CACR Technical Report

CACR-182 **February 2000**

Macroservers: An Execution Model for DRAM Processor-In-Memory Arrays

Hans P. Zima and Thomas L. Sterling

Mailing Address: CACR Technical Publications, California Institute of Technology,
Mail Code 158-79, Pasadena, CA 91125. Phone: (626) 395-6953 Fax: (626) 584-5917

Macroservers*

An Execution Model for DRAM Processor-In-Memory Arrays

Hans P. Zima[a,b] and Thomas L. Sterling[b]

[a] Institute for Software Science, University of Vienna, Austria

[b] Center for Advanced Computing Research (CACR), California Institute of Technology, Pasadena, CA 91125, U.S.A.

Abstract

The emergence of semiconductor fabrication technology allowing a tight coupling between high-density DRAM and CMOS logic on the same chip has led to the important new class of Processor-In-Memory (PIM) architectures. Newer developments provide powerful parallel processing capabilities on the chip, exploiting the facility to load wide words in single memory accesses and supporting complex address manipulations in the memory. Furthermore, large arrays of PIMs can be arranged into a massively parallel architecture. In this report, we describe an object-based programming model based on the notion of a *macroserver*. Macroservers encapsulate a set of variables and methods; threads, spawned by the activation of methods, operate asynchronously on the variables' state space. Data distributions provide a mechanism for mapping large data structures across the memory region of a macroserver, while work distributions allow explicit control of bindings between threads and data. Both data and work distributions are first-class objects of the model, supporting the dynamic management of data and threads in memory. This offers the flexibility required for fully exploiting the processing power and memory bandwidth of a PIM array, in particular for irregular and adaptive applications. Thread synchronization is based on atomic methods, condition variables, and futures. A special type of lightweight macroserver allows the formulation of flexible scheduling strategies for the access to resources, using a monitor-like mechanism.

*The work described in this paper was partially supported by the Priority Research Project F011 "AURORA" funded by the Austrian Science Fund and by the HTMT Project funded by NASA/JPL Grant Number 49-220-85602-0-3950 and NASA/Goddard Grant NAG5-4203.

Contents

1 Introduction

"Processor in Memory" or PIM technology and architecture has emerged as one of the most important domains of parallel computer architecture research and development. It is being pursued as a means of accelerating conventional systems for array processing [50] and for manipulating irregular data structures [25]. It is being considered as a basis for scalable spaceborne computing [54], as smart memory to manage systems resources in a hybrid technology multithreaded architecture for ultra-scale computing [55], and most recently as the means for achieving Petaflops performance [33]. PIM exploits recent advances in semiconductor fabrication processes that enable the integration of DRAM cell blocks and CMOS logic on the same chip. The benefit of PIM structures is that processing logic can have direct access to the memory block row buffers at an internal memory bandwidth on the order of 100 Gbps yielding the potential performance of 10 Gips (32-bit operands) on a memory chip with a 16 Mbyte capacity. Because of the efficiencies derived from staying on-chip, power consumption can be an order of magnitude lower than comparable performance with conventional microprocessor based systems. But the dramatic advances in performance will be derived from arrays of tightly coupled PIM chips in the hundreds or thousands, either alone, or in conjunction with external microprocessors. Such systems could deliver low Teraflops scale peak performance within the next couple of years at a cost of only a few million dollars (or less than $1M if mass produced) and possibly a Petaflops, at least for some applications, in five years.

The challenge to realizing the extraordinary potential of arrays of PIM is not simply the interesting problem of the basic on-chip structure and processor architecture but also the methodology for coordinating the synthesis of as much as a million PIM processors to engage in concert in the solution of a single parallel application. A large PIM array is not simply another MPP, it is a new balance of processing and memory in a new organization. Its local operation and global emergent behavior will be a direct reflection of a shared highly parallel system-wide model of computation that governs the execution and interactions of the PIM processors and chips. Such a computing paradigm must treat the semantic requirements of the whole system even as it derives its processing capabilities from the local mechanisms of the individual parts. A synergy of cooperating elements is to be accomplished through this shared execution model.

PIM differs significantly from more common MPP structures in several key ways. The ratio of computation performance to associated memory capacity is much higher. Access bandwidth (to on-chip memory) is a hundred times greater. And latency is lower by a factor of two to four while logic clock speeds are approximately half that of the highest speed microprocessors. Like clusters, PIM favors data oriented computing where operations are scheduled and performed at the site of the data, and tasks are often moved from one PIM to another depending on where the argument data is rather than moving the data. PIM processor utilization is less important than memory bandwidth. A natural organization of computation on a PIM array is a binding of tasks and data segments logically to coincide with physical data allocation while making remote service requests where data is non-local. This is very similar to evolving practices for accomplishing tasks on the Web including the use of Java and encourages an object-oriented approach to managing the logical tasks and physical resources of the PIM array.

This report presents a strategy for relating the physical resources of next generation PIM arrays to the logical requirements of user defined applications. The strategy is embodied in an intermediate form of an execution model that provides the generalized abstractions of both local and global computation in a unified framework. The principal abstract entity of the proposed model is the *macroserver*, a distributed agent of state and action. It complements the concept of the *microserver*, a purely local agent [10]. This early work explores one possible model that is object-based in a manner highly suitable to PIM structures but of a sufficiently high level with task virtualization that aggregations of PIM nodes can be cooperatively applied to a segment of parallel computation without phase changes in representations (as would be found with OpenMP combined with MPI).

The next section describes PIM architectures including the likely direction of their evolution over the next one to three years. Then, in Section 3, a brief overview of the concepts and the terminology used in the rest of the paper is presented. In Section 4, we outline the main components of macroserver class

declarations. Section 5 then discusses some specific features of macroservers, in particular the distribution and alignment of data structures. Section 6 introduces threads and explains the major mechanisms for their creation, management, and termination. The subsequent section deals with thread synchronization, covering mutual exclusion and synchronization via condition and future variables. The remaining sections provide an overview of related research (Section 8) and discuss the motivation for a number of design decisions, possible alternatives, and directions for future research (Section 9). The final Section 10 provides concluding remarks. The Appendix illustrates solutions for a number of example problems (Section A) and defines an interface to an underlying abstract machine model (Section B).

A final remark is in order here. This report does *not* attempt to provide a programming language definition – its emphasis is on the *semantics* of the model. However, since it was obviously necessary to adopt some programming language notation, in particular for examples, we decided to use Fortran 95 syntax, with ad-hoc extensions mainly motivated by HPF [31] and Opus [15].

2 Processor in Memory

For more than a decade, research experiments have been conducted with semiconductor devices that merged both logic and static RAM cell blocks on the same chips. Even earlier, simple processors and small blocks of SRAM could be found on simple control processors for embedded applications and of course modern microprocessors include high speed SRAM on chip for level 1 caches. But it was not until recently that industrial semiconductor fabrication processes made possible tightly coupled combinations of logic with DRAM cell blocks bringing relatively large memory capacities to PIM design. A host of research projects has been undertaken to explore the design and application space of PIM (many under DARPA sponsorship) culminating in the recent IBM announcement to build a Petaflops scale PIM array for the application of protein folding.

The opportunity of PIM is primarily one of bandwidth. Typical memory parts access a row of memory from a memory block and then select a subsegment of the row of bits to be sent to a requesting processor through the external interface. While newer generations of memory chips are improving effective bandwidth, PIMs make possible immediate access to all the bits of a memory row acquired through the sense amps. Processing logic, placed at the row buffer, can operate on all the data read (typically 64 32-bit words) in a single memory access under favorable conditions. While a number of PIM proposals plan to use previously developed processor cores to be "dropped into" the die, PIM offers important opportunities for new processor architecture design that simplifies operation, lowers development cost and time, and greatly improves efficiency and performance over classical processor architecture. Many of the mechanisms incorporated in today's processors are largely unnecessary in a PIM processor. At the same time, effective manipulation of the very wide words available on the PIM imply the need for augmented instruction sets.

PIM chips include several major subsystems, some of them replicated as space is available:

- memory blocks
- processor control
- wide ALU and data path/register set
- shared functional units
- external interfaces

Typically PIMs are organized into sets of memory block/processor pairs while sharing some larger functional units and the external interfaces among them [38]. Detailed design studies suggest that PIM processors comprise less than 20% of the available chip real estate while the memory capacity has access to more than half of the total space. Approximately a third of the die area is used for external I/O interface and control as well as shared functional units. This is an excellent ratio and emphasizes the value of optimizing for bandwidth utilization rather than processor throughput. An important advantage of the PIM approach is the ability to operate on all bits in a given row simultaneously. A new generation of very wide ALU and

corresponding instruction sets exploit this high memory bandwidth to accomplish the equivalent of many conventional operations (e.g. 32-bit integer) in a single cycle. An example of such a wide ALU is the ASAP ISA developed at the University of Notre Dame and used in such experimental PIM designs as Shamrock and MIND. Other fundamental advances over previous generation PIMs are also in development to provide unprecedented capability and applicability. Among the most important of these are on-PIM virtual to physical address translation, message driven computation, and multithreading.

Virtual-to-Physical Address Translation Early PIM designs have been very simple assuming a physically addressed memory and often a SIMD control structure [21]. But such basic designs are limited in their applicability to a narrow range of problems. One requirement not satisfied by such designs is the ability to manipulate irregular data structures. This requires the handling of user virtual addresses embedded in the structure metadata. PIM virtual to physical address translation is key to extending PIM into this more generalized domain. Translation Lookaside Buffers can be of some assistance but they are limited in scalability and may not be the best solution. Virtual address translation is also important for protection in the context of multitasking systems. Address translation mechanisms are being provided for both the USC DIVA chip and the HTMT MIND chip.

Message-Driven Computation A second important advance for PIM architecture is message driven computation. Like simple memories, PIMs acquire external requests to access and manipulate the contents of memory cells. Unlike simple memories, PIMs may have to perform complex sequences of operations on the contents of memory defined by user application or supervisor service routines. Mechanisms are necessary that provide efficient response to complex requests while maintaining generality. Message driven computation assumes a sophisticated protocol and on-chip fast interpretation mechanisms that quickly identify both the operation sequence to be performed and the data rows upon which to be operated. A general message driven low-level infrastructure goes beyond interactions between system processors and the incorporated PIMs, it permits direct PIM to PIM interactions and control without system processor intervention. This reduces the impact of the system processors as a bottleneck and allows the PIMs to exploit data level parallelism at the fine grain level intrinsic to pointer linked sparse and irregular data structures. Both the USC DIVA chip and the HTMT MIND chip will incorporate "parcel" message driven computation while the IBM Blue Gene chip will permit direct PIM to PIM communications as well.

Multithreading A third important advance is the incorporation of multithreading into the PIM processor architecture. Although counter intuitive, multithreading actually greatly simplifies processor design rather than further complicating it because it provides a uniform hardware methodology for dynamically managing physical processor resources and virtual application tasks. Multithreading is also important because it permits rapid response to incoming service requests with low overhead context switching and also enables overlapping of computation, communication, and memory access activities, thus achieving much higher utilization and efficiency of these important resources. Multithreading also provides some latency hiding to local shared functional units, on-chip memory (for other processor/memory nodes on the chip), and remote service requests to external chips. The IBM Blue Gene chip and the HTMT MIND chip both will incorporate multithreading.

Advanced PIM structures like MIND, DIVA, and Blue Gene require a sophisticated execution model that binds the actions of the independent processor/memory pairs distributed throughout the PIM array into a single coherent parallel/distributed computation. An intermediate level execution model is required to organize and coordinate the management of the distributed PIM resources and the parallel tasks comprising the user applications and much of the high level system software. Three factors contributing to total system operation must be combined at this level: the application parallel code and the relative associations of its tasks and data objects, the low level PIM node operation and services, and the distributed resource management and task scheduling. This layer of abstraction is required because it permits a runtime system perspective not available at compile time and not available locally to a PIM node. It provides a target for language, compiler, runtime system, and local PIM node service support. In a very real sense, it defines the semantics of the PIM based distributed system. However, it is not a language in the sense of a human programming interface. Some of the environmental drivers influencing the behavior of the system under the

control of the intermediate model are not even available to the programmer, or the compiler for that matter, much of it being derived at runtime about the hardware status. Nonetheless, it can be depicted textually, and so its syntactical representation has many of the trappings of language, even implying attributes of user languages that might prove of value. Ultimately, actions are local even as task objectives span distributed data and processing resources. An intermediate model must map these distributed entities and goals to local resources and service mechanisms. The purpose of this technical report is to provide an initial description of such an intermediate model as a basis of investigation, evolution, and prototype implementation.

