. They also allow programmers to deal with remote memory accesses more flexibly.

- Exposure: APIs and languages that make available more information and features

to the programmer facilitate better performance. For example, many implementations allow the programmer to determine the actual location (in a node's local memory) of a given portion of global address space. They may also provide extra synchronization primitives not strictly required by the model. Depending on the model used for global memory, they may provide additional operations on global data structures for the convenience of the programmer. This kind of information exposure allows programmers to adaptively adjust programs for performance.

Currently available APIs (Ch. 3) provide various levels of these features and, as such, vary widely in their syntax and performance characteristics. As such, the different types of languages and APIs vary widely. This results in an overabundance of programming APIs that become cumbersome to learn and difficult for programmers who wish to port software to a variety of platforms and system types.

# CHAPTER 3 Available PGAS implementations

## 3.1 Library-based Approaches

Many distributed/shared memory APIs are implemented as libraries in existing programming languages. This allows users and developers alike to continue to utilize familiar programming models with the syntactic and semantic constructs they are used to. It also leverages previously familiar language conventions. For example, it tends to benefit from the subset of existing standard library functionality that marries itself well with the implemented DSM API.

Library-based approaches also permit code to be re-used and/or re-fitted to conform to the new API, such as optimized local operations which may fit well as components in a larger-scale, global computation. This leverages the extant codebase that has evolved in serial and other parallel constructs to increase the immediate power of such APIs.

Library-based APIs are also, to some extent, source-code independent. Bindings for them may be created for multiple implementation languages, which allows for more broad support and acceptance. Of course, this requires that the languages for which bindings are created facilitate similar APIs. Still, there are broad categories of programming languages which together prove host to the same sorts of API concepts. Even object-oriented programming languages such as C++ and Java tend to lend themselves well to thinly-veiled procedural approaches. The broader acceptance achievable in this way fuels development effort both in improving the API itself and expanding the codebase available for it.

A marked drawback to library-based systems is that they cannot become involved in the type systems of the programming languages in which they are implemented. While object-oriented languages have inherently extensible type systems that allow for extensions within the existing framework, pure procedural languages do not generally have this ability. Unfortunately, more object-oriented programming languages have not found favor with high-performance software developers or API authors, so most library implementations are constrained to the relatively simple type systems provided by the procedural languages that implement them. This has the practical effect of forcing function-call syntax to be used for concepts which are more elegantly expressible using other semantic constructs. Furthermore, exporting these concepts into function code inhibits compiler optimizations specific to the underlying hardware.

### 3.1.1 SHMEM

The SHared MEMory (SHMEM) programming model was one of the earliest APIs to support DSM programming techniques. SHMEM allows individual processes to designate addresses through which data structures on an individual node may be manipulated by other nodes. These addresses must remain static throughout the lifetime of the program. Thus, the global memory space is defined piecewise, as the union of all such shared memory regions.

This arrangement is unlike other DSM APIs in that it does not strictly conceptualize the representation of data in the global address space. Any data may be shared simply by virtue of knowing the address and data type on the remote node. In addition, the global address space is not defined explicitly in terms of shared data structures. Rather, it is formed from statically allocated memory regions on individual nodes. These regions obey standard semantics for any other memory region.

The SHMEM API, while inherently library-based, requires a great deal of support from the underlying architecture. This support may be at the hardware level (Cray,

SGI, and Quadrics, and Dolphing all offer implementations), or at the software level in the form of code translators to other APIs. Obviously, tradeoffs for performance and portability are serious concerns in determining how to implement SHMEM on any given architecture.

### 3.1.2 Global Arrays

The Global Arrays (GA) toolkit [12] from Pacific Northwest National Laboratory provides a rich API for PGAS systems. It has an explicit model for data transfer between global and local memory, with primitives provided for the programmer to locate global memory on local nodes. All global data storage is modelled as shared arrays which may be explicitly accessed and manipulated by various one-sided methods.

GA also provides optimized collective, data-parallel operations for advanced functions on global data. Since the data are conceptualized as arrays, these include array, vector, and matrix operations. In addition to providing programmer convenience, these routines work to improve performance through optimized cooperation among participating nodes. This relieves the programmer of the intricate details of optimally managing the locality of the global memory space.

Data transfer between user code and library code is provided in the form of one-sided data transfer operations. Of course, because of the nature of C's multidimensional array support, single-dimensional buffers must be mapped onto GA's n-dimensional space. To perform this mapping, the user fills in an array specifying the number of elements in each dimension that are represented by a continuous buffer. The library then uses this information to determine how to read and write data from and to user buffers.

Basic locking ("mutual exclusion") primitives are also provided, as any PGAS implementation must, to avoid deadlocks when managing shared resources. GA's locking primitives are quite basic, lacking signalling semaphores, but they provide what is required by the model. Another form of synchronization is provided by sequential

scheduling of updates to remote arrays.

GA is implemented on top of the Aggregate Remote Memory Copy Interface (ARMCI), another PGAS implementation which does not provide as rich a feature set. This is an example of how the functional and expressive equivalence of these implementations facilitate cross-implementation and translation.

Conceptualizing global data as arrays carries interesting consequences for the programmer. Data that does not lend itself well to array-based data structures can often not be represented effectively with GA. Sparsely populated arrays (although recent developments are addressing this), binary code fragments, and compressed data are all difficult to model as arrays. Because its implementation languages are both strongly-typed procedural languages with little type-extensibility, GA does not extend the notion of type to deal with these types of issues.

An interesting aspect of a broad-sweeping library-based implementation is that it can transcend individual programming languages. In fact, it was originally developed for a Fortran application. GA bindings are available for Fortran, C++, and Python, providing virtually the same functionality for these programmers programmers as for C programmers. This illustrates a benefit of library-based approaches, as they may be extended to provide bindings in other languages. Rebinding and portability concerns are addressed especially well by library-based approaches that strictly maintain the opacity of shared data structures. GA lacks any method other than function calls to GA routines to access data stored in the shared memory space, thereby satisfying such a condition.

## 3.2 Language-based Approaches

Language-based approaches to DSM programming models aim to provide feature-rich programming environments with a more "natural" feel to them. They attempt to provide the programmer with strong semantic constructs that conceptualize shared

data more conveniently. Language-based approaches tend to express parallelism using simpler, more succinct syntax than is possible using serial programming languages. Most are extensions of existing serial programming languages that aim to provide a tight integration of shared data structures and operations with existing features.

Extending existing serial languages is an excellent way to provide for the library and codebase re-use offered by library-based models. It has been observed that an individual programming language is powerless without a rich standard library, and extension is a mechanism which frees the language developer from having to implement a new one. This does cause its share of integration constraints, though. New semantic constructs must work well with the existing standard library, which can limit the scope and usefulness of extensions to existing semantics and types.

It can be argued that language extensions are ill-conceived from the beginning, as they try to retrofit semantics for distributed/shared memory operations onto an inherently serial, data-private framework. At least libraries work within the function call semantics of a language, thereby preserving its outwardly serial nature. Still, the nature of the PGAS theoretical model dictates that individual nodes operate using a serial execution model and a hybrid data model. This adapts much more smoothly than other models to serial execution semantics.

### 3.2.1 UPC

Unified Parallel C [5], developed primarily by the University of California, Berkeley, George Washington University, and Michigan Tech University, is one of the most recent entries in the field of language-based PGAS implementations. On the surface, it appears to provide a sensible, flexible model for extending the C programming language [10] to program PGAS systems.

UPC extends the C language by providing a "shared" type qualifier for declaring and accessing arrays in the global address space. Applied to pointers and arrays, this type

qualifier indicates that the given address resides in the global address space. This allows the use of standard C array syntax to reference values in the global address space.

Shared-qualified pointers have another property known as the "block size". This property defines the number of continuous elements that reside on a single node. Using this property, UPC shared-qualified pointers are able to reference memory with a programmer-defined stride without special syntax. Shared arrays have a property known as the "layout qualifier," which dictates the number of elements (in the least significant dimension) that are to be stored together on a single process.

Although shared-qualified arrays reside in the global address space, UPC has a strong concept of locality. The programmer is provided with convenient routines to determine the location of portions of memory in the global address space. Global memory resident on a given node is said to have "affinity" to that node. "Affinity" can be conveniently managed by a set of user functions, and UPC even provides a work-sharing parallel loop. Users are encouraged to utilize affinity to write more efficient programs that tend to localize shared data computation where possible.

`upc_forall` is a work-sharing loop unique among PGAS implementations. It is identical to the standard C `for` loop, except that it requires an additional expression used to predicate loop execution on the local location of a global address. If this address resides on ("has affinity to") the executing node, the iteration of the loop is executed. Otherwise, it is skipped. This implicit work sharing allows compilers to dramatically improve application performance by providing the programmer with an implicit work-sharing model. The user may also provide an integer constant as opposed to an address in the global address space. In this case, the loop is scheduled on a strided basis, keyed by the provided value. While locality-based optimizations are provided by nearly all PGAS implementations, the are generally explicit and require the programmer to manage this locality alone. The `upc_forall` statement provides this functionality in a transparent manner that minimizes code alteration.

Explicit primitives for global data transfer are also provided for bulk operations. Strong locking operations are also available, as required by the PGAS model. In addition to simple "mutex" operations, semaphores are available as statements added to the C standard. Implicit synchronization is also performed where "strict references" are observed.

A significant issue with UPC programming comes from the language facilities used to implement some of its more useful (and critical) features. The ISO standard [10] provides that type qualifiers are only relevant as declarations. This makes sense because type qualifiers were initially defined to hint to the compiler how a given variable should be stored. For example the, `register` and `volatile` storage-class specifiers hint that a given variable ought to be stored in a register with or without backing memory store respectively.

While it may seem appropriate that `shared` would be equivalent in some sense to these other storage-class types, the properties of a `shared` array may often be relevant beyond the scope that declared them. Of course, if a `volatile` or `register` variable is passed as a argument for a function, the called function could not do anything based on these storage classes even if that information were available to it. On the other hand, a UPC `shared` array or pointer carries with it some useful information that could be quite useful indeed to a called procedure.

The most important piece of such information is the block size. Blocked shared arrays that are passed as arguments to functions do not retain block size data. Not only can the block size of such a pointer not be extracted, it cannot even be relied on implicitly – the pointer behaves as if it has an "indefinite" (or, more accurately, infinite) block size. Therefore, even `upc_forall` cannot be relied upon to share any work effectively. Worse, the programmer can no longer rely on the pointer accessing the shared array elements in the same order! This severely limits any advanced library authors' ability to code abstractions centered around UPC shared arrays. Not only does it deny the ability to

take advantage of the convenient optimizations made available by the language standard itself, it critically complicates authorship of workable code.

Another consequence of implementing `shared` as a storage class is that the unique and useful block size parameter must be a compile-time constant. This prohibits dynamic load balancing or scheduling, two critical components of complex parallel scientific codes. It also severely complicates development of applications that have varying needs with respect to memory usage.

The powers of these two factors combine to cripple UPC's usefulness for practical scientific computing software. Real-world scientific codes demand flexibility, abstraction, and dynamic adaptability in a parallel communication model. While dynamic memory allocation is provided in UPC, it cannot be used effectively because shared-qualified pointer block sizes must be defined statically at compile-time.

As a token example of this, assume an scientific computing programmer needed a set of parallel linear algebra routines for a computational fluid dynamics code. Of course, this would logically be implemented as a library which could be re-used and exported to other programmers in the field. Such a library would necessarily take as parameters the sizes of the argument arrays to be operated on, in addition to their addresses. These arguments would be useless in UPC since the block size cannot be known to the library unless it is edited and recompiled!