3 The Macroserver Model: A Brief Overview

The proposed intermediate level computing model for PIM-based systems is made up of a collection of cooperating encapsulated activities. We call these organized distributed activities "macroservers" and distinguish them from "microservers" devised by Jay Brockman which are local to a single PIM processor/memory node. A macroserver is an object, in the sense of object-oriented computing, although not all the properties ordinarily attributed to object-oriented execution are assigned to macroservers. A macroserver has responsibility for some of the program data, and has a set of routines that operate on that data called "methods". It also reflects an external logical interface by which other macroservers coordinate with it. These three elements, data, methods, and interface define a macroserver and establish it as the basis for organizing all computation on an array of PIMs.

The relationship between a macroserver and the underlying hardware is important to appreciate. A macroserver is a virtual named object as are the data and methods of which it is made. In principle, a given macroserver can exist on any part of the underlying physical PIMs and over time move across this physical medium as the virtual pages holding the data migrate. Support services to manage the creation, execution, interaction, and migration of macroservers are provided by a set of microserver routines available within any PIM node. This interface is an important aspect of the macroserver implementation. Macroservers cooperate by calls to each other's methods. The underlying representation of the data is transparent as it is accessed and manipulated through the methods which therefore define the data semantics. A macroserver is not in general a static object. While an application program will have a "main" macroserver that represents its beginning and end, other macroservers may be created and destroyed if the program state is highly dynamic. Macroservers can also provide system software services and may be ephemeral as well. Macroservers are first-class objects; they are named and may be manipulated by other macroservers which makes parallel system software daemons particularly easy to construct.

We continue with a more concrete overview of the key concepts and their relationships.

A macroserver comes into existence by being created as an instantiation of a parameterized template called a macroserver class, which contains declarations of variables and the methods defining its "behavior". While the hardware architecture provides a shared address space, the discipline imposed by the object-based framework requires all accesses to external data to be performed via method calls, optionally controlled through a set of access privileges. At the time a macroserver is created, a region in the virtual PIM array memory is allocated to it. This allocation can be explicitly controlled by expressing a set of constraints.

A data structure belonging to a macroserver can be distributed across the associated memory region. Such a distribution is established by binding the data structure to a first-class distribution object. Bindings can be performed dynamically and may be changed during runtime. A data distribution can also be specified indirectly, using an alignment relationship.

Threads are generated by spawning methods of a macroserver; they operate in the distributed data space of the macroserver. Similar to the (data) distribution objects introduced above we propose first-class work distribution objects that specify the mapping of a set of threads to a memory region. In most cases, work distributions are used to establish a relationship between the home of a thread – the memory unit where its arguments and private data are stored – and a memory region allocated to a segment of a distributed data structure.

Threads are lightweight in the sense that, unlike UNIX processes, they operate in the macroserver (i.e., user) data space. Threads execute asynchronously as long as they are not subject to synchronization. Mutual exclusion can be controlled via atomic methods. A macroserver whose methods are all atomic is a *monitor*

and can be used as a flexible instrument for scheduling access to resources. A "small" monitor can be associated with each element of a large data structure (such as a reservation system), co-allocating the set of variables required by the monitor with the associated element (for an example, see Section A.2). This organization allows the ASAP to perform the scheduling in a highly efficient way [1]. State synchronization can be expressed using condition variables [29], which provide a low-level efficiently implementable mechanism. Finally, future variables [26] can be bound to threads and used for implicit or explicit synchronization based upon the thread status.

4 Macroserver Classes

At execution time, a macroserver will be created to hold part of the program data and to perform useful work on that data for itself and on behalf of other macroservers. Such creation requires a definition of the template from which the macroserver is to be constructed. This definition is referred to as a "macroserver class". Many executing macroservers can be created from a single macroserver class in a similar way as many threads can be generated as separate instantiations of the same method.

More specifically, a **macroserver class** is a parameterized template for the creation of macroservers. In general, it contains the following components:

- the name of the class,
- zero or more formal parameters,
- variable declarations,
- method declarations, and
- a specification of the external interface (*acquaintances*).

At this point we provide a first overview of variable and method declarations as well as the external interface. We also discuss commands for the creation, migration, and destruction of macroservers. A well-known coordination task – the producer/consumer problem – will serve as a running example.

4.1 Variable Declarations

The specification of a macroserver class defines a set of variables that will be instantiated in every macroserver created for this class. As a general rule, these variables can only be accessed by methods declared in the class.

Our model does not assume any specific type system, and thus allows the embedding of any of the commonly used programming languages, such as Pascal, Fortran, C and C++, or Lisp. Likewise, we interpret the concept of a variable in a very broad sense. For example, a variable may be a conventional scalar or array variable as in Fortran 95, a sparse matrix, or a pointer to a Lisp data structure.

We introduce a number of new features as discussed below.

- **Data and work distribution**
  Data distributions and alignments are used to explicitly control the allocation of data structures in the memory region of a macroserver. They are managed as first-class objects which can be dynamically bound to data structures and variables. More details are discussed in Section 5.3.

  Similarly, we introduce first-class work distributions specifying mappings between threads and regions of virtual memory in which these threads are to be executed. Most often, such mappings are based upon an alignment between threads and distributed data structures. The combined features of data and work distributions allow a flexible approach to the control of parallelism and locality. Moreover, the model provides explicit management of first-class communication schedules which can be associated with a method or a region of code such as a parallel loop. Work distributions and schedules are discussed in Sections 6.6.1 and 6.6.2.

[1] This refers to the ASAP ISA, a row wide ALU developed at Notre Dame University [10].

- **The macroserver type**
  The values of the *macroserver type* are references to macroservers. Such values are generated whenever a macroserver is created from a class specification; they can be assigned to variables which then serve as handles to macroservers.

  We represent a macroserver type in the form **macroserver** ([C]), where C is a class identifier. If C is specified, then the range of values associated with the type is the set of references to all macroservers that are created based on class C. Otherwise, references to all macroservers regardless of the underlying class are in the value range.

  In terms of implementation, a macroserver value is a reference to the home (Section 5.1) of the designated macroserver. Macroserver types are directly supported by the global addressing scheme of the PIM array hardware, in particular the in-memory address translation facilities [10, 46].

- **The future type**
  Values of the *future* type are references to threads. A future value is generated whenever a new thread is spawned; at that time, it can be bound to a future variable. Once this is done, the future variable can be used to access that thread for status inquiries and thread management. Furthermore, future variables support an elegant synchronization syntax, which can be efficiently implemented in PIM arrays (see Sections 6 and 7).

  Futures are based on Multilisp [26]; see also [11].

- **The condition type**
  Condition variables are used for handling synchronization conditions in the context of a monitor-based mechanism. They are discussed in Section 7.

4.2 Method Declarations

A method is a procedure for operating on data within a macroserver. It is defined within the context of a macroserver class. A method can be called from within the macroserver or, through the external interface, by other macroservers.

The set of methods specified in a macroserver class collectively determines the *behavior* of macroservers created from that class. Methods can either be *built-in* – without being explicitly declared –, or user-defined. They are characterized by the following components and attributes:

1. The *method name*, a unique identification of the method within the name space of the macroserver class.
2. The *type* of the value, if any, yielded by an activation of the method.
3. A set of formal *method parameters*. All method parameters are input parameters subject to copy-in semantics.
4. A declaration of *private* variables. For private variables, a separate instance is created in every thread resulting from an activation of the method. We propose mechanisms for a value transfer from macroserver variables to private variables of a thread similar to those in OpenMP [47] and in [16].
5. The *method code*: a block of code that is executed when the method is activated as a thread.

Attributes that are used to characterize the properties of a method or its execution as a thread include

- the *access attribute*, which specifies if the method is only accessible from inside the macroserver (a *private* method), or also from the outside (a *public* method).
- the *atomic* attribute. This attribute specifies that the method code is executed under mutual exclusion. For a more precise specification, see Section 7.

- the *pure* attribute, specifying that the execution of the method has no side effects [18].
- the *purest* attribute[14], designating a method that is pure and, furthermore, does not require access to nonlocal entities during its execution.
- the *non-preemptive* attribute, specifying that the execution of the method, once initiated, must proceed to the end without interruption.

The types of the method value, its formal parameters, and the private variables are not restricted in the model; they include reference types as well as the additional types introduced in Section 4.1. Furthermore, these entities may be distributed or aligned in the same way as the variables of a macroserver.

The encapsulation of data and code provided by method definitions makes the method boundary an appropriate interface for dealing with different programming paradigms and languages. For example, a method may be

- a sequential procedure dealing with local data in a specific memory unit,
- a driver for a heterogeneous parallel application such as a multidisciplinary optimization,
- a schedulng routine controlling the accesses of a set of threads to a large shared data structure (for example, in a readers/writers problem),
- a data parallel Single-Program-Multiple-Data (SPMD) application such as a sparse matrix-vector product, which explicitly manages the communication between its constitutent threads, or
- an SPMD data parallel program in HPF style.

Moreover, the model is general enough to also allow the specification of very fine-grain methods such as those to be executed in the high performance processors of the HTMT architecture (see Section 9.4). The non-preemptive attribute has been introduced to deal with such methods.

In a concrete system specification using the macroserver model it may be useful to classify methods according to their code and execution complexity, and provide a corresponding range of scheduling strategies. While our model does not make any specific assumptions in this context, it provides the mechanisms for defining and managing such a classification.

Example 1 *Figure 1 describes a macroserver class,* buffer_template*, which is parameterized with an integer* size*. The class contains declarations for a data array* fifo *– the buffer data structure, a number of related auxiliary variables, and two condition variables. Two atomic methods –* put *and* get *– are defined. This example will be extended to show the creation of a macroserver (Fig. 2), and later completed to provide a full specification of a producer/consumer problem (Section 7.4).*□

4.3 Acquaintances and Visibility

The PIM array supports a global addressing scheme. As a consequence, intructions executed in an ASAP have, in principle, access to the whole name space of the application. Our model provides features that allow to restrict this freedom for software engineering as well as security purposes. The mechanisms to achieve this include encapsulation and the acquaintance relation.

First, the model enforces encapsulation in the conventional way: a thread executing in a macroserver has *direct* access only to the variables declared in the associated class and the private variables and arguments of the executed method. Variables declared in other macroservers can only be accessed via associated methods. Furthermore, methods declared in a macroserver class can be specified as *private*, excluding any reference to them from outside the macroserver.

Secondly, we use a generalized semantics of *acquaintances*, a concept originally introduced for *actors* [1], for controlling accesses to external entities. Acquaintances define the interface of a macroserver to the outside world by specifying a relation in the Cartesian product of (1) the set of methods/threads of the

```
MACROSERVER CLASS buffer_template(size)      ! declaration of the macroserver class buffer_template

  INTEGER :: size                           ! declaration of the class parameter size
! declarations of the class variables:
  REAL      :: fifo(0:size-1)
  INTEGER :: count = 0
  INTEGER :: px=0, cx=0
  CONDITION :: c_empty, c_full
   ...
CONTAINS  ! Method declarations:

 ATOMIC METHOD put(x)          ! put a data item into the buffer
     REAL :: x
  ...
 END put

 ATOMIC REAL METHOD get()      ! get a data item from the buffer
  ...
 END get

   ...
END MACROSERVER CLASS buffer_template
```

Figure 1: Skeleton of a macroserver class

macroserver, (2) access rights, and (3) the set of all existing external entities.

We are now in a position to describe which entities are visible during the execution of a thread in a macroserver:

- variables and methods declared in the associated macroserver class,
- formal parameters, private variables, and statement labels of the thread, and
- external entities – all macroserver classes, macroservers, and associated methods – as determined by the acquaintance relation.

4.4 Creation of a Macroserver

The create statement generates and initializes a new macroserver based on a given macroserver class, and returns a reference which may be assigned to a macroserver variable. As a part of the create statement, a constraint regarding the location and/or size of the PIM memory region allocated to the macroserver can be specified.

4.4.1 Regions

Let $\mathcal{M}$ denote the total PIM memory accessible to the application at a given time. $\mathcal{M}$ is a non-empty set of virtual *memory units*; for our purposes, $\mathcal{M}$ can be considered invariant [2].

A **region** is a non-empty subset of $\mathcal{M}$. A *region constraint* specifies a set of constraints over $\mathcal{M}$. Examples for such constraints include:

- an *explicit specification* of a region, $\mathcal{R}$, as a subset of $\mathcal{M}$.

  For example: $\mathcal{R}:= \mathcal{M}(p_1 : q_1, p_2 : q_2)$, if we assume that the elements of $\mathcal{M}$ are arranged in a two-dimensional grid.