Brightwell et. al. discuss [2] a way to work around this structural failure by dynamically computing the location of desired array elements using the definitions provided by the UPC specification. This approach is effective at permitting the dynamic allocation and use of UPC shared-qualified arrays, but it has several drawbacks that, overall, cannot overcome UPC's failures.

Requiring additional syntax to address individual array elements convolutes the natural feel of UPC's syntax. It also adds three multiplications, one `div` operation, and one `mod` operation, which significantly impedes performance since it generally must be

performed for each element. Workarounds are possible, but they further convolute the syntax and complicate array access. Note that the "string" functions (`upc_memget`, etc.) cannot be used since they also depend on the compile-time constant block size information.

While these issues significantly complicate coding in UPC as an initial development language, they are less of an obstacle for UPC as a translation target. They do, however, complicate human-readability of the translated output. The addition of additional math and bookkeeping information can be exported to macros or functions, but this provides little more simplicity since parameters must then be created and properly assigned. The mathematical addressing operations also require the choice of an arbitrary UPC block size that bears no actual relationship to the underlying data distribution (§5.6.1.1).

# CHAPTER 4 Common ground for DSM APIs

The simple set of assumptions made by the PGAS model (§2.2.2) imply a strong similarity among its various implementations. More similarities arise when it is considered that most such implementations are either available as libraries for the C programming language, or as an extension to the C programming language. Therefore, considering C-based implementations is not restrictive and yields a reasonable cross-section of similar APIs to analyze.

In addition, restricting our analysis to C-based APIs allows us to more readily study the similarities and differences among them by studying their deviations from standard, serial C. Syntactic and semantic alterations are convenient to compare, as well as library semantics. For simplicity, let us consider library-based implementations as if they were language-based implementations. Given that our analysis is based on the same source language, a library-based implementation is equivalent to a language-based implementation that adds to vocabulary and semantics but not syntax. Furthermore, ideas developed here should be adaptable to other procedural programming languages (such as Fortran) and parallel extensions for them (such as Co-Array Fortran [13]).

Intuitively, it seems quite likely that any two models derived from the C language could be translated to each other. Obviously, any library-based implementation is implemented in "standard, serial C" itself and can thus be considered to translate to it trivially. It also seems likely that, given the alterations to C by even a language-based and a library-based model, any such API could be translated to any of the others. This intuition is further bolstered by the simple set of assumptions made by the DSM model

for which these extensions are designed.

There also appears to be a functional equivalence here – any sufficiently simple idiom expressed in one model can easily be expressed in another. For example, let us select two models which appear on the surface to diverge greatly in terms of semantic equivalence. Global Arrays (§3.1.2) is a library-based model that maintains an opaque view of distributed data types. UPC (§3.2.1), is a language-based model that integrates shared pointers (at least, to some extent) into its type system. These two seem sufficiently dissimilar to serve as examples in this context.

For comparison purposes, let us lay out GA's characteristics as defined by a DSM API's critical properties listed in section 2.3.

- Familiarity: GA's syntactic model is no different from the standard C syntactic model, as it is a library-based implementation. While not strictly part of C's type system, GA's shared array types are opaquely controlled by the library and presented to the user in the form of multidimensional, indexable arrays. Data transfer is handled using the buffer mapping mechanism mentioned in section 3.1.2. While this differs from the standard C array model, C programmers (especially those who use high-performance computing techniques) are generally familiar with the concept of linearizing multidimensional arrays.

- Paradigmatic simplicity: GA's programming paradigm is based around shared arrays that are divided into "chunks" across the participating processes. Arrays are indexed by arrays of integers corresponding to the rank in each dimension of the indexed element. Data transfer operations are conducted by creating arrays corresponding to the limit elements to be operated upon, along with arrays to describe the local buffer to be used for data transfer. Optimized collective operations are also available to GA programmers. Most of these collective operations generally consider the global arrays to be n-dimensional arrays for their purposes.

These collectives simplify GA programming greatly in situations where they apply. More complex data transfer operations are provided, such as strided and diagonal transfer operations.

- Transparency: As it is a library-based model, GA's methods for global memory access are function calls. These function calls transfer data into local arrays, however, so the data are then accessed according to C's memory access model (pointers and arrays) in local address space.

- Exposure: In nearly all cases, the user is free to ignore the actual data distribution when conducting data transfer. To improve performance, however, the user may access the data distribution by either determining the location of an element or querying the library for the range of elements stored on a given node. The user may also consider the data distribution to be used upon the creation of a new shared array. A node may even determine its location in a virtual processor grid based on the portion of a given global array it stores.

UPC's characteristics along the same lines seem to be:

- Familiarity: UPC extends the standard C syntax in several ways. Those that generally pertain to global data transfer are in the form of shared declarations. While UPC pointers have slightly different declaration syntax, their usage follows standard C syntax. As a result, most common C array operations have exactly the same syntax in UPC. UPC offers some useful extensions whose syntax and semantics are very similar to those in standard, serial C. Its data-parallel loop intuitively extends C's `for` loop with an extra argument that localizes each iteration, ensuring computation and data are in the same place.

- Paradigmatic simplicity: UPC's programming paradigm introduces the concept of "affinity" of shared data, or the node on which a given set of global memory

is stored. Managing affinity is not strictly required if shared arrays are accessed directly. When shared pointers are involved, however, the user is burdened with manual affinity management. Shared pointers must have a compile-time constant "block size," which determines the order in which elements in shared arrays are accessed. This increases the complexity involved in dealing with shared arrays dramatically.

- Transparency: Global memory references have syntax that is the same as local array references. Therefore, global and local memory accesses are almost indistinguishable from each other. As a result, UPC satisfies this property remarkably well.

- Exposure: As mentioned above, if certain features of UPC are avoided, the location of global memory data may be safely ignored. Dealing with UPC "affinity," however, can improve application performance. The affinity of any address in the shared memory space can be determined easily via the `upc_threadof` expression or it can be easily computed using the definitions provided in the UPC specification. A node may obtain a pointer to its local portion of a block of shared memory with a simple typecast to a local pointer type. This integration with C's type system makes such transitions very convenient. Using the `upc_forall` loop, applications may guarantee the fastest possible global memory access without explicitly checking the "affinity" of any portion of the shared data at all.

The assumptions made by DSM APIs are satisfied by GA as follows:

1. *The system is composed of a set of interconnected nodes.* GA runs above an existing message transport layer, often MPI [11]. As such, it takes on the concept of nodes as its message transport layer does. It passes an abstraction of this idea along to application programmers as a linear arrangement of integer process ID's.

2. *Each node contains some amount of local memory.* As a library implementation, local memory concepts are not altered.

3. *From a portion of memory on each node, a globally addressable, shared memory is formed.* GA delivers global memory that is addressable in the form of arrays keyed by integers returned from the API's allocation functions. Because the allocation functions are collective operations called simultaneously by all nodes, they are able to return the same array key to each process, which guarantees global accessibility. In addition, as mentioned above, GA allows a participating node to determine the portion of its local memory used for global memory space and read and write to these regions.

4. Each node supports the following operations using function calls and/or language constructs:

   (a) *Atomic copying from local memory to shared memory.* This is supported through `NGA_put` and its variants.

   (b) *Atomic copying from shared memory to local memory.* This is supported through `NGA_get` and its variants.

   (c) *Obtaining an identifier unique to each node.* This is supported through `GA_Nodeid`.

   (d) Global barriers. This is supported through `GA_Sync`, with finer-grained operations available through fence operations.

   (e) Basic mutex operations. This is supported through `GA_Lock` and `GA_Unlock`.

Now, we observe how UPC satisfies these properties:

1. *The system is composed of a set of interconnected nodes.* These interconnected nodes are conceptualized as "threads" and depend on the runtime environment

used to implement them. The UPC specification simply requires that the number of threads used in the program be either defined at compile time (enabling additional features) or at runtime, with conforming implementations supporting both options. It does not require that these threads execute on the same system.

2. *Each node contains some amount of local memory.* UPC does not alter standard C's local memory model.

3. *From a portion of memory on each node, a globally addressable, shared memory is formed.* This shared memory may be either statically allocated (via declarations of shared arrays) or dynamically allocated using functions available in the UPC standard library. Localization of global memory is done through expression extensions or mathematical computation.

4. Each node supports the following operations using function calls and/or language constructs:

   (a) *Atomic copying from local memory to shared memory.* This is supported through assignment from local variables to shared address destinations or the `upc_memput` routine.

   (b) *Atomic copying from shared memory to local memory.* This is supported through assignment from shared addresses to local address destinations or the `upc_memget` routine.

   (c) *Obtaining an identifier unique to each node.* This is supported through the constant `MYTHREAD`.

   (d) Global barriers. This is supported through `upc_barrier`.

   (e) Basic mutex operations. This is supported through `upc_lock` and `upc_unlock`.

Observe that there is a virtual one-to-one correspondence between the way each model satisfies the requirements of a conforming PGAS implementation. The equivalence of these models may be proven conclusively by constructing a source-to-source translator capable of rendering any program written in one model to a program written in the other. The existence of such a program proves that there is a strong, complete equivalence between these models and a notion of similarity among other DSM models.

# CHAPTER 5 A translator for two DSM APIs

To establish the equivalence we seek, we treat each subsystem of the constructed translator here in detail. A code translator is in many ways equivalent to a compiler whose output language is another source language. Therefore, we will treat each component in the traditional order used to explain compiler design. The translator implemented here was formed from the ctool [7] project, a C-to-C translator implemented in C++. Therefore, code examples from the translator will be in the C++ programming language. C++ provides unique opportunities for such a translations system, such as the possibility to implement an extensible framework for many-to-many DSM API translation (Ch. 6).

## 5.1 Translator structure

The translator program was created initially from the ctool project, C translator whose code generator simply outputs C code functionally identical to its input. It is structured like a typical compiler. The translator extends this structure to enable its additional functionality. It does so by requiring a stronger relationship between the lexer and parser for symbol table maintenance, and by adding an additional AST modification step (§5.4) before the code generation step. The final structure is given in figure 5.1 [1].

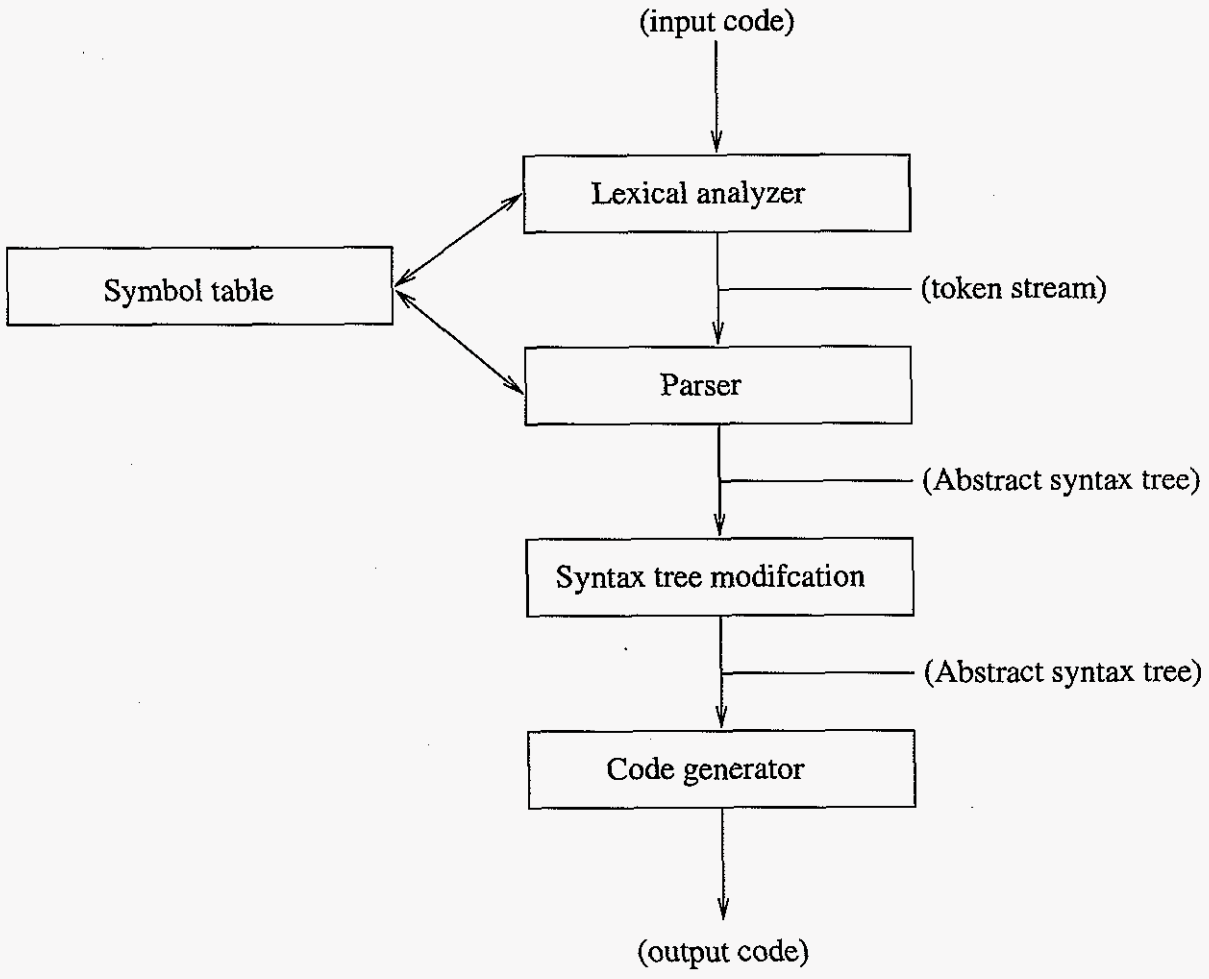


Figure 5.1 Translator structure

## 5.2 Lexical analysis

Ctool's lexical analyzer is implemented using the GNU flex [14] lexical analyzer generator. Because we treat GA as a language and not as a library here, it is necessary to recognize the names of GA function calls as individual tokens. This is simple enough. Of course, UPC also adds a number of new tokens to the language, which are incorporated here as well.

Because ctool builds the C symbol table on the fly, a C lexical analyzer must interact

with the symbol table as constructed by the parser (see figure 5.1). UPC adds an additional layer of symbol analysis, as it is necessary to keep track of the symbols that are indicated to be `shared`. `shared` symbols have different semantics, so they require special handling by the parser. Thus, the lexer and parser must work together to maintain sufficient context to identify shared symbols as they are encountered. When a shared context is entered, the parser advises the lexer by setting a global flag that the lexer checks upon encountering a new symbol. Upon encountering a symbol in this context, the lexer flags the corresponding symbol table entry as shared.

## 5.3 Parsing

Ctool's parser is implemented using the GNU bison parser generator [4]. The bison input file is generally known as a "grammar file" and adopts a convenient syntax for direct grammar specification. The basic grammar used in ctool is due to Harbison and Steele [9]. Reduction rules in the grammar are paired with C++ program code to compute their semantic values from the values presented it by the lexer or by levels lower down the parse tree. Therefore, `bison` parsers are bottom-up parsers.

As extensions to grammar rules are elaborated here, existing rules (and alternatives) will be omitted for brevity. The complete ruleset is listed in: (appendix?)

### 5.3.1 Parsing extensions for GA

GA's addition to C's token set requires that a standard C parser be extended with new grammar rules to handle them and react appropriately. Because each GA operation was previously equivalent to a function call, the function call-related syntax rules were changed to accommodate them. As depicted below, these extensions simply combine these former function calls into new rules for new, unique types of expressions. This permits the translator to treat each operation individually and react appropriately, al-

tering other parts of the parse tree if necessary. Avoiding specific parsing for these operations may also limit our options later, as it restricts the translators "knowledge" of the input code, thereby restricting the flexibility of the output rendering.

GA-based extensions to the grammar begin at the `postfix_expr` rule where the `ga_call` rule is created to represent them. For example, the grammar rule for `GA_Create` is depicted in figure 5.2.

```
ga_call : G_CREATE LPAREN assign_expr COMMA assign_expr COMMA
          assign_expr COMMA assign_expr COMMA assign_expr RPAREN;
```

Figure 5.2 Modified `ga_call` rule

This is parsed into the newly created `GACreateExpr`, which corresponds to the existing `Expression` class. Like the other subclasses of `Expression`, code generation is handled through `GACreateExpr`'s `print` method (§5.6.1.1).

### 5.3.2 Parsing extensions for UPC

Obviously, a language-based DSM implementation will require grammar modifications to accommodate it. The UPC specification conveniently describes in detail the extensions to the standard C grammar that it provides. These extensions give rise to additional grammar extensions that permit smoother code generation. Several semantic actions in the modified grammar involve the creation and manipulation of additional abstract syntax trees that are attached to the global tree for additional functionality. These "synthetic trees," as they will be called, will be noted when they are used by designating that rules (or rule parameterizations, see below) "rewrite" portions of the abstract syntax tree.

UPC extends C's grammar in several places. The simplest is its advanced `sizeof` expressions that apply to shared-qualified types. `upc_elemsizeof`, `upc_blocksizeof`,

upc_localsizeof and upc_threadof fit squarely within the existing grammar construct for the C sizeof operator. The extensions are depicted in figure 5.3.

```
unary_expr : upc_esizeof_expr
           | upc_bsizeof_expr
           | upc_localsizeof_expr
           | upc_threadof_expr;
```

Figure 5.3 Extensions to the unary_expr grammar rule

Of course, for a type to be shared-qualified, there must exist a type qualifier named shared. Additionally, shared-qualified types may be parameterized by a block size (in the case of a shared pointer) or a layout qualifier (in the case of a shared array). Thus, the combined extensions to C's type system give rise to the extensions depicted in figure 5.4.

```
                shared_token : SHARED
                             | STRICT SHARED
                             | RELAXED SHARED;
                 shared_decl : shared_token
                             | shared_token LBRCKT STAR RBRCKT
                             | shared_token LBRCKT opt_const_expr RBRCKT;
opt_decl_specs_reentrance : shared_decl opt_decl_specs_reentrance;
```

Figure 5.4 Extensions to grammar rules accommodating shared declarations

It is worth noting that there are several consequences to account for when the parser encounters a shared-qualified variable declaration. Because GA is a library-based model, simple static declarations of GA types are not possible. Therefore, code must be added to enable this to occur. A straightforward translation here is not always possible, since

function calls in C are not permitted except as initializers. The translator must determine in a second pass where shared-qualified array initialization code should be placed and utility code must be written to account for this.

A simple way to handle several extensions that follow C's syntax but differ in semantics is to check the C++ runtime type information (RTTI) contained within the subclasses of `Expression` created to handle UPC extensions [15]. This preserves the simplicity of the existing grammar (to the extent it can be considered simple to begin with) and limits complications that can arise from too many explicit extensions. We will note this where appropriate as in figure 5.5.

```
predec_expr : DECR unary_expr [ UPCArrayExpr, UPCVariable ];
```

Figure 5.5 Example of node type extension

This indicates that the semantic action for the `predec_expr` rule is parameterized by the actual type of the semantic value of the incoming `unary_expr` rule.

Also, see (§5.4) for an explanation of the hooks provided to schedule abstract syntax tree alteration after main parsing is complete. The scheduling events that occur during parsing will be noted as in figure 5.6.

`declaration : decl_specs opt_init_decl_list`[*modifiesCode*, *Header*, *Exec*];

Figure 5.6 Example of scheduled AST modification notation

`upc_forall`, the data-parallel loop extends the `for` loop syntax and therefore the C grammar for loop statements. Its optional "affinity" parameter is allowed to be one of several already existing C statements, as shown in figure 5.7

Because the semantics of shared-qualified types are somewhat different than for their local counterparts, the notion of these types gives rise to several extensions to the C

```
affinity : opt_expr
         | CONT;
```

Figure 5.7 The `upc_forall` affinity statement rule

grammar to deliver the proper semantics when necessary. For simplicity, they will be called "utility extensions" from this point. A UPC shared-qualified pointer consists of three main components: [5] a global address, an offset from that address, and a phase component that tracks the pointer's current location within its current block. This can conveniently be represented in other models by a simple C `struct`. In most cases, taking the address of a UPC shared-qualified datatype also follows the same semantics, however, there are cases (such as the parameter to `upc_forall`, where the address of a shared-qualified type is used to implicitly determine one of these pieces of information. A translator must therefore account for this in the grammar. Therefore, we have the construct in figure 5.8.

```
addr_expr : B_AND cast_expr [ UPCArrayExpr, UPCVariable ];
```

Figure 5.8 Extensions to `addr_expr` rule for UPC

Dereferencing a shared pointer obviously has markedly different semantics than dereferencing a local pointer. The referenced global address (offset by the pointer's own offset value) must be located and appropriate data transferred. Dereferencing of shared pointers (or the indexing of shared arrays) must therefore be tracked to as to process this translation where necessary. This involves the extensions in figure 5.9.

Likewise, writing to a shared pointer or array element must be handled separately from writing to a local array or pointer. This gives rise to extensions analogous to those given in figure 5.9, leading us to figure 5.10.

```
indirection_expr : STAR cast_expr [ UPCVariable ];
```

Figure 5.9 Extensions to `indirection_expr` rule for UPC

```
assign_expr : cond_expr
            | unary_expr assign_op assign_expr [ UPCArrayExpr, UPCPtrExpr ];
```

Figure 5.10 Extensions to `assign_expr` rule for UPC

C has several "utility assignment" operators which read a value, operate on it, and then write it back to where it came from. These operators include: `++`, `--`, `+=`, `/=`, `*=`, `-=`, etc. They also require special handling in much the same way as those mentioned previously, however, there is an additional complication that must be addressed. These operators imply the creation of temporary values to hold the results of the computation. A processor normally handles this condition gracefully (using its register set) in serial cases, however, the shared-qualified cases here present an extra difficulty. The parser, upon detecting this type of expression, references a temporary variable to be created after code generation for this purpose (§5.4). The creation of these temporary variables is triggered by the declaration of a shared-qualified type. Most of the assignment operators are subsumed in the `assign_op` rule which is then referenced by the `assign_expr` rule listed above. The preincrement and postincrement operators have their own rules. They are extended as in figure 5.11.

Pointer arithmetic is also slightly different with respect to shared-qualified UPC types. Extensions to the C grammar to handle it properly come from the same roots as those for assignment operators, giving us the `UPCVariable` extensions mentioned above.