[2] Note that as a result of memory failures, the mapping of $\mathcal{M}$ to the physical PIM memory may change without affecting $\mathcal{M}$.

- an *identity alignment*, specifying a region already allocated to a macroserver or a variable.
  For example, $\mathcal{R}$:= *reg*(**S**), where **S** is a macroserver and *reg(***S***)* is the associated region.
- a more *complex alignment*; for example a neighborhood relation with respect to the region associated with one or more macroservers.
- a *size specification*, indicating the number of memory units required for a region.

While the first two examples yield a unique region for the specified constraint (if defined at all), the third and fourth examples may have zero or more solutions.

4.4.2 The Create Statement

We write the create statement in the form[3]

$$\textbf{CREATE}\,(C, a_1, \ldots, a_n, cstr, sv)$$

where,

- C is a macroserver class with formal parameters $x_1, \ldots, x_n$ $(n \geq 0)$,
- $a_1, \ldots, a_n$ are argument expressions conforming to the respective formal parameters,
- *cstr* is a region constraint with at least one solution, as discussed in Section 4.4.1, and
- *sv* is a status variable which is used to return status information regarding the success and effect of the execution of the create statement.

The execution of the statement creates a new macroserver, **S**, as follows:

1. Solve *cstr* and select a region, $\mathcal{R}$, from the set of solutions of *cstr*.
2. Define $\mathcal{R}$ as the *region* of **S**: *reg*(**S**) := $\mathcal{R}$.
3. Define the *home* of **S**, h(**S**), by selecting a distinguished location in *reg*(**S**).
4. Evaluate the arguments $a_1, \ldots, a_n$, yielding $a'_1, \ldots, a'_n$.
5. Allocate space in $\mathcal{R}$ for
   - the formal parameters $x_1, \ldots, x_n$,
   - the variables declared (implicitly or explicitly) in C,
   - a heap and additional storage areas required in **S**.

   Note that the above allocation may depend on the argument values.
6. Assign the argument values, a'_i, to the corresponding formal parameters, $x_i, 1 \leq i \leq n$.
7. Initialize variables as necessary.
8. The value yielded by the execution of the create statement is a reference to **S**, pointing to h(**S**).

Information about the effect of the create statement can be recorded in the status variable, *sv*. If an error occurs during any of the above steps, the execution of the create statement aborts, yielding the value undefined.

In certain contexts it may be necessary to create not just one macroserver at a time but a structured, parameterized set of similar objects. Special constructs for this have been defined for actor languages [49].

Example 2 *We continue the example of Figure 1 by adding the declaration of a macroserver variable and creating an instance of the macroserver class* buffer_type. *See Fig. 2.* □

[3] Except for the first these components are optional as discussed in the text. We ignore this in the semi-formal syntax described here to simplify the presentation, but use it in examples. This approach will also be taken for other syntactic constructs in the rest of the paper.

```
MACROSERVER CLASS  buffer_template(size)

   INTEGER :: size
   REAL    :: fifo(0:size-1)
   INTEGER :: count = 0
   INTEGER :: px=0, cx=0
   CONDITION :: c_empty, c_full
     ...

 CONTAINS

  ATOMIC  METHOD put(x)
    REAL :: x
    ...
  END

  ATOMIC  REAL METHOD get()
    ...
  END

     ...
END MACROSERVER CLASS  buffer_template

 ! Main program:
INTEGER   buffersize, status
MACROSERVER (buffer_template)  my_buffer          ! declaration of the macroserver variable  my_buffer
READ (buffersize)
my_buffer= CREATE (buffer_template, buffersize,  M(p1 : q1,p2 : q2), status)     ! This creates a macroserver
          ! which is an instance of class  buffer_template, in a rectangular memory area. The new macroserver
          ! is parameterized with the value of  buffersize; a reference to it is assigned to  my_buffer.
  ...
CALL  my_buffer%put(...)    ! Synchronous call of the method  put  in the macroserver associated with  my_buffer
  ...
```

Figure 2: Creation of a macroserver

4.5 Macroserver Management

The model currently provides two commands for the management of macroservers. We discuss migration and destruction below; additional commands may be introduced to deal with *persistency*, allowing the saving and retrieval of macroservers in long-term storage [15].

4.5.1 Migration

An existing macroserver may be migrated in the virtual memory space:

MIGRATE (*mv*, *cstr*, *sv*)

where *mv* is a macroserver variable referring to a macroserver **S**, *cstr* is a region constraint with at least one solution (Section 4.4.1), and *sv* is a status variable. The execution of the migrate statement results in the transfer of the representation of **S** to a new region of the virtual memory space, as determined by *cstr*. This requires an update of all links involving the macroserver.

4.5.2 Destruction

A macroserver can be deleted by applying a destroy statement:

DESTROY (*mv*, *sv*)

where *mv*, **S**, and *sv* have the same meaning as above. The execution of the destroy statement results in the termination of all active threads of **S** and the release of all resources occupied by it.

5 Macroservers

This section provides an overview of the components and attributes characterizing a macroserver at runtime, with a special focus on data distribution and alignment. Whenever we talk about a macroserver, there is an implicit understanding that we discuss its properties at a given point in time. Many components – for example, the set of allocated variables, their distributions, and the set of threads operating in the macroserver – may change over time.

Definition 1 *A* **macroserver**, **S**, *is a tuple*

$$\mathbf{S} = (C, reg, h, \mathcal{V}, M, A, \mathcal{T})$$

where,

1. C *is the underlying class,*
2. reg *denotes the region allocated to the macroserver,*
3. h *is the home,*
4. $\mathcal{V}$ *is the variable specification,*
5. M *is the set of methods,*
6. A *is the acquaintance relation, and*
7. $\mathcal{T}$ *is the set of threads.* □

Given **S** as above, we explain below the meaning of some components based on the discussion in Section 4. Threads will be discussed in Sections 6 and 7.

5.1 Memory Region and Home

At the time of its creation, a region, *reg*, of the virtual address space is allocated for a macroserver; the home, h, is a distinguished location in *reg* (see Section 4.4).

At the home, key information about the macroserver is stored, together with code for its management. This information is sometimes referred to as "metadata"; it contains all information required to fully access the representation of the macroserver in the memory.

The PIM array provides direct hardware and software support for this organization: a *microserver*, at the node of the home, can be made to represent the key data of the macroserver and the associated management routines, which can be activated using *parcels* [46]. If the macroserver contains a distributed data structure, a corresponding set of distributed microservers has to be set up for the management of the data structure's components.

5.2 Variable Specification

The *variable specification* associated with a macroserver is a triple,

$$\mathcal{V} = (V, state, \delta)$$

where,

- V is a finite set of *variables*,
- the *state* of V binds variables to their values, and
- δ, the *distribution* of $\mathcal{V}$, specifies for each variable a mapping to a set of memory units in $reg(\mathbf{S})$.

At the time of macroserver creation, V is generated by instantiating the variable declarations in class C; at that time, V may be bound to a data structure, and an initial state as well as an initial distribution may be defined (Section 4.4). During the execution of threads in the macroserver, V as well as its state and distribution may be modified.

5.3 Data Distribution and Alignment

5.3.1 Basic Concepts

In this section we develop a set of basic abstractions underlying data structures, data distributions, and data alignments relevant in the context of our discussion. We generalize the approach adopted in Vienna Fortran [59] and later HPF [31] in the context of data parallel SPMD languages by

1. developing an abstract language-independent framework for data distribution and alignment, targeted to arbitrary memory regions,
2. generalizing the set of data structures to which distributions can be applied,
3. generalizing the distribution mechanism to include arbitrary mappings, dynamic data structures, and incremental redistributions, and
4. making distributions first-class objects.

In combination with a facility for binding threads to data (Section 6.3) our scheme can support data and work distribution as well as migration for a broad range of parallel processing strategies.

We associate with a *data structure* a mapping, $D : \mathbf{I} \rightarrow \Omega$, where $\mathbf{I}$ is an *index domain*, and Ω is some "universal" set of values [4]. The idea here is that we can always decompose a data structure into its "atomic" components, which designate elementary values such as fixed-point or floating-point numbers, logical values, or pointers, and that each of these components can be 1-1 mapped to a unique "name" in $\mathbf{I}$. For example,

[4] For the purpose of discussing distributions we take this limited view, not dealing explicitly with such information as the topology of the data structure.

if D is a simple numerical variable, then we can choose for the index domain the singleton set $\mathbf{I} = \{1\}$. If $D(1:n, 1:m)$ is a two-dimensional Fortran array, then we define $\mathbf{I} = [1:n] \times [1:m]$. If D is an n-ary tree of height m, then each leaf can be uniquely identified by a string $i_1.i_2 \ldots .i_k$, where $k \leq m$ and all i_j are integers between 1 and n. In this way, we can represent data structures associated with arbitrary graphs if we include $\mathbf{I}$ as a subset of Ω in order to be able to deal with pointers.

Based upon the notion introduced above we are now in a position to define a data distribution as a mapping from the index domain of a data structure to the powerset of a memory region.

Definition 2 Data distribution
Let D denote a data structure, $\mathbf{I}$ its index domain, and $\mathcal{R}$ a memory region. A **data distribution**, *δ^D, for D is a total function* [5]

$$\delta^D : \mathbf{I} \rightarrow \mathcal{P}(\mathcal{R}) - \{\phi\}.$$

□

Assume $\mathbf{i} \in \mathbf{I}$ and $\delta^D(\mathbf{i}) = \{u_1, \ldots, u_n\} \subseteq \mathcal{R}$, where $n \geq 1$. Then the data structure element $D(\mathbf{i})$ is mapped to each memory unit $u_i, 1 \leq i \leq n$, meaning that each u_i stores a representation of $D(\mathbf{i})$[6]. The pair (D, δ^D) is called a **distributed data structure**.

Definition 2 specifies mappings of individual indices to *subsets* of $\mathcal{R}$ rather than to single units. The purpose of this is to be able to deal with replication. However, for practical purposes we will mostly use *replication-free* distributions, which are constrained by requiring $\mid \delta^D(\mathbf{i}) \mid = 1$ for all $\mathbf{i} \in \mathbf{I}$. Such data distributions can be considered total functions that map indices to memory units in $\mathcal{R}$: $\delta^D : \mathbf{I} \rightarrow \mathcal{R}$.

Note that the definition of a data distribution, δ^D, depends only on the index domain associated with the data structure and none of its other properties. This allows us to interpret a distribution as a distribution of an index domain, $\delta^{\mathbf{I}}$, with the further consequence of being able to associate a distribution with more than one data structure with identical index domains. This has important practical consequences for the internal representation of distributions and the associated access mechanism to their components. In effect, we treat distributions $\delta^{\mathbf{I}}$ as first-class objects that can be dynamically bound to data structures and variables (see Section 5.3.3).

Definition 3 Distribution Segment
Assume D, $\mathbf{I}$, and $\delta^D : \mathbf{I} \rightarrow \mathcal{P}(\mathcal{R}) - \{\phi\}$ are given as above. Further assume $u \in \mathcal{R}$ to be an arbitrary memory unit. Then the set of indices mapped to u is called the **distribution segment**, *$\lambda^D(u)$, of u:*

$$\lambda^D(u) := \{\mathbf{i} \in \mathbf{I} \mid u \in \delta^D(\mathbf{i})\}$$

u is called a **home** *for all elements of D whose index is in the distribution segment.* □

We illustrate distributions with a few simple examples.

Example 3 Distribution of a scalar data structure
Assume D is a scalar data structure with index domain $\mathbf{I} = \{1\}$, specifying one component. Examples for possible distributions include

- *δ_1^D, with $\delta_1^D(1) = \{u\}$ for some $u \in \mathcal{R}$.*
- *δ_2^D, with $\delta_2^D(1) = \mathcal{R}$.*

In the first case, the data structure is mapped to exactly one memory unit, u. In the second case, it is totally replicated, *i.e., a copy exists in each memory unit that belongs to $\mathcal{R}$.* □

[5] $\mathcal{P}(X)$ denotes the powerset of a set X, i.e., the set of all subsets of X.
[6] A *memory unit* is either a single virtual page or a set of pages subject to a physical locality constraint.

Example 4 Distribution of a Fortran array
Assume $D = A(1:n, 1:m)$ is a two-dimensional Fortran array, and $\mathcal{R} = \mathcal{M}(1:q)$, where n is a multiple of q, $n = b * q$. A row block distribution *establishes a mapping $\delta^D(i,j) = \{\mathcal{M}(\lceil \frac{i}{q} \rceil)\}$ for all $i, 1 \leq i \leq n$, and all $j, 1 \leq j \leq m$. This distribution maps contiguous blocks of b rows to subsequent memory units; it corresponds to a standard distribution that can be easily expressed in data parallel languages.* □

Example 5 Distribution of a tree structure
Assume D is a small binary tree with index domain $\mathbf{I} = \{1, 1.1, 1.2, 1.2.1, 1.2.2\}$, and $\mathcal{R} = \{u_1, u_2, u_3\}$. A possible distribution, δ^D, could be defined as $\delta^D_1(1) = \delta^D_1(1.1) = \{u_1\}$, $\delta^D_1(1.2) = \delta^D_1(1.2.1) = \{u_2\}$, and $\delta^D_1(1.2.2) = \{u_3\}$.
Such a distribution cannot be expressed directly by the current data parallel languages. □

As we will see later (Section A.3), the mechanism introduced here is general enough to also deal with sparse data structures.

5.3.2 Distribution Functions and Libraries

Existing data parallel languages [59, 19, 31, 14] propose a range of distribution functions that are expected to be also useful in the context of macroservers. These standard distributions include *block*, *general block* and *cyclic* as well as *indirect* distributions which allow arbitrary, replication-free array mappings for the support of irregular algorithms.

Our concept of a data distribution is more general because of the generality of the underlying data structures and the mapping to arbitrary memory regions rather than only rectilinear processor structures. Furthermore, we expect new classes of distributions, such as random distributions, to become relevant for PIM arrays.

For practical purposes, we assume that a set of basic distributions will be complemented by a library of special distribution functions tied to classes of data structures such as n-ary trees, forests, particular graph structures, and various categories of sparse matrix representations.

5.3.3 Distribution Management

We now establish a link to the discussion of Section 5.2. At the time a variable is used in a computation, it will be bound to a data structure. If, in a language, this binding is etablished just once and remains invariant therafter, we speak of a *fixed-structure variable*. For example, all Fortran 95 variables that are not allocatable and do not involve pointers are of that type. For such variables, the index domain of the associated data structure can actually be specified in the declaration of the variable.

A variable which can be bound to different data structures during execution is called a *variant-structure variable*. Lisp, C++, and the full Fortran 95 language offer variables of that kind.

If we speak of the "distribution of a variable" in the following, we always mean the distribution of the data structure to which the variable is currently bound. This is independent of whether or not the variable is variant-structure; however, for a fixed-structure variable an (initial) distribution can be specified in its declaration if the distribution is statically known. As mentioned before, distributions are first-class objects; during execution, a given variable – even if it is fixed-structure – may be *redistributed*, i.e., the data structure to which it is bound may be dynamically associated with a new distribution. An important special case of redistribution that is not dealt with in current data parallel languages is *incremental redistribution*, which changes the distribution of a data structure only within a given (small) area of the index domain. Explicit optimization targeting this case in a compiler and runtime system can result in a significant improvement of redistribution efficiency. The incremental modification of a data structure and its binding to a distribution which results from an incremental change of the original distribution can be also seen in this context.

The actual details of the allocation of a data structure in a PIM memory region depend on the distribution, the types of the data structure's components, and other, implementation-dependent parameters (such as the specific choice of representation for a sparse matrix).

The following section outlines a number of functions and commands that deal with data distributions.

5.3.4 The Distribute Statement

The distribute statement is written as

$$\textbf{DISTRIBUTE}\,(v, dv, sv)$$

where,

- $v \in V$ is a variable bound to a data structure D with index domain $\mathbf{I}$,
- dv is a distribution variable whose value is a data distribution δ based on the index domain $\mathbf{I}$, and
- sv is a status variable.

The effect of an execution of the distribute statement is the establishment of a binding between D and δ, resulting in a distributed data structure (D, δ).

The distribute statement can be applied in two different contexts:

1. The distribute statement is tied to the allocation of the data structure, i.e., the generation of D and its distributed allocation according to δ are performed hand in hand.
2. At the time the distribute statement is executed, D is already distributed according to some distribution $\delta' : (D, \delta')$. In this case, the distributed data structure (D, δ') is transformed into the new distributed data structure (D, δ). We speak in this case of a *redistribution* of D.

A variant of the above statement performs an *incremental* redistribution. This is written as

$$\textbf{INC\_REDISTRIBUTE}\,(v, dv, sv)$$

where, v and sv have the same meaning as above, but the value of dv is a distribution, δ_{inc}, based on an index domain $\mathbf{I}_{inc} \subseteq \mathbf{I}$. Here, only case 2 from above applies, i.e., at the time the incremental redistribution is performed, D must already exist as a distributed data structure (D, δ'). The effect of the incremental redistribution is a distributed data structure (D, δ), where $\delta(\mathbf{i}) = \delta_{inc}(\mathbf{i})$ for all $\mathbf{i} \in \mathbf{I}_{inc}$ and $\delta(\mathbf{i}) = \delta'(\mathbf{i})$ for all other $\mathbf{i} \in \mathbf{I}$.

Incremental redistribution can play an important role in improving the efficiency of redistribution, if only a small number of indices is affected.

5.3.5 Distribution Inquiries

Distribution inquiries are pure functions which are applied to a distribution variable or a distributed data structure and yield selected properties of the distribution. For example, properties that may characterize the distribution, δ^A, of a two-dimensional array $A(N, M)$ include the number of distributed dimensions, the memory region involved in the mapping, and the sizes or topologies of the distribution segments associated with specific memory units.

Of special importance are functions that allow a thread to identify and access those parts of a data structure that are local to the memory unit in which the thread is executing. More specifically, a thread operating in unit u on a data structure D must be able to identify the local distribution segment, $\lambda^D(u)$, of u, and access it via appropriate syntactic mechanisms.

5.3.6 Alignment

If (D_1, δ^{D_1}) and (D_2, δ^{D_2}) are distributed data structures processed in a common context, then δ^{D_1}, δ^{D_2} and their relationship determine the degree of parallelism and locality in the algorithm. As a simple example, if D_1 and D_2 are matrices with the same index domain whose sum has to be computed and assigned to a third matrix, then distributing all three matrices identically results in completely local operations executed in parallel in all memory units involved in the distribution.

In general, we say that two distributed data structures are *aligned* if, during a phase of the execution, an alignment relationship, as described below, is established between their distributions. We distinguish two types of alignment, fine-grain and coarse-grain. Fine-grain alignment enforces the mapping of two different data structures to the same set of *virtual* memory units. This is formally defined below.

In contrast, coarse-grain alignment establishes a physical locality constraint for two or more (possibly disjoint) regions associated with a set of data structures. The details of this feature are not further discussed here.

Definition 4 Fine-grain alignment
Let D_1, D_2 denote data structures with respective index domains $\mathbf{I}_1$ and $\mathbf{I}_2$.

1. *A total mapping*

$$\alpha : \mathbf{I}_1 \rightarrow \mathcal{P}(\mathbf{I}_2) - \{\phi\}$$

   is called a **fine-grain alignment** *for the* **target data structure** D_1 *with respect to the* **source data structure** D_2.

2. *Given α and a distribution δ^{D_2} for the source data structure, then the* **aligned distribution**, *δ^{D_1}, for the target data structure is determined as follows. For each $\mathbf{i} \in \mathbf{I}_1$*

$$\delta^{D_1}(\mathbf{i}) := \bigcup_{\mathbf{j} \in \alpha(\mathbf{i})} \delta^{D_2}(\mathbf{j})$$

□

In other words, the distribution for D_1 is constructed by mapping each element $D_1(\mathbf{i})$ to each memory unit to which any $D_2(\mathbf{j})$ is mapped in δ^{D_2}, where $\mathbf{j}$ ranges over all values in the non-empty set $\alpha(\mathbf{i})$.

While alignments per se are highly important in the context of macroservers, their generality is far less important than in a language such as HPF which used sophisticated alignments to partially offset the weakness of the distribution concept offered by the first version of the language. Since general alignment functions are very difficult to implement efficiently [8], we focus on identity alignment for isomorphic data structures as well as collapsing, replicating, or permutating substructures such as array dimensions or subtrees.

5.3.7 The Align Statement

We write the align statement in the form

$$\textbf{ALIGN}\ (v_1, v_2, av, sv)$$

where,

- $v_1, v_2 \in V$ are variables bound to data structures D_1 with index domain $\mathbf{I_1}$, and D_2 with index domain $\mathbf{I_2}$.
- av is an alignment variable whose value is a fine-grain alignment α establishing a mapping from $\mathbf{I_1}$ to the powerset of $\mathbf{I_2}$ (Def.4).
- sv is a status variable.

At the time the align statement is executed, D_2 must exist as a distributed data structure (D_2, δ_2). The effect of its execution is the establishment of a distributed data structure (D_1, δ_1), according to the rules set forth in Def. 4, Part 2.

As in the case of the distribute statement, the align statement may be either tied to an allocation of the data structure D_1, or it may have the effect of a redistribution, if D_1 already exists as a distributed data structure.

5.4 Methods

At the time of its creation, the "behavior" of a macroserver is defined by the set of built-in methods and the user-defined methods introduced in the class declaration (Section 4). During its lifetime, this initial behavior may be modified. A particularly interesting case which is directly supported by the PIM hardware [46] is that of *transient methods*: they are imported into a macroserver as a part of an activation. More specifically, in this case, the activation carries a special argument which contains either the code of the method to be activated or a pointer to such a method – rather than referring to a built-in or user-defined method of the macroserver class.

Method names are required to be unique in the namespace of a macroserver class. However, the same method name may be used in different macroserver classes; moreover, it is necessary to distinguish between activations of the same method in different macroservers based on the same class. We can avoid ambiguity by qualifying the method name with the identification of a macroserver: the pair $mid = (\mathbf{S}, m)$, where $\mathbf{S}$ is a macroserver and m a method name, provides a system-wide unique method identification.

6 Threads

A complex application may display parallelism at many levels of abstraction within a dynamic hierarchy of activities. For example, a multidisciplinary optimization for the design of an aircraft [44] will have a coarse-grain layer of heterogeneous task parallelism, combining modules for structural computation, aerodynamics, analysis, and optimization with respect to some objective cost function. Within a module, large-scale linear equation systems may have to be solved, leading to module-internal levels of data parallelism. At still lower levels, the fine-grain parallelism of vector operations or scalar expressions may be exploited.

A PIM array, as the target architecture considered in this work, offers many levels of parallelism ranging from the inter-node parallelism at the top level all the way to the parallel processing of wide words in an individual ASAP. In order to allow an efficient mapping from application to target parallelism, the execution model must provide support for method definitions across a broad range of complexity and for the control of the distribution of data across the PIM memory, as discussed previously. Furthermore, there must be a flexible mechanism for spawning threads at different levels of abstraction, and binding their locus of execution to the home of data.

A thread is an autonomous dynamic entity, coming into existence as a result of the spawning of a method in a macroserver. For any given macroserver, $\mathbf{S}$, $\mathcal{T}(\mathbf{S})$ specifies the set of threads operating in $\mathbf{S}$ at a given point in time (Section 5). All threads in $\mathcal{T}(\mathbf{S})$ are considered peers, having the same rights and sharing all resources allocated to the macroserver. Different threads – within one or different macroservers – may execute asynchronously in parallel unless subject to synchronization constraints (see Section 7).

A thread ends its existence when the associated method execution finishes or when it is explicitly terminated – by an action of its own or of another thread. Termination may be regular or irregular. At the time of (regular) termination, a thread yields a value if the method it is executing is declared with a type specification.

At the time of thread creation, a future variable may be bound to a thread. This variable can be used to make inquiries about the status of the thread, retrieve its attributes, synchronize the thread with other threads, and access its value after termination. A thread which is not accessible via futures is called detached.

In contrast to UNIX processes, threads are lightweight, living in macroserver, i.e., user space. Depending on the actual method a thread is executing, it may be ultra lightweight, carrying its context entirely in registers. On the other hand, a thread may be a significant computation, such as a sparse matrix vector multiply, with many levels of parallel subthreads.

The following subsections make our concept of a thread more precise by defining an abstract thread specification, proposing a set of pure thread functions, introducing thread groups, and outlining mechanisms for the spawning and terminating of threads. Finally, we outline work distributions and communication schedules, both of which are dealt with as first-class objects in our model.