Finally, UPC permits the user to typecast a shared-qualified pointer to a local ("private") pointer to directly access the locally stored portion of a shared-qualified memory

```
preinc_expr : INCR unary_expr [ UPCArrayExpr, UPCVariable ];

predec_expr : DECR unary_expr [ UPCArrayExpr, UPCVariable ];

postinc_expr : postfix_expr INCR [ UPCVariable (rewrites), UPCIndexExpr (rewrites)

postdec_expr : postfix_expr DECR [ UPCVariable (rewrites), UPCIndexExpr (rewrites)
```

Figure 5.11 Modified rules for assignment expressions

space. This requires some rather significant manipulation of the underlying GA data for how syntactically simple an operation it is. Therefore, care must be taken to perform this function as efficiently as possible. There are also coherency issues created in this situation (§5.6.2.11). Gramatically, this situation is very similar to the serial case; it is presented in figure 5.12.

```
cast_expr : unary_expr
          | LPAREN type_name RPAREN cast_expr [ UPCVariable ];
```

Figure 5.12 Extensions to `cast_expr` rule for UPC

## 5.4 Abstract syntax tree alteration

Hooks are provided in the translator to schedule alterations to the abstract syntax tree (AST) once it has been generated. The AST for the translation unit containing the `main` function is special, and modifications to it may even be requested by semantic actions from other translation units. Because of this, the presence of a translation unit

containing `main` is required in order to process any code. This limitation could be overcome by generating a separate translation unit to contain these modifications that may be saved in source form (after a pass through the code generator) and referenced by `main` once it has been processed (Ch. 6). Currently, these features are used only by the UPC translator.

Three types of hooks are present to manipulate three special translation units created by the translator after the first pass has completed. Two are generated directly as pure source. They are not created dynamically and passed to the code generator since the output code is known to the parser at the state where it is able to trigger the hook. These include the "header" unit and the "code" unit.

The "header" unit corresponds to a header file that will be generated for inclusion in all translated source files. It contains some default content which is output by the translator along with the code attached to it by hooks from the parser. The default content includes the declarations of the implementations for `upc_elemsizeof`, `upc_blocksizeof`, `upc_localsizeof`, `upc_threadof`, `upc_all_alloc`, `upc_lock_alloc`, `upc_lock`, `upc_unlock`. It also includes the definitions for the type extensions (such as UPC pointers), and function declarations for any code attached to the "code" synthetic translation unit.

The "code" synthetic translation unit consists of additional utility functions required whose definitions cannot be known entirely before translation. This includes functions for accessing UPC shared-qualified arrays, whose number of dimensions cannot be known until their declarations are parsed. Therefore, the number of parameters necessary to access them cannot be known *a priori*. As noted above (§5.3), the parser attaches code to this hook upon discovering a shared array declaration, thereby allowing the code generator (§5.6) to assume these functions exist when outputting relevant code.

The "exec" synthetic translation unit is generated dynamically so that it may be grafted onto the AST for the `main` function before it is passed to the code generator.

Code in the "exec" tree is attached after `main`'s primary declaration statements, before the first statement is executed. As a result, variables in `main` cannot have initializers that depend on the execution of this code. Workarounds (Ch. 6) could include either altering `main`'s AST further to move initializers to a point after this code is attached, or generating a new `main` altogether and renaming the user's `main` function to a function that is executed by code in this translation unit directly.

## 5.5 AST structure

After parsing, an AST remains that is quite similar to one generated from an ordinary, serial C program. Here is a brief treatment of the additional nodes that arise from the translator code.

### 5.5.1 Nodes added for GA

Each translated GA function call has a corresponding AST node added to the standard set to represent it. These nodes have similar names and all subclass (either directly or indirectly) the `Expression` class, corresponding to the AST node of the same name. From now on, we refer to translator C++ classes from AST code and AST nodes interchangeably, as each AST node is implemented as a C++ class.

`Expression` has a `virtual print` method, which enables its subclasses to output the appropriate code during the code generation phase (§5.6). These subclasses act in place of other `Expression` nodes with no additional side effects. Unlike some of their UPC counterparts, GA nodes all arise from unique syntax rules, and RTTI is not generally used to distinguish them. A benefit of treating each call as a unique AST node, however, is that the translator could be extended to use this method in the future should the need or utility arise (Ch. 6).

A special node is created for all MPI expressions detected in GA code. This node,

called `MPIExpr` is generic, not remembering the syntax rule that generated it. It is used as a placeholder for the few MPI calls that tend to be present in typical GA codes, which use MPI as a transport layer. Although explicit calls to MPI code are discouraged, it is necessary to start and stop MPI through the `MPI_Init` and `MPI_Finalize` calls.

### 5.5.2 Nodes added for UPC

Several new nodes are introduced to handle UPC code. Many of these nodes are subclassed from existing serial C node types, and even "masquerade" as their serial C counterparts at many points in the parse tree, identified only by their C++ RTTI where relevant. That is, they are not the result of their own semantic reductions by the parser as are ordinary nodes. This greatly simplifies the grammar in cases where the semantic actions to be taken are not significantly different and are shared with C counterparts most of the time. Many of the semantic actions corresponding to these nodes are merely side-effects that do not necessarily merit their own syntax rules.

Unlike the nodes added for GA support, supporting UPC translation requires that nodes to represent UPC's extension to the C type system be implemented. Nodes to represent UPC variable types (`UPCBaseType`, `UPCArrayType`), declarations (`UPCArrayDecl`), and semantics (`UPCPtrMathExpr`) are all necessary for correct translation.

Because many of the operations performed by UPC function calls are quite simple, some such calls are implemented by dynamically creating a small subtree using existing AST nodes and grafting it to the primary AST. This avoids an overabundance of new nodes for simple operations.

## 5.6 Code Generation

Of course, for each new expression and statement type we create, an appropriate backend output must be defined that corresponds with it. This phase contains most of

the complexity in the GA translator, whose extensions to the C grammar are relatively light.

### 5.6.1 Code generation for GA

The execution of GA code in UPC is quite complex. Because GA maintains so much bookkeeping information about all global data, this information must be synthesized properly using GA code. Most of this bookkeeping (or at least, how to do it) can be known to the translator *a priori*, from the nature of the GA calls to be emulated. Therefore, a large block of auxiliary code accompanies the translator that is not dynamically generated at translation time.

This code is widely referenced by the code output from the translator's code generator, so a brief treatment is warranted here. Because of the way UPC arrays must be allocated by the output code (§5.6.1.3), utility functions are provided to linearize and localize individual GA elements within a dynamically allocated UPC block. These functions can exist *a priori* as a result of the arbitrarily chosen static block size. In fact, the implementation of this choice is contained within the auxiliary code.

Functions also exist to provide access to the bookkeeping information kept as a result of a user call to `NGA_Create`. Using a utility function interface to this data permits the underlying data storage and reference implementations to change in the future with a minimum of code alteration. All *a priori* auxiliary code access all bookkeeping information in this way. Some queries possible in GA are simple enough that they are not given individual auxiliary functions, instead calling one or more query functions directly.

Using these utility functions, generation of code from the input AST is fairly straightforward. Many output rules contain only code output references to the auxiliary routines that accompany the translator code.

#### 5.6.1.1 GACreateExpr and GADupExpr

GACreateExpr subclasses Expression to form the translation for the NGA_Create call. This calls the appropriate auxiliary code, which allocates space for the data and computes the relevant bookkeeping information. A UPC pointer is then returned to the user using the upc_all_alloc function. GADupExpr copies this information and returns a pointer to the copy. This bookkeeping information includes data necessary to linearize and locate arbitrary elements within the array space and type and state information to optimize queries.

Of course, upc_all_alloc cannot conveniently allocate space suitable for multidimensional array addressing. Thus, bookkeeping information must be used to determine the location of referenced elements. Furthermore, since UPC pointer block size information cannot persist across function calls, a globally fixed block size must be used and mapped to an appropriate block size as indicated by the "chunk" parameter to NGA_Create. This block size was arbitrarily chosen to be 8 bytes, sufficient to hold a single C double variable on most 32-bit architectures, including the IA-32 architecture.

#### 5.6.1.2 GADestroyExpr

As its name implies, the GADestroyExpr node outputs code that deallocates the array itself and the associated bookkeeping data. Although this is defined to be a collective operation in GA, the dynamic lifetime of a UPC array is defined only until any thread calls upc_free. A second thread reaching this call causes an error, because it has attempted to deallocate an object outside its dynamic lifetime. Therefore, it corresponds well to a collective operation anyway, as the threads must complete all outstanding references to the global data and exactly one thread must deallocate it. The outstanding references are completed via a call to upc_barrier and thread 0 is arbitrarily chosen to deallocate the array. Because array destruction is a collective

operation in GA, we know in advance that it will be called by all processes accordingly.

#### 5.6.1.3 `GAGetExpr` and `GAPutExpr`

The `GAGetExpr` and `GAPutExpr` nodes are, of course, a node critical to the operation of the system. Their output translations make use of the bookkeeping data created by the auxiliary code called by the output from the `GACreateExpr` node. Their main input parameters are a pair of integer arrays corresponding to the boundary limits in the global array space to be accessed and transferred to or from user code.

Localizing a single element of a multidimensional global array requires several steps, some of them rather computationally expensive when compared to a standard, single memory access. First, the source element must be *linearized*, that is, it must be mapped into the one-dimensional space provided by the initial call to `upc_all_alloc`. Static UPC array declarations cannot be used, because it can not be guaranteed that the translator can know the dimensions (or even the number of dimensions) of any of the global arrays at translation time. This linearization requires time linear in the number of array dimensions. A future optimization could be a mechanism (perhaps utilizing C's `#pragma` for the programmer to hint to the compiler when the entire dimension set of a global array can be known *a priori* (Ch. 6), which would allow direct localization in constant time.

Once an element has been linearized, the location of that element in the UPC dynamically allocated array must be determined. Block size information for the pointer used to allocate the data cannot be stored and referenced later, because UPC requires that blocked pointers have a compile-time constant block size. Thus, even though the block size can be computed at runtime, it cannot be used directly to determine the location of a given element. This is in direct conflict with GA, which permits the user to compute the contents of the "chunk" parameter to `NGA_Create` on the fly.

"Chunk" data is among the bookkeeping information kept by the translator's auxil-

iary code. This information may be used to locate individual elements by use of some clever mathematical computations (due to [2]) from the way UPC implementations are required to store blocked data [5]. Unfortunately, these mathematics involve expensive "div" (integer division) and "mod" (modulus) operations. Although these are constant-time operations (not depending on the element location or the array size), they are nonetheless considerably more expensive than serial array access. To streamline this localization process (and avoid having to perform these computations for each individual array element), the auxiliary code provides a primitive address caching mechanism useful for continuous data transfer of the like supported by the GA model. This cuts down somewhat on the number of computations necessary to transfer each individual element.

A consequence of this method is that the UPC pointers actually computed and used for data localization cannot be relied upon to automatically "jump" to memory located on another node where necessary and there is no guaranteed correspondence between a pointer to the last byte of data on one node and a pointer to the first byte of data in the next. Consequently, no caching method can be guaranteed to provide reliable data across UPC "affinity" bounds. This limits the continuous number of elements whose localization can benefit from the provided caching optimizations.

#### 5.6.1.4 `GAZeroExpr`

The `GAZeroExpr` node corresponds to the `GA_Zero` function call. This call sets each element of the target array to 0. This call is unique in that little bookkeeping information is required to achieve this objective. Owing to the fact that the representation of 0 for the C types `int` and `double` are similar (except for length), the output code may simply iterate over all bytes stored in the global address space and set them to 0.