Note that the features described in this section are only intended to provide a framework and a guideline for a full specification in the context of an actual system implementation. Among the issues that are left deliberately open are high level parallel language constructs (for example, variants of parallel loops and parallel regions), the details of thread group management, and the interface with data parallel SPMD computations. Also, any real system implementing macroservers will conceivably impose a hierarchical scheme on threads by differentiating them according to the complexity of their methods, the size of their specification, attributes such as *non-preemptive* or *atomic*, and expected runtimes.

6.1 Thread Specification

An abstract *thread specification*, as described below, contains the components and attributes that characterize the execution behavior of a thread and the relationship to its environment. Not all of this representation need to be stored in the macroserver memory; for example, the whole specification of ultra-lightweight threads may be kept in machine registers. We assume that each thread in the system is assigned a unique identification.

Definition 5 Thread Specification
A **thread specification** *is a tuple*

$$\mathbf{t} = (mid, h, f, \mathbf{a}, status, result, \mathcal{V}_\mathbf{t})$$

where,

1. mid *is the unique identification of the method which* **t** *is executing*
2. *h, the* home *of the thread, is the memory unit in which the thread is to be executed.*
3. f *is a future variable bound to the thread.*
4. **a**, *the* **input vector**, *contains the argument values.*
5. status *specifies attributes and actual status information for* **t**.
6. result *is the container for the result value of the thread (if any).*
7. $\mathcal{V}_\mathbf{t}$ *is the specification of the private variables of* **t**.

The components f, **a**, *and* result *are optional.* □

Here and in the following, the "memory unit in which a thread is executing" is to be understood as the memory unit in which its arguments and private data are kept. Nothing is said about the allocation of the associated method code.

6.1.1 Thread Status

The **status** of a thread **t** at a given point in time provides information about its attributes, the state of its progress, its scheduling priority, potential detachment, and the existence of a blocking or error condition. More specifically, it includes the following components:

- *thread attributes*: the *atomic* and *non-preemptive* attributes as specified in the method declaration (Section 4.2).
- *completion_status*: The completion status is *pending*, if the execution of **t** has not yet terminated, otherwise *completed*.
- *blocking_status*: The blocking status is *blocked* if the thread is waiting for a synchronization condition (Section 7.2), the termination of one or more threads (Section 7.3.1), or the release of a mutual exclusion lock; else *active*.
- *error_status*: The error status is set if an error condition occurs during its execution.

- *scheduling_status*: The scheduling status specifies, in an implementation-specific way, the priority of a thread and other information relevant for its scheduling.
- *detachment_status* is **true** iff the thread is detached, i.e., it cannot be externally controlled via a future variable.

At the time a thread is created, the components of the status are set as follows: the *atomic* and *non-preemptive* attributes are set as defined in the method declaration; *completion_status:=pending*, *blocking_status:=* **false**, and *error_status:=* **false**. The *scheduling_status* is initialized in an implementation-dependent way, and the initial *detachment_status* is determined by the spawn statement initiating the thread.

6.1.2 Private Variables

A method may specify a set of private variables. Whenever a thread is generated by activating the method, a thread-specific instance of these variables is generated. A language may specify mechanisms for transferring the value of a macroserver variable to a conforming private variable at the time of thread spawning [47].

The instruction count, the stack, synchronization queues (Section 7) and other auxiliary data structures of a thread are considered implicitly declared private variables.

6.2 Pure Thread Functions

In this section we provide a collection of pure functions dealing with threads. A call to one of these functions can never result in the blocking of the executing thread.

Let f, f' denote future expressions whose values respectively reference threads **t**, **t'**.

- **completed** (f): predicate that is satisfied iff the *completion_status* of **t** has the value *completed*
- **blocked** (f): predicate that is satisfied iff the *blocking_status* of **t** has the value *blocked*.
- **error** (f): predicate that is satisfied iff the *error_status* of **t** is **true**.
- **detached** (f): predicate that is satisfied iff the *detachment_status* of **t** is **true**.
- **non-preemptive** (f): predicate that is satisfied iff **t** is declared a non-preemptive thread.
- **priority** (f): integer function yielding the priority of thread **t**.
- **equal** (f, f'): predicate that is satisfied iff the threads **t** and **t'** are identical.
- **who_am_I** (): future function yielding a reference to the executing thread.

6.3 Spawning of Threads

We write a spawn statement in the form

$$[f =] \ \textbf{SPAWN}\ (\mathbf{S}, m, arg_1, \ldots, arg_n, status\_input, h)$$

where,

- f is a future variable,
- **S** is a macroserver in which the new thread is to be executed,
- m is a method identification: it can either specify a method declared in **S**, or a *transient* method (see Section 4.2),
- $arg_1, \ldots, arg_n$ is the list of arguments for the method execution,

- *status_input* provides optional information regarding the status of the thread to be generated, such as priority, and
- h is the home of the thread.

Assume the above statement is executed in a thread **t'**. Then the following steps are performed:

1. Generate a new thread, $\mathbf{t} = ((\mathbf{S}, m), h, f, \mathbf{a}, t\_status, result, \mathcal{V}_\mathbf{t})$, and determine a unique identification for **t**.
2. Initialize *t_status* as specified by *status_input* and in Section 6.1.1. The *detachment_status* of the thread is **true** iff a future variable is *not* specified [7].
3. Compute the values of the arguments, arg_i, and assign them to the corresponding elements of the argument vector, **a**.
4. Create an instantiation of the private variables specified in m and identify this instance with $\mathcal{V}_\mathbf{t}$.
5. Add **t** to the thread set, $\mathcal{T}$, of **S**.
6. Start with the execution of method m. The spawning thread, **t'**, proceeds asynchronously in parallel.

6.4 Termination of Threads

We write the terminate statement in the form

TERMINATE (f)

where f is a future variable. The effect of the terminate statement is to end the execution of the thread designated by f and release the resources occupied by the thread. The thread status and its value, if any, are retained as long as there is a future variable referencing the thread.

More specifically, let $\mathbf{t} = (mid, h, f, \mathbf{a}, status, result, \mathcal{V}_\mathbf{t})$ denote the thread to be terminated. The following actions take place.

1. The execution of **t** is discontinued.
2. The *status* of the thread is updated.
3. The result value, if any, of the thread is stored in *result* (if the thread ends in an error status, this value may not be well-defined).
4. The resources allocated to **t** are released.
5. **t** is deleted from the thread set, $\mathcal{T}$, of the associated macroserver.
6. If f occurs in future expressions associated with one or more blocked threads, these expressions are re-evaluated, and, if they yield **true**, all associated threads are released (see Section 7.3.1). If f occurs as a term in a "regular" expression of the language, then f is replaced in that expression by *result*.

As long as there exists a future variable referring to **t**, a residual representation of **t** is retained, allowing synchronization and access to its status components.

[7] The model may provide a command to change a *detachment_status* from **false** to **true** during runtime, but not vice versa [45].

6.5 Thread Groups

Until now our discussion focused on individual threads and their creation and management. In many situations, a parallel algorithm can be simplified if the execution model provides support for dealing with a set of threads, abstracting from details of the individual threads. Examples include a heterogeneous set of modules cooperating in a multidisciplinary optimization, searches in a tree data structure, or relaxation algorithms where a stencil operation is applied independently to each element (or submatrix) of a matrix.

A system supporting such *thread groups* must provide features for dynamically defining the membership of a group, associating it with a name and establishing a scope for collective group operations. Such operations may include (1) spawning, (2) termination, (3) pure thread functions such as **completed** or **error**, (4) broadcast and multicast, (5) reduction and prefix operations, and (6) synchronization.

A particularly important special case is represented by the Single-Program-Multiple-Data (SPMD) data parallel paradigm, where a set of threads, all executing the same method in a "loosely synchronous" manner, are applied to disjoint segments of distributed data structures. This paradigm, which has proven to be highly important for programming distributed-memory multiprocessing systems (DMMPs) is also relevant for a significant set of applications for a PIM array. Note that the level of abstraction at which the SPMD paradigm is actually used is an orthogonal issue. This may include including low-level MPI and PVM based approaches as well as the higher-level HPF paradigm. Our model is open for any of these programming approaches, providing essential support via its data and work distribution capabilities.

We do not pursue the general discussion of thread groups any further here. Also, we do not propose a syntax for high-level language features supporting thread groups such as the independent loops of HPF [31] or the parallel region and work distribution concepts of OpenMP [47]. However, we will use thread groups in some examples; a syntax for collective parallel spawn and termination is presented in the subsections below.

6.5.1 Spawning a Set of Threads

We use a variant of the Fortran 95 forall statement to indicate the parallel creation of a set of threads all executing the same method. We illustrate this facility with a slightly simplified code fragment from Fig. 3:

FORALL THREADS (I=1:100, J=1:100, **ON HOME** (A(I,J)))
F(I,J)= **SPAWN** (intra_block_transpose,I,J)

This statement has the following effect:

- 10000 threads, say $\mathbf{t}(I, J), 1 \leq I, J \leq 100$, are created in parallel.
- Each thread $\mathbf{t}(I, J)$ activates method *intra_block_transpose* with arguments I and J.
- For each I and J, thread $\mathbf{t}(I, J)$ is executed in the ASAP which is the home of array element $A(I, J)$.
- For each I and J, the future variable $F(I, J)$ is assigned a reference to $\mathbf{t}(I, J)$.

6.5.2 Terminating a Set of Threads

A syntactic variant, similar to the one described for the spawning of threads, allows the termination of a set of threads:

FORALL THREADS (I = ..., J= ...) **TERMINATE** (F(I,J))

6.6 Work Distributions and Communication Schedules

Many important applications for PIM array architectures use dynamic data structures whose size and distribution is not known until execution time, resulting in the need to dynamically balance the work performed in the computation depending on input data or intermediate results. Examples for such applications include particle-in-cell codes, adaptive finite-element computations, or sweeps over unstructured grids.

Such applications must cope with dynamic bindings of important parameters that may deeply affect their performance: data distributions and alignments, the number of threads spawned for the execution of a particular method, the mapping of threads to their homes, and the communication required for the execution of a particular program region such as a method or a parallel loop. As a consequence, our model must provide a flexible scheme for the dynamic management of such parameters in an adaptive environment.

An important component for satisfying this requirement is provided by the facilities for data distribution and alignment (Section 5.3). In this section, we develop a formal framework for the mapping of threads to memory units, which generalizes the features already introduced in the discussion of thread spawning above.

6.6.1 Work Distributions

We introduce work distributions as mappings from a set of threads to a memory region:

Definition 6 Work Distribution
Let $\mathbf{T}$ *denote a set of threads, and* $\mathcal{R}$ *a region of virtual memory.*
A **work distribution** *for* $\mathbf{T}$ *is a total mapping* $\omega^{\mathbf{T}} : \mathbf{T} \rightarrow \mathcal{R}$. *For each* $\mathbf{t} \in \mathbf{T}$, $\omega^{\mathbf{T}}(\mathbf{t})$ *specifies the memory unit in which thread* $\mathbf{t}$ *is to be executed.* □

The spawn commands discussed in Sections 6.3 and 6.5.1 provide a means for establishing a specific work distribution for the involved threads. Since we treat work distributions as first-class objects, the *home* of a thread occurring in these constructs can actually be replaced by a work distribution variable whose value specifies the home. Threads can be migrated by dynamically associating them with a new home.

Most often, work distributions are not specified directly but rather via an alignment of a thread set with a data distribution.

Let $\mathbf{T}$ denote a set of threads, and (D, δ) a replication-free distributed data structure with domain $\mathbf{I}$ and range $\mathcal{R}$. A **work alignment** for $\mathbf{T}$ is a total mapping, $\beta : \mathbf{T} \rightarrow \mathbf{I}$. Given such a mapping, a work distribution, $\omega^{\mathbf{T}}$, is determined by

$$\omega^{\mathbf{T}}(\mathbf{t}) = \delta(\beta(\mathbf{t})) \text{ for all } \mathbf{t} \in \mathbf{T}$$

Languages such as HPF [31] or OpenMP [47] use high-level constructs such as on-clauses and "iteration chunks" for associating processors with the threads tied to a parallel loop construct. Such constructs can be easily mapped to our work distributions, which provide a more general facility for allocating work.

6.6.2 Schedules

We introduce schedules to provide a concise specification of the gather or scatter communication required to deal with a data structure in a certain section of code. More specifically, assume B to be a block of code, and $\mathbf{t}$, with home h, a thread executing B. For example, B could be a parallel loop and $\mathbf{t}$ a thread executing one iteration of that loop. Further assume that there exists no interference between $\mathbf{t}$ and any other thread while $\mathbf{t}$ is executing B.