This allows the use of UPC's `upc_forall` data parallel loop (§3.2.1), ignoring type and distribution information. This guarantees that the loop will execute optimally

(for a given UPC implementation), without regard to precisely what data is located where. Since even type information may be safely ignored, this results in a clean, simple operation.

#### 5.6.1.5 GAAddExpr

The GAAddExpr node corresponds to the GA_Add function call. This call adds a constant value to each element in the global array. Like GAZeroExpr, the code for GAAddExpr need not be aware of the precise distribution of the global array data. It does, however, need to use underlying type information since addition must be performed.

The use of this type information requires the static declaration of properly-typed UPC pointers to access the global data. Here, we exploit the arbitrarily chosen block size to allocate pointers with the appropriate block size to access single elements. While using a unit (for the proper type) block size results in out-of-order access in almost all cases, this is unimportant since the GA_Add function call adds element-wise.

upc_forall is used with statically declared pointers to carry out this operation. The extra time necessary to initialize the static pointers based on type information is negligible, so this operation can also be regarded as optimal for a given UPC implementation.

#### 5.6.1.6 GAAccExpr

The GAAccExpr node corresponds to the GA_Acc function call. This call adds each element of the target portion of a global array with the corresponding element of a user-provided input buffer, scaled by an arbitrary input constant. While fundamentally similar to the two operations detailed above, the GAAccExpr must output code that is concerned with the proper addressing of the global array elements.

This restriction prohibits the simple use of upc_forall. To do so would involve an inverse mapping for each element in the global array, a computationally expensive prospect (requiring linear time per element). Also, there is no guarantee that the range

of indices provided by the user correspond to contiguous sections of globally-allocated memory, making the `upc_forall` loop impractical for this purpose. While `upc_forall` could possibly be used here, the benefits of doing so are unclear (Ch. 6).

#### 5.6.1.7 `GADdotExpr`

The `GADdotExpr` node corresponds to the `GA_Ddot` function call. This call computes the element-wise dot product of two input global arrays and returns this value to the user. This call requires that the input arrays have the same type and dimensions, so this can be regarded as a global reduction operation.

GA assumes that the input arrays may have different distributions. The output code for this node assumes that the input arrays have the same distribution to allow this operation to be accomplished using `upc_forall`. This restriction could be removed later by testing this condition or reverting to the access model used by the output code for `NGA_Get` and `NGA_Put`.

The `upc_forall` used here uses a statically declared pointer with unit block size for the data type just as `GAAddExpr` does.

#### 5.6.1.8 `MPIExpr`

All MPI expressions detected within GA code are generalized into the `MPIExpr` generic node. This node is created from multiple rules in the input grammar, each specifying a unique MPI function call. As a result, only MPI calls specifically detected by the translator are detected. As mentioned earlier, explicit MPI calls in GA code are discouraged, but they are permitted. A consequence of this handling is that unanticipated MPI calls must be prohibited, and those that are anticipated must have an expected effect (or an effect rendered moot by translation, since MPI is no longer assumed to be the underlying message transport layer).

This node always outputs the symbolic constant `MPI_SUCCESS`, indicating for purposes of client code error checking, that the detected call had the desired effect. This reinforces the limitations mentioned above (Ch. 6).

#### 5.6.1.9 `GAInqExpr`, `GANidExpr`, `GANnoExpr`, and `GADstExpr`

These are all GA queries for bookkeeping information kept by the library. Their AST nodes all output trivial code that locates the proper information (originally created by the output code for `GACreateExpr`) and returns it to the user.

### 5.6.2 Code generation for UPC

Generating GA code from the UPC input AST depends strongly on the auxiliary code generated in the parsing phase (§5.5). In several cases, intermediate function calls are needed, which necessitate the use of the C comma operator. Intermediate operations are carried out in the "left" portions of a string of comma-delimited expressions, and the desired result is used at the right. This syntactic construct will be noted when it is used.

#### 5.6.2.1 `UPCBaseType`

The `UPCBaseType` node is subclassed from ctool's serial C `BaseType` node. The `BaseType` node is used to represent the most simple C type of a variable (`int`, `long`, `double`, `struct`, etc.). This extension tracks the layout qualifier (or block size, in the case of pointers) that may be attached to UPC types. The type of this parameter is defined to be an `Expression`, which is tantamount to an extension to standard UPC [5].

This node also serves to identify to the translator when extended types, such as arrays, are known to be `shared`. Since this is a subclass of `BaseType` rather than a type qualifier (as dictated by standard UPC), this constitutes another extension with implications for extended type checking and parameter passing. At present, these features

have not been leveraged. One notable feature of `UPCBaseType` is that its code output methods are never called by array-typed variables (§5.6.2.3). Therefore, its code output can be concerned only with output for pointer variables (since shared automatic variables are prohibited by UPC 1.1). Since all shared pointers are represented by a single `struct` type defined in the auxiliary headers, the output code for this type is simply the name of this `struct`.

#### 5.6.2.2 `UPCVariable`

The `UPCVariable` node, which subclasses `Variable` (and thus `Expression`) exists to differentiate between expressions that contain UPC pointer or array references and those that do not. Its inherits its `print` method from `Variable`, so its output code is identical to that for a `Variable`. This is possible because the `UPCBaseType` class takes care of printing the type information properly.

#### 5.6.2.3 `UPCArrayDecl`

The `UPCArrayDecl` node is a direct subclass of `Decl`. It is instantiated whenever the translator detects a complete declaration whose corresponding entry in the symbol table indicates that it is for a shared array. Instantiation of a `UPCArrayDecl` has several critical side-effects with respect to code generation.

These side effects include modification of each of the special translation units generated by the translator (§5.4). An AST containing a single block of code is synthesized and attached to the "exec" unit using the hooks provided by the translator. This block contains an AST representing code to configure and initialize the declared array. A temporary array is created to hold the dimensions of the declared array for passing to GA. The expressions for the sizes of each dimension of the array are written out to be evaluated when the "exec" unit code executes. This constitutes an extension from standard UPC.

Three important functions are created to access the declared array. Since this is the earliest time at which the number of dimensions in the array can be known to the translator, this is the earliest point at which the exact content of these functions can be known. Another result of this situation is that each of these functions must be individualized – single versions of each of them cannot be created *a priori*.

The "getter" function takes one parameter for each dimension of the target array. It passes these parameters to `NGA_Get` and returns the result element. The "setter" function takes one parameter for each array dimension and one parameter for an updated value. It uses `NGA_Put` to write the value to the global address space. For the convenience of the translator (as the "setter" function is used in assignment statements), the "setter" function returns the updated value. Finally, an "indirector" is created to handle the dynamic creation of a UPC pointer from the target array. It takes the same arguments as the "getter" function. These function definitions are attached to the "code" translation unit and declarations for them are passed to the "header" translation unit. These hooks are processed after translation of all input files is complete.

Because all global arrays are initialized by the execution of the code generated from the "exec" hook, they must be visible both in the scope of their declarations and the execution scope of the "exec" hook's output code. This is accomplished by placing this declaration in the "header" translation unit where it will be globally visible. As a result, all declared UPC arrays must have globally unique names. This is not restrictive, since static UPC arrays must be globally declared anyway. In fact, permitting local declaration of static UPC arrays constitutes an extension to the standard.

After completing these side effects, there is actually nothing left for `UPCArrayDecl` to do. It does not directly output any code save for a comment to note to the programmer that a UPC global array declaration was processed successfully.

#### 5.6.2.4 UPCArrayExpr

The UPCArrayExpr node is one of two subclasses of UPCIndexExpr (which is never itself used as a node) for locating and returning individual global array elements. It relies upon the generated auxiliary code created when the declaration for its target array was parsed.

The output code for UPCArrayExpr calls the target array's "getter" function with an argument for each dimension in the array, corresponding to the subscript expressions specified by the UPC client code. Thus, the standard C semantics related to multidimensional arrays are slightly subverted here. This is not restrictive, because the translator can output different code if less subscripts are specified than the number of dimensions in the array.

#### 5.6.2.5 UPCPtrExpr

The UPCPtrExpr node is the other subclass of UPCIndexExpr. It handles dereferencing UPC pointers by subscripting.

The UPCBaseType node's code output guarantees that any declared UPC pointer will have the type ptr_t (the struct created to support them) in the output code. Therefore, we can use the *a priori* auxiliary functions created by the translator here to operate on the target pointer. The generic pointer "get" method is called to access the subscripted value using NGA_Get.

The UPCPtrExpr node only supports the use of a single subscript, owing to the C pointer semantics pertaining to multidimensional arrays. A pointer to a two-dimensional array, for example, can simply be represented as a pointer to a one-dimensional array of the struct type of translated pointers. Therefore, no special code is necessary to handle this case.

#### 5.6.2.6 UPCPtrMathExpr

The UPCPtrMathExpr node handles pointer arithmetic with respect to UPC pointers to shared data types. The struct used to represent a UPC pointer to a shared array includes a set of indices that track precisely which element in the target array the pointer currently refers to. The UPCPtrMathExpr node outputs a reference to auxiliary code that updates this data, moving the pointer forward (or backward, as required) to the proper location in the array.

A pointer that runs past any dimension in its target array is automatically reassigned by incrementing (or decrementing) its next-most significant dimension and continuing until the end (or beginning) of the array is reached. Bounds checking is performed, so pointers that travel entirely outside the bounds of their target arrays cause a warning message to be printed and the entire program to be terminated. Future work may attempt to provide a simpler and less catastrophic consequence for this error.

#### 5.6.2.7 UPCAssignExpr

The UPCAssignExpr node handles assignment to specific location in a global array. It subclasses AssignExpr for this purpose. UPC pointer assignments are handled by functions in the static auxiliary code added to the "exec" tree (with declarations in the "header" tree).

Unlike AssignExpr, UPCAssignExpr distinguishes between types of assignment for performance reasons. Simple assignment (using the C '=' operator) ignores the value currently in the target location, so a simple call to the global array's "putter" is all that is necessary. For update assignment (using any of C's compound assignment operators), the previous value is downloaded using the "getter" and stored in a temporary variable, also created at the instantiation of the array's UPCArrayDecl node. The temporary variable is updated with the assigned value and replaced using the array's "putter"

auxiliary function. This could produce incorrect values in cases where update assignment expressions are used to subscript update assignment expressions that reference the same array. While this is believed to be rare and easy to work around, future work should eliminate this restriction.

The output code for this node utilizes the C comma operator (fig. 5.13) to ensure that the value of the expression is the updated value of the array location, as is the case in UPC when update assignment is used on global array locations. Because `CommaExpr` is a subclass of `Expression`, this is not restrictive, since a `CommaExpr` is legal wherever an assignment expression is.

```
f(x),3;
```

Figure 5.13 This statement has a value of 3 regardless of the value returned by `f(x)`.

#### 5.6.2.8 `UPCPtrAssignExpr`

The `UPCPtrAssignExpr` node handles assignment through a pointer to a UPC shared data type. Like `UPCAssignExpr`, it treats simple assignment differently than update assignment for performance reasons. A single temporary variable is used for all pointers in this case. This places some restrictions on the type of subscript that may be used. In particular, another UPC shared array update assignment expression may not be used to subscript a UPC shared array update assignment expression. While this case is believed to be quite rare, future work should involve removing this prohibition.

It is necessary to separate this node from `UPCAssignExpr` because the auxiliary functions for pointer access differ from those for array access. Pointer-specific information (such as the GA array handle) is needed to perform operations on the data referenced

by translated UPC pointers. Also, like UPCPtrExpr, only a single subscript is supported without restriction.

#### 5.6.2.9 UPCAllAllocExpr and UPCFreeExpr

The UPCAllAllocExpr and UPCFreeExpr correspond to user calls to the upc_all_alloc and upc_free functions, respectively. The UPCAllAllocExpr node outputs a reference to auxiliary code to allocate a new global array and the UPCFreeExpr node outputs a direct call to GA_Destroy.

#### 5.6.2.10 UPCRefExpr

The UPCRefExpr node corresponds to the use of the C indirection operator ('&') on a global array element. It outputs a call to the target array's "indirector" auxiliary function. Since this node naively outputs this function call, a UPC global array reference must be fully subscripted in order to take its address in this way. This restriction could be removed by detecting the number of subscripted dimensions (a trivial task given the existing implementation of UPCIndexExpr) and inserting zeros as arguments to the function call for dimensions not given. Future work should eliminate this issue.

#### 5.6.2.11 UPCCastExpr

The UPCCastExpr node is created in lieu of its superclass, the CastExpr, when a UPC shared pointer is casted to a local ("private") pointer. It outputs a reference to auxiliary code to perform this operation.

GA is an opaque model that does not allow client code to access GA elements directly. Rather, this data must be transferred into a user-space buffer to allow it to be modified through a local pointer. The auxiliary code for handling UPCCastExpr nodes determines the GA data distribution, then transfers the data to the target address using NGA_Get. Code following the typecast expression may freely modify local data through the local

pointer. The linearization expected by the UPC programmer is guaranteed through the use of block size information stored about the UPC pointer when it was declared.

Several issues arise from this method of data management. UPC guarantees that both pointers to "private" and pointers to shared may be used to manipulate the global data after a typecast like this. With two copies of the data now on the node performing the typecast, coherency issues abound. To resolve these issues, a write-back mechanism is employed by the translator to update global data in certain situations when it is modified through a "private" pointer.

The translator's second pass over the AST locates all `ReturnStemnts` and inserts before them a check to determine if any pointers to shared arrays were typecast during the executing scope. If so, the local buffers are written back to their corresponding global arrays. Write-backs are also performed by the code output by the `UPCSynchStemnt` node. Still, mixed references to the same array using shared pointers and local pointers produce undefined results. This could be corrected by checking all references to global arrays to ensure that the accessed portions are not buffered on any local host. Performance would, however, suffer as a result. A write-through policy could be implemented that updates the global data after one or several updates to local data could be implemented. Performance degradation would also result from this situation.

Considering the performance tradeoff, this restriction (prohibiting mixed references to the same portion of shared arrays) seems rather reasonable. Mixed access to to the same shared array elements with local pointers on one node and shared pointers on another results in undefined behavior anyway, unless a synchronization method is employed. Thus, identical behavior is achieved by the translator here. The only consideration is for unsynchronized access by a single node using both methods. In any UPC implementation, a program performs better when, in the event the user has typecasted a pointer to a shared type to a local pointer, only the local pointer is used on the casting node while it is in scope. Therefore, this implementation highlights the best-performance

situation in both the translated and untranslated cases.

Future work could move toward making the translation optional if it were determined that this restriction is too severe. A configurable data management policy (write-back versus write-through) could also be implemented to deal with this situation. Existing translator code would make this policy implementation relatively straightforward.

#### 5.6.2.12 `UPCSynchStemnt`

The `UPCSynchStemnt` node corresponds to a `upc_barrier` statement. This node outputs the equivalent GA barrier, `GA_Sync()`, along with an explicit call for write-back synchronization in cases where typecasting may have caused data transfer. Since this must complete before the end of the synchronization phase, a second `GA_Sync()` is then performed.

#### 5.6.2.13 `UPCForallStemnt`

The `UPCForallStemnt` node, as its name implies, corresponds to UPC's `upc_forall` statement. Its output code is a simple C `for` loop, with an extra AST grafted onto the beginning.

The format of the grafted AST depends on the type of expression given as the loop's fourth component. The UPC standard allows this statement to be `continue` (in which case no AST is grafted), an integer expression (corresponding to an `Expression` node whose `Type` field is a `BaseType` node indicating an integer), or a UPC shared address (corresponding to a `UPCVariable` or `UPCRefExpr`).

When an `Expression` node appears here, the translator inserts a conditional statement (an `IfStemnt` node), testing whether the value of the expression taken `mod` the number of nodes equals the identifier of the current node [5]. The false-branch of this `IfStemnt` node is a `ContinueStemnt` node, indicating that the current loop iteration is to be skipped. When a `UPCVariable` or `UPCRefExpr` appears as the affinity component,

an `IfStement` is created to test whether the current node holds the expressed address. Like above, the "false" branch skips the current loop iteration.

Observation of extant UPC code seems to indicate there are a few simple patterns often followed by UPC users when writing `upc_forall` loops. Future work could leverage these patterns to dramatically improve performance.

## 5.7 Extensibility

It is worth noting here that the methods used to develop this translator are extensible to permit additional APIs to be added with a minimum of effort. The methods used to program for extensibility have interesting differences, though, with respect to the type of model being translated. Language-based models such as UPC require more implementation at the grammar level, whereas library-based models employ more *a priori* auxiliary code.

A priori auxiliary code does not significantly complicate the overall translator implementation, because it is largely self-contained. Its purpose is mainly to iron out inconsistencies in how library-based approaches handle bookkeeping information about data in the global memory space. Thus, it is not of much use to other models except as an example. Grammar modifications for library-based implementations are generally all contained together, so they do not overly complicate the grammar file. A modular translator might export these grammar modifications to separate files to be conditionally included depending on the model (or models) to be utilized.

A common framework for AST node subclassing could easily be used to share code generation mechanisms. Because of the strong equivalences among the methods used by various APIs to implement the required elements of DSM models, such a framework could be developed and extended to encompass translations for many different models.

To avoid excessive grammar complications (and conflicts) involved in multiple language-

based extensions, run-time type information can be used to distinguish extension nodes, as opposed to the introduction of additional grammar rules, especially considering the complexity of the C grammar. This also serves to more clearly highlight the similarities among models that use this type of implementation, facilitating cooperative code development. It also assists in the formation of coherent AST class hierarchies.

## 5.8 Extensions

A translation environment encompassing multiple DSM APIs presents many interesting opportunities besides simply porting software. A modular translator with support for many different types of APIs provides intriguing opportunities for paradigm development within the DSM framework. One such opportunity is for simple extensions of existing APIs to be implemented conveniently where they are needed or useful. As a demonstration of this concept, this translator implements several extensions to the UPC model that drastically improve its usefulness over a wide range of use cases.

1. *Locally scoped shared arrays:* This translator permits the local scoping and declaration of UPC shared arrays in any legal C scope. The UPC standard requires that all UPC shared arrays be declared with global scope. This is counterintuitive to programmers, who generally prefer to scope variables as tightly as possible. Although the output code declares these arrays with global scope, that fact is hidden from the user by the translator, so programmers are free to scope shared arrays as they desire.

2. *Dynamically sized shared arrays:* While standard UPC permits (and, in some cases, requires) the user to size shared arrays based upon the number of participating nodes, it severely restricts this behavior. C programmers are accustomed to being required by compilers to declare all array dimensions in compile-time constant

terms. Yet, it is easy to work around this limitation when dealing with local memory: multi-level dynamic memory allocation is quite straightforward, with familiar semantics. For shared arrays, however, the desired semantics cannot be achieved through dynamic allocation. `upc_all_alloc` is able to allocate memory as required, but memory allocated through that call cannot be conveniently addressed through pointers using familiar C syntax. To make UPC practical and natural for use in component code (and to increase UPC's utility in application code), the translator permits any expression with a defined value at program startup to be used to dimension shared arrays. These expressions are added to the "exec" AST and evaluated after the declaration block in `main`.

3. *Dynamically blocked shared arrays:* Much as programmers cannot necessarily know the proper dimensions for all shared arrays at compile time, the programmer cannot be expected to know how the data should be laid out. Heuristic computations for dynamic load balancing could be used to determine an optimal layout. System calls to query for cache and memory sizes may assist in fine-tuning layout parameters. Large scientific computations utilizing UPC code for utility computation could call the same code for different array sizes without recompilation. Dynamic blocking also follows from dynamic dimensioning, as one could hardly expect a user that provided a dynamically sized array to specify a static block size. The translator implements an extension accepting any expression, subject to the same conditions required of the dimensioning expressions mentioned above, to be used as the layout qualifier of a UPC shared array.

4. *Dynamically aware pointers to shared arrays:* Following from the above two extensions, it seems reasonable that pointers to shared arrays automatically access the arrays to which they refer in an intuitive manner, rather than the crude and inflexible manner required by standard UPC. Standard UPC allows multiple point-

ers to the same array to index that array differently, traversing the elements in a different order. Furthermore, since all block size parameters must be known at compile-time, this concept does not fit with any extension offering dynamic sizing and blocking. Because of the way they are typed (§3.2.1), UPC pointers also cannot pass even this static blocking information across function calls. Therefore, the translator's implementation of the `UPCPtrMathExpr` node (§5.6.2.6) causes all pointers to follow the logical bounds of the arrays to which they point. This results in a greatly streamlined and intuitive interface for performing array-wide operations.

5. *Strong typing for pointers to shared arrays:* Because UPC handles `shared` as a storage class, it does not carry with it strong type information that persists across function calls or even assignment. This severely weakens type checking of UPC code. Although explicit typecasting is allowed in many cases, the purpose of typecasting is to permit the programmer to circumvent the compiler's standard notions of type semantics when they are unknown to it or inconvenient for the programmer. As such, typecasting operations are not altered except where necessary. This translator extends UPC's notion of type to differentiate among pointers to local memory, pointers to shared memory, and shared arrays. This provides the possibility for far more robust type checking and handling than an implementation of the standard.

# CHAPTER 6 Conclusions and future work

Obviously, as distributed-shared memory (DSM) APIs continue to develop and mature, parts of the translator system will need to adapt to accommodate them. The equivalence of a broad range of DSM programming models provides further opportunities for development, both theoretical and practical.

## 6.1 Near-future adaptations for GA

The initial work with the Global Arrays library (GA) translation was, of course, focused on producing equivalent programs. Future work should focus on performance optimization, especially of GA collective operations. Another way to improve performance would be to devise an alternate method of linearizing and localizing GA array elements within UPC shared address space.

GA's array declaration and initialization syntax does not mesh well with models that support static global address space data structure allocation. A mechanism could be provided that allows the programmer to provide hints to the translator when an array is allocated and its dimensions are known (or are based on a set of parameters that can be known) at compile time. For example, a UPC output code in this circumstance could statically allocate space with equivalent dimensions and avoid all linearization and localization overhead. This would also enhance the legibility of the output, since much less bookkeeping information would be necessary for such arrays, allowing for more direct integration with the UPC model. It seems likely that this would be the case for

other language extensions supporting static allocations as well.

Additional collective operations could be implemented. This implementation focused on the necessary primitives to create working DSM programs, along with several common "convenience routines" commonly used by GA programmers. More of these convenience routines could be implemented. A complete set of GA convenience routines, translated to another API, would essentially compose a de facto standard library for DSM programming in that API. Libraries such as these, if they were to become standards, could be used by vendors as a vehicle to provide further optimized versions of their implementations of specific APIs on specific DSM hardware. In turn, these more aggressively optimized APIs can serve as translation targets for any input language accepted by the translator.

More direct calls to the underlying transport layer could be implemented. This could either be done directly, or by virtue of the addition of other transport models to the translation environment. This could also explore the realm of "mixed-API" DSM programming, that could utilize more than one API simultaneously in a single program (or programs composed of routines that utilize different APIs) as needs arise.