Given this situation, the execution of $\mathbf{t}$ in memory unit h may encounter a number of non-local read or write accesses. Then, due to the absence of dependences, the semantics of the execution is not modified if all communication for non-local reads is performed immediately *before*, and all communication for non-local writes is performed immediately *after* B. Under the assumption that the collective communication for a large number of data items may be more efficient than separate communications for the individual items, the program transformation described above yields an improved program. This is the background for the introduction of schedules, which essentially define the collective gather or scatter communication respectively required at the beginning or end of a code section for a given distributed data structure [52, 51, 7].

Definition 7 Schedule function
Assume $\mathcal{R}$ *is a memory region (Section 4.4.1), and* $\mathbf{I}$ *is an index domain.*
A **schedule function** *is a total function* $\sigma : \mathcal{R} \rightarrow \mathcal{P}(\mathbf{I})$. □

Assume that $h \in \mathcal{R}$ is the home of a thread $\mathbf{t}$, and $\sigma(h) = \{\mathbf{i}_1, \ldots, \mathbf{i}_n\}$, where $n \geq 1$. This can be used to express the fact that the execution of $\mathbf{t}$ in h accesses a set of non-local objects whose indices are given by $\mathbf{i}_1, \ldots, \mathbf{i}_n$. This observation leads us to the definition of a data structure schedule.

Definition 8 Data structure schedule

Assume (D, δ) is a distributed data structure with index domain $\mathbf{I}$, $\mathcal{R} \subseteq range(\delta)$ is a memory region, and $\sigma : \mathcal{R} \to \mathcal{P}(\mathbf{I})$ is a schedule function. Further assume that for each $u \in \mathcal{R}, \sigma(u) \cap \lambda^D(u) = \phi$.

The quadruple (D, δ, σ, d) is called a **data structure schedule** *for D, where $d \in \{R, W\}$ specifies a* direction*: read ("R"), or* write *("W"). If d="R", then the schedule is called a read or gather schedule, otherwise a write or scatter schedule.* □

A schedule (D, δ, σ, d) can describe the non-local accesses made in a thread to elements of D in a given section of code, and thus can be directly used to fully specify the required communication.

6.7 Synchronous Method Activations

Synchronous method activations are initiated by call statements. They behave like (local or remote) procedure calls: the caller is blocked until execution terminates. Such activations do not have an identification accessible to the user, thus they cannot be managed or subject to synchronization in the way of asynchronous threads.

Example 6 *This example provides a simple solution for a matrix transposition (Fig. 3). The matrix, $A(N, N)$, is dynamically allocated in the PIM memory, depending on a runtime-determined size specification, and distributed by block in both dimensions. The blocksize is chosen in such a way that each block fits completely into one ASAP (8 by 8 elements). Here we make the simplifying assumption that 8 divides the number of elements in each dimension of the matrix. Each block can be uniquely identified by a pair of indices (II, JJ), where II and JJ range from 1 to $N/8$.*

The algorithm consists of two steps [40]: In Step 1, whole blocks are moved to their transposed location, without changing the order of their elements. Note that blocks in the main diagonal need not be moved in this step. The subsequent Step 2, applied to each individual block, transposes the elements inside this block. The specific choice of the blocksize makes it possible to perform the intra-block transpose in a highly efficient way within one ASAP. The keyword **ASAP** *in the corresponding method definition provides a corresponding hint to the compiler.*

The algorithm spawns a separate thread, $\mathbf{t}(II, JJ)$, for each block (II, JJ). The threads $\mathbf{t}(II, JJ)$ with $II \leq JJ$ are spawned at the beginning. (1) If $II < JJ$, then $\mathbf{t}(II, JJ)$ exchanges blocks (II, JJ) and (JJ, II), and subsequently spawns the thread $\mathbf{t}(JJ, II)$. It then proceeds to perform Step 2, its local transpose. (2) If $II \geq JJ$, then $\mathbf{t}(II, JJ)$ can immediately proceed to Step 2.

A further remark is necessary here. We have modeled the transpose at a low level of abstraction to illustrate some of the features provided by our model in an easy-to-understand environment. A sufficiently sophisticated compiler/runtime system could conceivably produce the same code automatically, based on the single Fortran 95 statement **FORALL** *(I=1:N,J=1:N) A(I,J)=A(J,I).* □

7 Synchronization

The threads of a macroserver execute asynchronously in parallel, sharing access to all its resources. Moreover, they may access other macroservers indirectly via method activations, subject to the constraints specified in the acquaintance relation. In certain situations, threads must be synchronized in order to preserve the integrity of data and maintain inherent resource invariants.

Our model provides conventional support for mutual exclusion and condition synchronization via atomic methods and condition variables [5]. *Mutual exclusion*, discussed in Section 7.1, guarantees atomic access of threads to a given resource, leaving the order in which simultaneously arriving threads are serviced

```
PROGRAM MATRIX_TRANSPOSE

MACROSERVER CLASS t_class
  INTEGER :: N
  FUTURE, ALLOCATABLE :: F(,)
  REAL, ALLOCATABLE, DISTRIBUTE(BLOCK(8),BLOCK(8)) :: A(,)

 CONTAINS

  METHOD initialize()
    READ(N)
    ALLOCATE(A(N,N),F(N/8,N/8))
  END initialize

  METHOD block_exchange(II,JJ)
    ! exchanges block (II,JJ) with block (JJ,II)
  END block_exchange

  ASAP METHOD intra_block_transpose(II,JJ)
    INTEGER :: II,JJ
    INTEGER :: I, J
    FORALL(I=II:II+7,J=JJ:JJ+7) A(I,J) = A(J,I)
  END intra_block_transpose

  METHOD transpose(II,JJ)
    INTEGER :: II,JJ
    IF II < JJ
        THEN block_exchange(II,JJ)
             F(JJ,II)= SPAWN(my_transpose%transpose,JJ,II, ON HOME(A(JJ,II)))
    END IF
    CALL intra_block_transpose(II,JJ)
  END METHOD transpose

  METHOD transpose_matrix()
    INTEGER :: II,JJ
    FORALL THREADS(II=1:N-7:8, JJ=1:N-7:8, II ≤ JJ, ON HOME(A(II,JJ)))
              F(II,JJ)= SPAWN(my_transpose%transpose,II,JJ)
    WAIT(ALL(F))        ! global barrier
  END METHOD transpose_all

END MACROSERVER CLASS t_class

 ! Main Program
MACROSERVER(t_class) my_transpose = CREATE(t_class)
CALL my_transpose%initialize()
CALL my_transpose%transpose_matrix()
 ...
END PROGRAM MATRIX_TRANSPOSE
```

Figure 3: Matrix Transpose

implementation-dependent. Mutual exclusion can be expressed using methods declared as atomic (Section 4.2).

Condition synchronization (Section 7.2) may be formulated using condition variables associated with programmed synchronization conditions. Threads whose synchronization condition at a given point is not satisfied can suspend themselves with respect to the associated condition variable, waiting for other threads to change state in such a way that the condition becomes true.

An additional synchronization mechanism is based on future variables, allowing implicit and explicit synchronization tied to the execution status of a thread. This will be discussed in Section 7.3.

7.1 Mutual Exclusion

Methods of a macroserver, **S**, may have the attribute **atomic** (Section 4.2). At any time, only one atomic method of **S** may execute. Such a method may activate another atomic method of the same macroserver without being blocked; furthermore, non-atomic methods of **S**, or atomic or non-atomic methods of any other macroserver may proceed in parallel with the execution of an atomic method in **S** [36].

A macroserver class may be declared as a *monitor class*. In this case, all methods belonging to that class, and all transient methods imported into a macroserver created for that class are implicitly atomic. In a monitor macroserver, each method execution has exclusive access to all variables of the macroserver.

7.2 Condition Synchronization

Condition synchronization can be expressed using *condition variables* and a set of associated operations for blocking and releasing threads. Condition variables are connected to *synchronization conditions* which are predicates in the state space of a macroserver. This association is implicit, its management being in the responsibility of the programmer.[8] All operations applied to condition variables must be placed in atomic methods.

7.2.1 Condition Variables

Condition variables are declared with the type denotation **condition**. Assume that c is a condition variable existing in a macroserver. At any time, c is associated with a set, Q^c, of threads **blocked** with respect to c.

Q^c is initially empty. Insertion and removal of threads with respect to Q^c are controlled by the operations described below. Apart from these operations, the programmer cannot manipulate Q^c; however, the status of the set can be polled using inquiry functions. Usually, Q^c is organized as a FIFO queue, guaranteeing fairness in the management of threads accessing c. However, priority considerations may lead to other scheduling strategies [5].

7.2.2 Synchronization Operations

In this section, we briefly describe a collection of operations and functions that can be applied to condition variables. We focus only on the basic functionality required, with no claim for completeness. In particular, additional inquiry functions could be introduced to check for properties of Q^c and the threads contained in it.

For the following, assume that the operation to be discussed occurs in a thread **t** executing an atomic method of macroserver **S**, and that c is a condition variable. Furthermore, let $mutex^{\mathbf{S}}$ denote a semaphore used for controlling access to atomic functions in **S**.

Wait: The execution of **WAIT** (c) results in the following actions:

1. Set the *blocking_status* of **t** to *blocked*.
2. $Q^c := Q^c \cup \{\mathbf{t}\}$

[8] This is the major difference to Hoare's *conditional critical regions* [28], which provide a more elegant formulation at the expense of an overhead which is difficult to control.

3. Release $mutex^{\mathbf{S}}$

Signal: The execution of **SIGNAL** (c) has no effect, if Q^c is empty. Otherwise,

1. A thread, say **t'**, is selected from Q^c.
2. $Q^c := Q^c - \{\mathbf{t'}\}$
3. The *blocking_status* of **t'** is set to *active*.

Note that **t** does not release $mutex^{\mathbf{S}}$, i.e., the signal operation has a *signal-and-continue* semantics [5].

Signal All: **SIGNAL_ALL** (c) has no effect if Q^c is empty; otherwise it applies Steps 2 and 3 in the signal operation to all threads in Q^c.

Empty: **EMPTY** (c) yields the (logical) value of the expression ($Q^c = \phi$) and has no other effect.

7.3 Future-Based Synchronization

Our model proposes special syntactic support for synchronization based on future variables. This takes two forms, explicit and implicit.

Explicit synchronization can be formulated via a version of the wait statement that can be applied to a logical expression depending on futures. Wait can also be used in the context of a forall threads statement, as shown in the example of Figure 3.

Implicit synchronization is automatically provided if a typed future variable occurs in an expression context that requires a value of that type.

7.3.1 Explicit Synchronization

Assume that thread **t** in macroserver **S** is executing. Explicit synchronization with respect to futures can be expressed in the form

WAIT (*future_exp*)

where *future_exp* is a side-effect free expression yielding a scalar logical value. It may contain the following types of primaries:

- read-only variables of **S**
- private variables and formal parameters of **t**
- future variables
- pure synchronous method calls

If, at the time the above statement is executed, the evaluation of *future_exp* yields **true**, thread **t** proceeds with its execution. Otherwise, the *blocking_status* of **t** is set to *blocked* and execution of **t** is suspended.

The termination of threads results in a re-evaluation of future expressions associated with blocked threads, and their potential release.

7.3.2 Implicit Synchronization

Assume that f is a future variable which is declared with type T. Then, any thread to which f can be bound must yield a value of type T at the time of its termination. Based on the future concept of Multilisp [26], we allow f to occur as a term of type T in "regular" expressions of the language. The evaluation of such an expression will be blocked up to the time the thread bound to f terminates.

7.4 The Producer/Consumer Problem

Based on the skeleton provided in Fig.1, we complete a specification of the producer/consumer problem using macroservers.

The consumer/producer problem represents a class of coordination problems where a set of cyclic producer threads compute some values that are processed by a set of cyclic consumer threads in the order in which they are produced. Producers and consumers are coordinated using a bounded buffer into which all producers write and from which all consumers read. The buffer is managed as a cyclic memory using a FIFO strategy.

More specifically, we formulate the problem as follows: Consider a bounded buffer as specified by the macroserver class *buffer_template* of Fig. 1. The buffer is organized as a FIFO queue; writing into and reading from the buffer is performed by atomic methods *put* and *get*, respectively. In the initial state, all elements of the buffer are "empty". When a producer thread finishes a computing cycle, it calls the *put* method, which deposits the new data item into the next empty slot of the buffer and transforms the state of that slot to "full". Similarly, when a consumer thread is ready to start its next cycle, it reads a data item from the next full slot, changes the state of the slot to "empty", and processes the data item.