## 6.2 Near-future adaptations for UPC

A number of adaptations could be made to the Unified Parallel C (UPC) translator code to relax restrictions on the possible inputs. The "exec" synthetic AST could be written to another file to be referenced from `main` later. This would eliminate the restriction requiring `main`'s translation unit to be processed in every input program. Or, like many UPC implementations, the `main` function could be renamed and executed manually by an auxiliary `main` function that sets up a valid parallel state in advance and cleans up afterward.

Tighter integration between UPC pointer-to-shared-array types and pointer-to-array

types could be implemented to relax syntactic restrictions imposed by the translator. This would allow for more convenient C notation for shared addresses in certain situations.

Pattern matching for `upc_forall` loops could be explored. There appear to be several very common forms of `upc_forall`, the classic example being `upc_forall(i=0;i=n;++i;&a[i])`. This is the classic traversal of a shared array, divided by the locality of the data. Most DSM APIs have mechanisms for such traversals. Also, many of these types of loops feature simple, one-argument bodies. These could be explored as pattern-matching candidates as well.

A library for optimized collective operations could be implemented in UPC. With the extensions already provided by this translator, UPC is a suitable implementation language for distributed libraries. Such libraries could be developed and used alongside more traditional high-performance scientific codes. Performance and implementation studies of these types of situations would be informative in determining the direction of DSM API development and research.

## 6.3 Implications

This work provides a mapping between GA programs and UPC programs, demonstrating that any program in one API can be expressed in the other by means of this translator. This proves by construction that these two models are equivalent within the constraints outlined by this thesis. GA and UPC were chosen as the first implementation test because of their divergent characteristics. Therefore, this translator serves as strong evidence that any other DSM API may be part of this equivalence class as well.

As more and more models are added to this framework, the flexibility of the entire system will increase. Extensions to existing APIs will become more apparent and easy to implement. Code generation code will be refactored and amalgamated to reduce

development time. More rigid standards will evolve for grammar extensions to simplify the addition process. More optimizations will be discovered to streamline the output code, both in terms of performance and legibility.

## 6.4 Partial equivalences

There may be additional models that may not strictly fit the DSM theoretical model, but do satisfy a subset of the required assumptions. Perhaps these models could be integrated into a translation framework as input-only, that is, without integrated code generation mechanisms. This could enable translation to a number of other models for performance or portability reasons. It would also serve to create an interesting hierarchy of theoretical machine models and their relationships to each other. These notions could have impact on performance analysis research. For example, if a parallel machine model X can simulate a parallel machine Y with no change in time complexity, then any algorithm that can be run on machine Y can be run on machine X in the same amount of time. Therefore, if an algorithm appears difficult to develop for Y's architecture, yet easy to develop for X's architecture, a simulation proof would permit algorithm designers more flexibility.

Along the same lines, some studied theoretical models could make supersets of the assumptions made by the DSM model. As a result, they could be studied for implementation at the code generation phase, lacking an input phase. This could have the same implications as discussed above.

# REFERENCES

[1] Aho, Sethi, and Ulman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1985.

[2] Brightwell, Brown, Stout, and Wen. Experiences implementing sorting algorithms in unified parallel C. Available from website, as yet unpublished., 2005.

[3] David E. Culler, Andrea C. Arpaci-Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine A. Yelick. Parallel programming in Split-C. In *Supercomputing*, pages 262–273, 1993.

[4] Donnelly and Stallman. *The Bison Manual.* Free Software Foundation, 2002.

[5] El-Ghazawi, Carlson, and Draper. *UPC Language Specifications V1.1.1*, 2003.

[6] Yelick et. al. Titanium: A high-performance java dialect. In *ACM Workshop on Java for High-Performance Network Computing*, 1998.

[7] Shaun Flisakowski. The ctool library. http://ctool.sourceforge.net/, as accessed throughout 2004.

[8] Fortune and Wyllie. Parallelism in random access machines. In *10th Symposium on the theory of Computing*, pages 114–118, 1978.

[9] Harbison and Steele. *C: A Reference Manual.* Prentice Hall, 2002.

[10] ISO/IEC. *International Standard 9899 (Programming Languages – C)*, 1999.

[11] Message-Passing Interface Forum. *MPI: A Message Passing Interface Standard*, 1995.

[12] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[13] Numrich and Reid. Co-array fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998.

[14] Paxson. *Flex, version 2.5*, 1990.

[15] Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1997.

[16] Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

# ACKNOWLEDGEMENTS

Thanks to the Ames Laboratory of the United States Department of Energy for providing funding for this research.

In particular, I would like to thank the Scalable Computing Laboratory, where I "grew up" from a freshman undergraduate to a successful graduate student and researcher.

Every graduate student thanks their committee, and I am no exception. Thanks specifically to Dr. Leavens for assistance in revising and completing this thesis, and to Dr. Gordon for motiviation to show that Cubs fans too, can succeed.

Last and most of all, for sincere mentorship and friendship, heartfelt thanks to Ricky Kendall without whom I would not be where I am today.