The problem is that at the time *put* is called there may be no empty slot – i.e., the buffer may be full; similarly, at the time *get* is called, there may be no full slot – i.e., the buffer may be empty. The algorithm uses the condition variables *c_notfull* and *c_notempty* to respectively represent the synchronization conditions "the buffer is not full" and "the buffer is not empty". A producer may write into the buffer only if the buffer is not full, and a consumer may read from the buffer only if it is not empty. If a synchronization condition is not satisfied, the executing thread is blocked.

The number of producer threads, *np*, and consumer threads, *nc*, is determined at runtime. Each of the *np* producer threads is created by spawning the method *produce* in the macroserver *producer_template*. Each consumer thread is created by spawning the method *consume* in the macroserver *consumer_template*.

Since producer and consumer threads are non-terminating cyclic threads which do not need any explicit control they are not bound to future variables. All these threads are detached.

A solution to the problem is given in Figures 4 and 5.

8 Related Work

The macroserver model as applied to distributed arrays of PIM chips is unique and contributes a new dimension and opportunity to parallel system design and operation. However, many of the individual ideas have precedence in previous work performed in different contexts. Some of those are briefly mentioned here.

The J-Machine project [17], conducted at MIT with important collaboration at Caltech, considered a system comprising an array of systems on a chip and employed an object-based model for governing their global behavior. While the technology was inadequate at the time and the project premature, the use of a global distributed name space, message driven computation, and hardware support for method execution, and on-chip multithreading were all implemented to some degree as part of this visionary project.

An important application of the macroserver model executing on PIMs is the work at the University of Delaware on "percolation" which is a powerful new methodology for latency management on large distributed systems. Percolation provides the means for proactive prestaging of computations to be performed. This is done by the PIMs, thus avoiding the overhead and hiding the latency of the system compute processors scheduled to do the work. Percolation will be implemented as a macroserver(s), monitoring the state of readiness for tasks, and migrating the tasks to high speed memory for the compute processors.

A significant contribution in this area has been made by University of Notre Dame where the original work on microservers was conducted for PIMs as well as the design of some of the most advanced PIMs yet conceived. Work detailing the low level system software for PIM nodes and details of the instruction set for the row wide ALU of the PIM processor have been conducted and simulated there [10].

The macroserver model has some features in common with the actor model of computation [1]. Similar to macroservers, actors encapsulate data and procedures (methods), use asynchronous messages to trigger computations, and perform communication in a location-transparent way. Recent versions of the model, in

```
PROGRAM CONSUMER-PRODUCER

MACROSERVER CLASS  buffer_template(size)

   INTEGER :: size
   REAL       :: fifo(0:size-1)                  ! buffer data structure
   INTEGER :: count = 0                          ! number of full elements in buffer
   INTEGER :: px=0, cx=0                         ! producer index, consumer index
   CONDITION :: c_notfull, c_notempty

 CONTAINS
  ATOMIC METHOD put(x)
     REAL :: x
     DO WHILE (count == size) WAIT(c_notfull)    ! wait until there is an empty element in the buffer
     fifo(px) = x                                ! Put x into first empty buffer element
     px = MOD(px+1,size)
     count = count + 1
     SIGNAL(c_full)
     END
  END put

  ATOMIC REAL METHOD get()
     DO WHILE (count == 0) WAIT(c_notempty)      ! wait until there is a full element in the buffer
     get = fifo(cx)                              ! Read next full buffer element
     cx = MOD(cx+1,size)
     count = count - 1
     SIGNAL(c_empty)
    END
   END get

    ...
END MACROSERVER CLASS buffer_template
```

Figure 4: Producer/Consumer Problem: Part 1

```
MACROSERVER CLASS producer_template
   ! Data declarations for all producer threads
 CONTAINS          ! producer methods:
    ...
METHOD produce(b)
   MACROSERVER(buffer_template) b    ! b is a reference to a macroserver based on the class buffer_template
   REAL A       ! private variable for a specific producer thread
   ...
   DO WHILE (TRUE)          ! forever
     ! compute a data item and assign to A
     CALL b%put(A)
   END DO
END produce
END MACROSERVER CLASS producer_template

MACROSERVER CLASS consumer_template
   ! Data declarations for all consumer threads
 CONTAINS  ! consumer methods:
   ...
METHOD consume(b)
   MACROSERVER(buffer_template) b
   REAL A       ! private variable for a specific consumer thread
   ...
   DO WHILE (TRUE)          ! forever
    A = b%get()
     ! process the data item in A
   END DO
END consume
END MACROSERVER CLASS consumer_template

 ! Main program:
INTEGER   np, nc, buffersize
MACROSERVER (buffer_template) my_buffer = CREATE(buffer_template, buffersize)
MACROSERVER(producer_template) my_producer = CREATE(producer_template,my_producer)
MACROSERVER(consumer_template) my_consumer = CREATE(consumer_template,my_consumer)

READ(np,nc,buffersize)

 ! Create np producer threads in the macroserver my_producer, passing my_buffer as an argument to each thread:
FORALL THREADS (I=1:np) SPAWN(my_producer, produce, my_buffer) DETACHED

 ! Create nc consumer threads:
FORALL THREADS (I=1:nc) SPAWN(my_consumer, consumer, my_buffer) DETACHED

...
END PROGRAM
```

Figure 5: Producer/Consumer Problem: Part 2

particular the THAL language [37], include support for dynamic actor placement. However, the actor model is simpler and at a lower level of abstraction than the macroserver model. State change in actors is always atomic, resulting in a serialization of message reception. The asynchronous execution and synchronization of threads sharing the state within an actor is not possible. Furthermore, actors do not provide features equivalent to the data and work distribution concepts in macroservers.

The thread model of macroservers was influenced by the object-oriented or object-based languages Concurrent C++ [36], pC++ [58], and Opus [15], as well as by the Pthreads standard [45]. The macroserver approach to synchronization essentially combines monitor-like features as proposed by Hoare [29] with MULtilisp's futures [26].

Other models providing programming support for distributed shared data include Linda [2], Agora [9], and Orca [6], a predecessor of Opus. Object-based operating systems include Choices [12] and COSMOS, an operating system for the J-machine.

Finally, the concepts of data distribution and alignment that form an important part of the macroserver model are generalizations of earlier work, in particular in Kali [42], Vienna Fortran [59, 13], HPF [31], and HPF+ [14]. The formalization of work distributions and schedules is based on [52, 51, 7].

9 Discussion

The challenge of defining a new model of execution, at any level in the system application, is that the criteria of success are qualitative and incomplete at best. Only as the complex design space is investigated are the dominant issues and metrics exposed and understood. While consistency and completeness can be established within the framework of the model itself, effectiveness in the larger context must be driven by the factors constraining the elicited behavior. These include the language and compiler interface from "above" and runtime interface determined by the hardware mechanisms and low level node microserver functions "below". Developing an appreciation for the implications of these interfaces is essential for ultimately judging the value of the proposed model. When considering alternative semantics for such constructs as conditional flow control, a number of choices are evident. A final decision has to be made based on the impact on performance, generality, and cost as well as other factors.

An important set of tasks are required to carry this work forward. The first is a prototype emulator that will provide the means to exercise the model. This will provide experience in translating application kernels to the macroserver model and expose errors or gaps in the model. The second task is to perform an implementation study to expose how the elements of the model will be mapped to the PIM array hardware/software system. Some measure of cost, both time and space, can then be attributed to the primitive base functions permitting a measure of effectiveness. While any such results will be sensitive to even small changes in the actual design details of the PIM chips to be employed, the major quantitative trends will be exposed and available to analysis. These results will also have implications for the development of the PIM architecture as well.

In the remainder of this section, we briefly comment on a number of design decisions, possible alternatives, and future paths of research.

9.1 Data and Work Distributions

Our approach introduces data distributions and alignments, work distributions, and communication schedules as first-class objects which can be dynamically bound to data structures or systems of threads.

This is a generalization of features found in a number of existing data parallel languages. They provide close control over the location and movement of data and work, allowing fine tuning of a parallel algorithm's behavior at a low level of abstraction. The amount of detail involved in such control suggests that most of these features are not explicitly provided at an application programming interface but are only made available through higher-level language layers. The specification of such layers is a research issue.

A possible direction for future research could target an extension of our distribution/alignment concept to address issues related to the scheduling of threads and data in deep memory hierarchies such as in the HTMT architecture (see also Section 9.4).

9.2 Methods, Threads, and Synchronization

Some features in the design space covering methods, threads, and synchronization could have been defined in a different way. We provide a short overview of alternatives:

- **Method results**
  In our current model, a thread can return only a single value. While this is not a restriction in principle, since the value returned may be a reference to an arbitrarily complex data structure, an approach which also allows the return of results via output parameters of a method [18, 15] is sometimes easier to use. We chose the simpler solution in order to avoid complicating the semantics of futures.

- **Method guards**
  Method guards as for example in Opus [15] provide an elegant mechanism for the specification of enabling conditions for method calls. However, since macroservers allow internal thread parallelism, they require a mechanism for synchronization inside methods, which can also be used to express the synchronization related to guards. Furthermore, guards imply busy waiting which is seen as an undesirable feature at the low level of abstraction the model is dealing with.

- **Condition Synchronization**
  We have chosen a conservative, low-level approach to condition synchronization, based on condition variables and direct control of blocking and releasing threads with respect to a synchronization condition.

  A more flexible and higher-level synchronization mechanism could be defined along the lines of Hoare's *conditional critical regions* [28]. Such an approach would also allow the unification of condition synchronization with future-based synchronization. However, the feasibility of an efficient implementation of such a mechanism is unclear at this time. We believe that this topic needs additional research before a definite decision can be made.

9.3 Languages, Compilation and Runtime Technology

An important direction for future work will be the specification of a higher-level application programming interface that can be included into either an existing or a new high-level language, and mapped to the macroserver model. This will also require significant new compiler and runtime technology. While many of the ideas developed for the compilation and runtime system optimization of data parallel languages in the past decade [61, 22, 8] will be useful for dealing with a subset of the problem, the much larger design space associated with the macroserver model will necessitate the development of new techniques. We expect that an important line of research will be based on feedback-directed and dynamic compilation technology, as exemplified in the work on flow sensitive profiling [3], qualified flow analysis [4], and the Java hotspot compiler [34]. Another direction of work will deal with optimization techniques targeting the row wide ASAP instruction set.

9.4 Macroservers in the Context of HTMT

In this section, we outline a role for the proposed macroserver in the context of the HTMT project.

The *Hybrid Technology Multi-Threaded* (HTMT) architecture exploits radically new technology that steps outside the evolution path of the computer systems currently under design of projected to the near-future [55]. It features the adaption of a moderate number of processors built with very fast device technology (up to 100 GHz) and a deep multi-level PIM memory hierarchy. The current architecture design for HTMT features 100 GHz super-conducting processors and a memory hierarchy with at least 4 levels: CRAM (Cryogenic RAM), SRAM, DRAM, and HRAM. The latency to access memory across memory levels will vary by 4-5 orders of magnitude or more. PIM technology is used in both the SRAM and DRAM levels [39].

It is anticipated that an enormous amount of parallelism will be available at all levels of the machine, and that balancing the tradeoff between latency and bandwidth in the memory hierarchy and interconnection networks will be essential for the success of the architecture. The applications enabled by such high-end machines are also expected to be significantly more complex and dynamic than the applications in the

past. A key question is: Should the complexity of these new high-end machines be explicitly exposed to programmers? And if so, where and how? We anticipate that a programming and program execution model that eliminates some of the unnecessary boundaries between the different components and layers, should be integrated in the software environment.

A key feature of the proposed HTMT execution model is its ability to explicitly expose the complex memory hierarchy, including the performance cost of data and computation movement within the hierarchy. The costs of migrating data/computation through the memory hierarchy are expressed in the *percolation* model that can facilitate latency/bandwidth management to achieve desirable performance. At a first glance, percolation appears to be a combination of multi-threading with dynamic prefetching of coarse-grain contexts. However, percolation is different from traditional prefetching in the following ways. In the past, prefetching concentrated mostly on moving blocks of contiguous *data* within the memory hierarchy. The thread percolation, however manages the movement of contexts: including data, program instructions, and control state. Furthermore, percolation may also involve data gathering/scattering as well as data (layout) reorganization within the memory hierarchy.