# APPENDIX Grammar File for the translator

For reference and completeness, the complete grammar for the implemented translator is given here. It is presented in a format similar to a bison [4] input file, with annotations as mentioned in figure 5.5.

```
program : (empty)
        | trans_unit
        | error;

trans_unit : top_level_decl top_level_exit
           | trans_unit top_level_decl top_level_exit;

top_level_exit : (empty);

top_level_decl : decl_stemnt
               | func_def
               | PP_LINE
               | error SEMICOLON
               | error RBRACE top_level_exit;

func_def : func_spec cmpnd_stemnt;

func_spec : decl_specs func_declarator opt_KnR_declaration_list
          | no_decl_specs declarator opt_KnR_declaration_list;
```

```
cmpnd_stemnt : LBRACE opt_declaration_list opt_stemnt_list RBRACE
             | error RBRACE;

opt_stemnt_list : (empty)
                | stemnt_list;

stemnt_list : stemnt
            | stemnt_list stemnt
            | stemnt_list PP_LINE;

stemnt : stemnt_reentrance;

cmpnd_stemnt_reentrance : LBRACE opt_declaration_list
                          opt_stemnt_list_reentrance RBRACE
                        | error RBRACE;

opt_stemnt_list_reentrance : (empty)
                           | stemnt_list_reentrance;

stemnt_list_reentrance : stemnt_reentrance
                       | stemnt_list_reentrance stemnt_reentrance
                       | stemnt_list_reentrance PP_LINE;
```

```
stemnt_reentrance : expr_stemnt
                  | labeled_stemnt
                  | cmpnd_stemnt_reentrance
                  | cond_stemnt
                  | iter_stemnt
                  | switch_stemnt
                  | break_stemnt
                  | continue_stemnt
                  | return_stemnt
                  | goto_stemnt
                  | synch_stemnt
                  | null_stemnt
                  | error SEMICOLON;

expr_stemnt : expr SEMICOLON;

labeled_stemnt : label COLON stemnt_reentrance;

cond_stemnt : if_stemnt
            | if_else_stemnt;
```

```
iter_stemnt : do_stemnt
            | while_stemnt
            | for_stemnt
            | forall_stemnt;

switch_stemnt : SWITCH LPAREN expr RPAREN stemnt_reentrance;

break_stemnt : BREAK SEMICOLON;

continue_stemnt : CONT SEMICOLON;

return_stemnt : RETURN opt_expr SEMICOLON;

goto_stemnt : GOTO LABEL_NAME SEMICOLON;

synch_stemnt : U_NOTIFY opt_expr SEMICOLON
             | U_WAIT opt_expr SEMICOLON
             | U_BARRIER opt_expr SEMICOLON
             | U_FENCE SEMICOLON;
```

```
null_stemnt : SEMICOLON;

if_stemnt : IF LPAREN expr RPAREN stemnt_reentrance;

if_else_stemnt : IF LPAREN expr RPAREN stemnt_reentrance
                 ELSE stemnt_reentrance;

do_stemnt : DO stemnt_reentrance WHILE LPAREN expr RPAREN SEMICOLON;

while_stemnt : WHILE LPAREN expr RPAREN stemnt_reentrance;

for_stemnt : FOR LPAREN opt_expr SEMICOLON opt_expr SEMICOLON
             opt_expr RPAREN stemnt_reentrance ;

forall_stemnt : U_FORALL LPAREN opt_expr SEMICOLON opt_expr SEMICOLON
                opt_expr SEMICOLON affinity RPAREN stemnt_reentrance;

affinity : opt_expr
         | CONT;
```

```
label : named_label
      | case_label
      | deflt_label;

named_label : ident;

case_label : CASE const_expr
           | CASE const_expr ELLIPSIS const_expr;

deflt_label : DEFLT;

cond_expr : log_or_expr
          | log_or_expr QUESTMARK expr COLON cond_expr;

assign_expr : cond_expr
            | unary_expr assign_op assign_expr [ UPCArrayExpr, UPCPtrExpr ];

opt_const_expr : (empty)
               | const_expr;
```

```
const_expr : expr;

opt_expr : (empty)
         | expr;

expr : comma_expr;

log_or_expr : log_and_expr
            | log_or_expr OR log_and_expr;

log_and_expr : bitwise_or_expr
             | log_and_expr AND bitwise_or_expr;

log_neg_expr : NOT cast_expr;

bitwise_or_expr : bitwise_xor_expr
                | bitwise_or_expr B_OR bitwise_xor_expr;

bitwise_xor_expr : bitwise_and_expr
                 | bitwise_xor_expr B_XOR bitwise_and_expr;
```

```
bitwise_and_expr : equality_expr
                 | bitwise_and_expr B_AND equality_expr;

bitwise_neg_expr : B_NOT cast_expr;

cast_expr : unary_expr
          | LPAREN type_name RPAREN cast_expr [ UPCVariable ];

equality_expr : relational_expr
              | equality_expr equality_op relational_expr;

relational_expr : shift_expr
                | relational_expr relation_op shift_expr;

shift_expr : additive_expr
           | shift_expr shift_op additive_expr;

additive_expr : mult_expr
              | additive_expr add_op mult_expr;
```

```
mult_expr : cast_expr
          | mult_expr mult_op cast_expr;

unary_expr : postfix_expr
           | sizeof_expr
           | unary_minus_expr
           | unary_plus_expr
           | log_neg_expr
           | bitwise_neg_expr
           | addr_expr
           | indirection_expr
           | preinc_expr
           | predec_expr;

sizeof_expr : SIZEOF LPAREN type_name RPAREN
            | SIZEOF unary_expr;

unary_minus_expr : MINUS cast_expr;

unary_plus_expr : PLUS cast_expr;
```

```
addr_expr : B_AND cast_expr [ UPCArrayExpr, UPCVariable ];

indirection_expr : STAR cast_expr;

preinc_expr : INCR unary_expr [ UPCArrayExpr, UPCVariable ];

predec_expr : DECR unary_expr [ UPCArrayExpr, UPCVariable ];

comma_expr : assign_expr
           | comma_expr COMMA assign_expr;

prim_expr : ident
          | paren_expr
          | constant;

upc_prim_expr : shared_ident;

paren_expr : LPAREN expr RPAREN
           | LPAREN error RPAREN;
```

```
postfix_expr : prim_expr
             | upc_prim_expr
             | subscript_expr
             | upc_subscript_expr
             | comp_select_expr
             | func_call
             | ga_call
             | postinc_expr
             | postdec_expr;

subscript_expr : postfix_expr LBRCKT expr RBRCKT;

upc_subscript_expr : upc_prim_expr LBRCKT expr RBRCKT
                   | upc_subscript_expr LBRCKT expr RBRCKT;

comp_select_expr : direct_comp_select
                 | indirect_comp_select;

postinc_expr : postfix_expr INCR [ UPCIndexExpr, UPCVariable ];

postdec_expr : postfix_expr DECR [ UPCIndexExpr, UPCVariable ];
```

```
field_ident : any_ident;

direct_comp_select : postfix_expr DOT field_ident;

indirect_comp_select : postfix_expr ARROW field_ident;

ga_call : G_CREATE LPAREN assign_expr COMMA assign_expr COMMA
          assign_expr COMMA assign_expr COMMA assign_expr RPAREN
        | G_CREATE_C LPAREN assign_expr COMMA assign_expr
          COMMA assign_expr COMMA assign_expr COMMA assign_expr COMMA assign_expr RI
        | G_DUP LPAREN assign_expr COMMA assign_expr RPAREN
        | G_NID LPAREN RPAREN
        | G_NNO LPAREN RPAREN
        | G_DST LPAREN assign_expr COMMA assign_expr COMMA
          assign_expr COMMA assign_expr RPAREN
        | G_SYNC LPAREN RPAREN
        | G_DEST LPAREN assign_expr RPAREN
        | G_GET LPAREN assign_expr COMMA assign_expr COMMA
          assign_expr COMMA assign_expr COMMA assign_expr RPAREN
        | G_PUT LPAREN assign_expr COMMA assign_expr COMMA
          assign_expr COMMA assign_expr COMMA assign_expr RPAREN
```

```
| G_INIT LPAREN RPAREN
| G_INQ LPAREN assign_expr COMMA assign_expr COMMA
  assign_expr COMMA assign_expr RPAREN
| G_TERM LPAREN RPAREN
| G_ZERO LPAREN assign_expr RPAREN
| G_ACC LPAREN assign_expr COMMA assign_expr COMMA
  assign_expr COMMA assign_expr COMMA assign_expr COMMA
  assign_expr RPAREN
| MPI_INIT LPAREN assign_expr COMMA assign_expr RPAREN
| MPI_FNAL LPAREN RPAREN
| MPI_ABRT LPAREN assign_expr COMMA assign_expr RPAREN
| G_BCAST LPAREN assign_expr COMMA assign_expr COMMA
  assign_expr RPAREN
| G_ERR LPAREN assign_expr COMMA assign_expr RPAREN
| G_ADD LPAREN assign_expr COMMA assign_expr COMMA
  assign_expr COMMA assign_expr COMMA assign_expr RPAREN
| G_PRNT LPAREN assign_expr RPAREN
| G_PRNTD LPAREN assign_expr RPAREN
| G_PRNTS LPAREN RPAREN
| G_DDOT LPAREN assign_expr COMMA assign_expr RPAREN
| U_A_ALLOC LPAREN assign_expr COMMA assign_expr RPAREN
| U_FREE LPAREN assign_expr RPAREN;
```

```
func_call : postfix_expr LPAREN opt_expr_list RPAREN;

opt_expr_list : (empty)
              | expr_list;

expr_list : assign_expr
          | expr_list COMMA assign_expr;

add_op : PLUS
       | MINUS;

mult_op : STAR
        | DIV
        | MOD;

equality_op : COMP_EQ;

relation_op : COMP_ARITH;
```

```
shift_op : L_SHIFT
         | R_SHIFT;

assign_op : EQ
          | ASSIGN;

constant : INUM
         | RNUM
         | CHAR_CONST
         | LCHAR_CONST
         | STRING
         | LSTRING;

opt_KnR_declaration_list : (empty)
                         | declaration_list ;

opt_declaration_list : (empty)
                     | declaration_list ;

declaration_list : declaration SEMICOLON
                 | declaration SEMICOLON declaration_list;
```

```
decl_stemnt : old_style_declaration SEMICOLON
            | declaration SEMICOLON;

old_style_declaration : no_decl_specs opt_init_decl_list;

declaration : decl_specs opt_init_decl_list;

no_decl_specs : (empty);

decl_specs : decl_specs_reentrance_bis ;

abs_decl : abs_decl_reentrance;

type_name : type_name_bis;

type_name_bis : decl_specs_reentrance_bis
              | decl_specs_reentrance_bis abs_decl;

decl_specs_reentrance_bis : decl_specs_reentrance;
```

```
local_or_global_storage_class : EXTRN
                              | STATIC
                              | TYPEDEF;

local_storage_class : AUTO
                    | REGISTR;

storage_class : local_or_global_storage_class
              | local_storage_class;

type_spec : type_spec_reentrance;

opt_decl_specs_reentrance : (empty)
                          | decl_specs_reentrance;

decl_specs_reentrance : storage_class opt_decl_specs_reentrance
                        opt_decl_specs_reentrance
                      | type_qual opt_decl_specs_reentrance
                      | shared_decl opt_decl_specs_reentrance;
```

```
opt_comp_decl_specs : (empty)
                    | comp_decl_specs_reentrance;

comp_decl_specs_reentrance : type_spec_reentrance opt_comp_decl_specs
                           | type_qual opt_comp_decl_specs;

comp_decl_specs : comp_decl_specs_reentrance;

decl : declarator opt_gcc_attrib;

init_decl : decl
          | decl EQ initializer;

opt_init_decl_list : (empty)
                   | init_decl_list;

init_decl_list : init_decl_list_reentrance;

init_decl_list_reentrance : init_decl
                          | init_decl_list_reentrance COMMA init_decl;
```

```
initializer : initializer_reentrance;

initializer_list : initializer_reentrance
                 | initializer_list COMMA initializer_reentrance;

initializer_reentrance : assign_expr
                       | LBRACE initializer_list opt_comma RBRACE;

opt_comma : (empty)
          | COMMA;

type_qual : type_qual_token;

type_qual_token : CONST
                | VOLATILE
                | RESTRICT;

shared_token : SHARED
             | STRICT SHARED
             | RELAXED SHARED;
```

```
shared_decl : shared_token
            | shared_token LBRCKT STAR RBRCKT
            | shared_token LBRCKT opt_const_expr RBRCKT;

type_qual_list : type_qual_token
               | type_qual_list type_qual_token;

opt_type_qual_list : (empty)
                   | type_qual_list;

type_spec_reentrance : enum_type_define
                     | struct_type_define
                     | union_type_define
                     | enum_tag_ref
                     | struct_tag_ref
                     | union_tag_ref
                     | typedef_name
                     | VOID
                     | CHAR
                     | SHORT
                     | INT
```

```
                | LONG
                | FLOAT
                | DOUBLE
                | SGNED
                | UNSGNED;

    typedef_name : TYPEDEF_NAME;

        tag_ref : TAG_NAME;

   struct_tag_ref : STRUCT tag_ref;

    union_tag_ref : UNION tag_ref;

     enum_tag_ref : ENUM tag_ref;

   struct_tag_def : STRUCT tag_ref;

struct_type_define : STRUCT LBRACE struct_or_union_definition RBRACE
                   | struct_tag_def LBRACE struct_or_union_definition RBRACE;
```

```
union_tag_def : UNION tag_ref;

union_type_define : UNION LBRACE struct_or_union_definition RBRACE
                  | union_tag_def LBRACE struct_or_union_definition RBRACE;

enum_tag_def : ENUM tag_ref;

enum_type_define : ENUM LBRACE enum_definition RBRACE
                 | enum_tag_def LBRACE enum_definition RBRACE;

struct_or_union_definition : (empty)
                           | field_list;

enum_definition : (empty)
                | enum_def_list opt_trailing_comma;

opt_trailing_comma : (empty)
                   | COMMA;
```

```
enum_def_list : enum_def_list_reentrance;

enum_def_list_reentrance : enum_const_def
                         | enum_def_list COMMA enum_const_def;

enum_const_def : enum_constant
               | enum_constant EQ assign_expr;

enum_constant : any_ident;

field_list : field_list_reentrance;

field_list_reentrance : comp_decl SEMICOLON
                      | field_list_reentrance SEMICOLON
                      | field_list_reentrance comp_decl SEMICOLON;

comp_decl : comp_decl_specs comp_decl_list
          | comp_decl_specs;
```

```
comp_decl_list : comp_decl_list_reentrance;

comp_decl_list_reentrance : comp_declarator opt_gcc_attrib
                          | comp_decl_list_reentrance COMMA
                            comp_declarator opt_gcc_attrib;

comp_declarator : simple_comp
                | bit_field;

simple_comp : declarator;

bit_field : opt_declarator COLON width;

width : cond_expr;

opt_declarator : (empty)
               | declarator;

declarator : declarator_reentrance_bis;
```

```
func_declarator : declarator_reentrance_bis;

declarator_reentrance_bis : pointer direct_declarator_reentrance_bis
                          | direct_declarator_reentrance_bis;

direct_declarator_reentrance_bis : direct_declarator_reentrance;

direct_declarator_reentrance : maybe_shared_ident
                             | LPAREN declarator_reentrance_bis RPAREN
                             | array_decl
                             | direct_declarator_reentrance LPAREN
                               param_type_list RPAREN
                             | direct_declarator_reentrance LPAREN
                               ident_list RPAREN
                             | direct_declarator_reentrance LPAREN RPAREN;

array_decl : direct_declarator_reentrance LBRCKT opt_const_expr RBRCKT;

pointer_start : STAR opt_type_qual_list;
```

```
pointer_reentrance : pointer_start
                   | pointer_reentrance pointer_start;

pointer : pointer_reentrance;

ident_list : (empty) ident_list_reentrance ;

ident_list_reentrance : maybe_shared_ident
                      | ident_list_reentrance COMMA ident;

ident : IDENT;

shared_ident : SHARED_IDENT;

maybe_shared_ident : ident
                   | shared_ident;

typename_as_ident : TYPEDEF_NAME;
```

```
any_ident : ident
          | typename_as_ident;

opt_param_type_list : (empty)
                    | param_type_list_bis ;

param_type_list : param_type_list_bis ;

param_type_list_bis : param_list
                    | param_list COMMA ELLIPSIS;

param_list : param_decl
           | param_list COMMA param_decl;

param_decl : param_decl_bis;

param_decl_bis : decl_specs_reentrance_bis declarator
               | decl_specs_reentrance_bis abs_decl_reentrance
               | decl_specs_reentrance_bis;
```

```
abs_decl_reentrance : pointer
                    | direct_abs_decl_reentrance_bis
                    | pointer direct_abs_decl_reentrance_bis;

direct_abs_decl_reentrance_bis : direct_abs_decl_reentrance;

direct_abs_decl_reentrance : LPAREN abs_decl_reentrance RPAREN
                           | LBRCKT opt_const_expr RBRCKT
                           | direct_abs_decl_reentrance LBRCKT
                             opt_const_expr RBRCKT
                           | LPAREN opt_param_type_list RPAREN
                           | direct_abs_decl_reentrance LPAREN
                             opt_param_type_list RPAREN;

opt_gcc_attrib : (empty)
               | gcc_attrib;

gcc_attrib : ATTRIBUTE LPAREN LPAREN gcc_inner RPAREN RPAREN;

gcc_inner : (empty)
```

```
    | PACKED
    | CDECL
    | CONST
    | NORETURN
    | ALIGNED LPAREN INUM RPAREN
    | MODE LPAREN ident RPAREN
    | FORMAT LPAREN ident COMMA INUM COMMA INUM RPAREN;
```