The HTMT percolation model will be realized through a massive array of MIND/PIM in the DRAM-PIM and SRAM-PIM region of the HTMT architecture. The proposed macroserver provides a high-level object-based programming model that bridges the gap between the low-level *microserver* for PIM architecture and the system-oriented functions that PIM users (compilers, runtime system software, etc.) need to utilize. System-wide functions required by the HTMT runtime and other system software (e.g. compilers for the proposed HTMT Threaded-C language [20]) include the management of virtual to physical address translations, the exploitation of the vast amount of parallelism available at each memory hierarchy level, the effective management of computations with variable granularity and disparate priorities, and the implementation of functions based on a unified model of computation that governs operations on the massive array of PIMs. Typical functions required in the percolation model, such as data gathering/scattering, layout reorganization, and movement of data up and down the memory hierarchy, will be made available by macroservers and its extensions.

10 Conclusion

The macroserver intermediate computing model has been devised in its initial form to facilitate the investigation of dynamic execution on arrays of future generation PIM chips. A consistent logical structure that favors parallelism and dynamic resource management has been presented with a number of examples demonstrating how distributed computation would be organized and performed according to these principals. The findings of this initial inquiry are strongly encouraging and support both the need for such an intermediate model and the value of an object-based approach in realizing it.

Acknowledgements

The authors thank Jay Brockman and Peter Kogge (University of Notre Dame), Guang Gao and Jose Amaral (University of Delaware) for many fruitful discussions about the topic of this report. Guang Gao and Jose Amaral contributed to Section 9.4.

References

[1] G.A.Agha. ACTORS: A Model of Concurrent Computation in Distributed Systems. *MIT Press*, 1986.

[2] S.Ahuja, N.Carriero, and D.Gelernter. Linda and Friends. *IEEE Computer*, 19, pp.26-34, August 1986.

[3] G.Ammons, T.Ball, and J.R.Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. *Proc.SIGPLAN'97 Conf.on Programming Language Design and Implementation (PLDI)*, 85-96, Las Vegas,Nevada, June 1997.

[4] G.Ammons and J.R.Larus. Improving Data-Flow Analysis with Path Profiles. *Proc.SIGPLAN'98 Conf.on Programming Language Design and Implementation (PLDI)*, 72-84, Montreal, Canada, June 1998.

[5] G.R.Andrews. Concurrent Programming. Principles and Practice. *Benjamin/Cummings*, 1991.

[6] H.E.Bal, M.F.Kaashoek, and A.S.Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3), pp.190-205, March 1992.

[7] S.Benkner,P.Mehrotra,J.Van Rosendale, and H.P.Zima. High-Level Management of Communication Schedules in HPF-like Languages. *Proc.International Conference on Supercomputing 1998 (ICS'98)*, Melbourne, Australia, July 1998.

[8] S.Benkner and H.P.Zima. Compiling High Performance Fortran for Distributed-Memory Architectures. In: Trystram,D.(Ed.): *Parallel Computing, Special Anniversary Issue* (2000, to appear).

[9] R.Bisiani and A.Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Transactions on Computers*, 37(8), pp.930-945, August 1988.

[10] J.B.Brockman,P.M.Kogge,V.W.Freeh,S.K.Kuntz, and T.L.Sterling. Microservers: A New Memory Semantics for Massively Parallel Computing. *Proceedings ACM International Conference on Supercomputing (ICS'99)*, June 1999.

[11] D.Callahan and B.Smith. A Future-Based Parallel Language for a General-Purpose Highly-Parallel Computer. *Technical Report, Tera Computer Company*, 1990.

[12] R.H.Campbell and N. Islam. Choices: A parallel object-oriented operating system. In G.A.Agha, P.Wegner, and A.Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pp. 393–451, MIT Press, 1993.

[13] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[14] B. Chapman, P. Mehrotra, and H. Zima. Extending HPF for Advanced Data Parallel Applications. *IEEE Parallel and Distributed Technology*, Fall 1994, pp. 59-70.

[15] B. Chapman, M. Haines, P. Mehrotra, J. Van Rosendale, and H. Zima. Opus: A Coordination Language for Multidisciplinary Applications. *Scientific Programming*, 1998.

[16] B.Chapman,P.Mehrotra, and H.Zima. Enhancing OpenMP with Features for Locality Control. *Proc. ECMWF Workshop "Towards Teracomputing – The Use of Parallel Processors in Meteorology"*. Reading, England, November 1998.

[17] W.J.Dally et al. The Message Driven Processor: A Multicomputer Processing Node With Efficient Mechanisms. *IEEE Micro*, April 1992,pp.23-28.

[18] J.C.Adams, W.S.Brainerd, J.T.Martin, B.T.Smith, and J.L.Wagener. Fortran 95 Handbook. Complete ISO/ANSI Reference. *The MIT Press*, 1997.

[19] G.Fox,S.Hiranandani,K.Kennedy,C.Koelbel,U.Kremer,C.Tseng, and M. Wu. Fortran D language specification. *Department of Computer Science Rice COMP TR90079*, Rice University, March 1991.

[20] G.R.Gao,J.N.Amaral,A.Marquez,K.Theobald,S.Ryan,Z.Ruiz,T.Geiger, and C.J.Morrone. HTMT Phase 2 Report. *CAPSL Technical Memo TM31*, University of Delaware, July 1999.

[21] M.Gokhale,W.Holmes,and K.Iobst. Processing in Memory: The Terasys Massively Parallel PIM Array. *IEEE Computer 28(4)*,pp.23-31,1995.

[22] M.Gupta,E.Schonberg, and H.Srinivasan. A Unified Framework for Optimizing Communication in Data-Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems* Vol.7(7), pp.689-704, July 1996.

[23] M.Haines,D.Cronk,and P.Mehrotra. On the Design of Chant: A Talking Threads Package. *Proc.Supercomputing 94*,pp.350-359, Washington,D.C., November 1994.

[24] M.Haines,P.Mehrotra,and D.Cronk. Ropes: Support for Collective Operations Among Distributed Threads. *Scientific Programming*, 1995. Also: *ICASE Report* 95-36,NASA Langley Research Center, Hampton, Virginia, May 1995.

[25] M.Hall,J.Koller,P.Diniz,J.Chame,J.Draper, J.LaCoss, J.Granacki, J.Brockman, A.Srivastava, W.Athas, V.Freeh, J.Shin, and J.Park. Mapping Irregular Applications to DIVA, a PIM-Based Data Intensive Architecture. *Proceedings SC'99*, November 1999.

[26] R.H.Halstead,Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems (TOPLAS), 7(4),501-538*, October 1985.

[27] C.Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence 8*, pp.323-364, 1977.

[28] C.A.R.Hoare. Towards a Theory of Parallel Programming. In:(C.A.R.Hoare and R.H.Perrott,Eds.) *Operating System Techniques*, Academic Press, pp.61-71, 1972.

[29] C.A.R.Hoare. Monitors: An Operating Systems Structuring Concept. *Comm.ACM* 17(10),pp.549-557,1974.

[30] W.Horwat, B.Totty, and W. J. Dally. Cosmos: An operating system for a fine-grain concurrent computer. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pp. 451–77. MIT Press, 1993.

[31] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0*, January 1997.

[32] L.H.Holley and B.K.Rosen. Qualified Data-Flow Problems. *IEEE Transactions on Software Engineering*,SE-7(1):60-78, January 1981.

[33] *http://www.ibm.com/news/1999/12/06.phtml*

[34] *http://www.javasoft.com/features/1999/04/hotspot.html*

[35] L.V.Kale and S.Krishnan. CHARM++. In: G.V.Wilson and P.Lu (Eds.): *Parallel Programming Using CC++*, Chapter 5, pp.175-213, The MIT Press, 1996.

[36] C.Kesselman. CC++. In: G.V.Wilson and P.Lu (Eds.): *Parallel Programming Using CC++*, Chapter 3, pp.91–130, The MIT Press, 1996.

[37] W.Kim. Thal: An Actor System for Efficient and Scalable Concurrent Computing. *Ph.D.Thesis, University of Illinois at Urbana-Champaign*, 1997.

[38] P.M.Kogge. The EXECUBE Approach to Massively Parallel Processing. In: *Proc.1994 Conference on Parallel Processing*, Chicago, August 1994.

[39] P.M.Kogge,J.B.Brockman,T.L.Sterling and G.R.Gao. Processing-in-Memory: Chips to Petaflops. *Proc. International Symposium on Computer Architecture*, Denver, Colorado, June 1997.

[40] V.Kumar,A.Grama,A.Gupta,and G.Karypis. Introduction to Parallel Computing. Design and Analysis of Algorithms. The Benjamin/Cummings Publishing Company,1994.

[41] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Proc. of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pp. 222–33, Cambridge,MA, October 1996.

[42] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 364–384. Pitman/MIT-Press, 1991.

[43] P.Mehrotra,J.Van Rosendale, and H.P. Zima. High Performance Fortran: History, Status and Future. In: Zapata,E. and Padua,D.(Eds.): *Parallel Computing, Special Issue on Languages and Compilers for Parallel Computers*, Vol.24, No.3-4,pp.325–354 (1998)

[44] P.Mehrotra,J.Van Rosendale, and H.P. Zima. Language Support for Multidisciplinary Applications. *IEEE Computational Science and Engineering* Vol.5,No.2,pp.64-75 (April-June 1998).

[45] B.Nichols,D.Buttlar,and J.Proulx Farrell. Pthreads Programming. *O'Reilly*,1998.

[46] Notre Dame University. Final Report: PIM Architecture Design and Supporting Trade Studies for the HTMT Project. *PIM Development Group, HTMT Project*, University of Notre Dame, September 1999.

[47] OpenMP Consortium. OpenMP Fortran Application Program Interface, Version 1.1. *http://www.openmp.org*, November 1999.

[48] J.K.Ousterhout. Scheduling Techniques for Concurrent Systems. *Proc.Distributed Systems Computing Conference*,pp.22-30,1982.

[49] R.Panwar and G.Agha. A Methodology for Programming Scalable Architectures. *Journal of Parallel and Distributed Computing*, 22(3),pp.479-487, September 1994.

[50] D.Patterson et al. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, April 1997.

[51] R. Ponnusamy, J. Saltz, A. Choudhary. Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse. *Technical Report, UMIACS-TR-93-32*, University of Maryland, April 1993.

[52] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.

[53] M.Snir, S.W.Otto, S.Huss-Lederman, D.W.Walker, and J.Dongarra. MPI – The Complete Reference. *The MIT Press*, Cambridge, Massachusetts, 1997.

[54] T.Sterling and P.Kogge. An Advanced PIM Architecture for Spaceborne Computing. *Proc.IEEE Aerospace Conference*, March 1998.

[55] T.Sterling and L.Bergman. A Design Analysis of a Hybrid Technology Multithreaded Architecture for Petaflops Scale Computation. *Proceedings ACM International Conference on Supercomputing (ICS'99)*, June 1999.

[56] V.Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4), pp.315-339, december 1990.

[57] M.Ujaldon,E.L.Zapata,B.M.Chapman,and H.P.Zima. Vienna Fortran/HPF Extensions for Sparse and Irregular Problems and Their Compilation. *IEEE Transactions on Parallel and Distributed Systems*, Vol.8, No.10, pp.1068-1083 (October 1997).

[58] S.X.Yang,D.Gannon,P.Beckman,J.Gotwals, and N.Sundaresan. pC++. In: G.V.Wilson and P.Lu (Eds.): *Parallel Programming Using CC++*, Chapter 13, pp.507–545, The MIT Press, 1996.

[59] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. *Internal Report 21, ICASE, Hampton, VA*, March 1992.

[60] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series, Addison-Wesley, 1990.

[61] H. Zima and B. Chapman. Compiling for Distributed Memory Systems. *Proceedings of the IEEE, Special Section on Languages and Compilers for Parallel Machines*, pp. 264-287, February 1993.

Appendix

A Examples

A.1 The Readers/Writers Problem

We deal with the following variant of the readers/writers coordination problem: a given data resource can be respectively read or written by a set of cyclic reader and writer threads. Writing is exclusive, while all readers can work in parallel. As a consequence, synchronization must enforce for each writer exclusive access to the resource.

We do not go here into details of the data resource and ignore the details of reading and writing: the set of reader and writer threads (whose number can be determined at runtime) can be spawned in a similar way as shown in the producer/consumer example in Section 7.4.

We use a monitor class to express the scheduling of the resource. The following terminology is adopted:

- If, at a given time, a reader or writer thread actually accesses the data resource, we say that the thread is *working*. The numbers of working readers and writers are respectively denoted by wr and ww. Note that ww may only assume the values 0 or 1.
- The reader and writer threads that are blocked as a result of an unsuccessful attempt to access the resource are called *registered*. Their numbers are respectively denoted by rr and rw.
- Two synchronization conditions, R and W, are used to control access for readers and writers, respectively:
  - R is the synchronization condition for the readers: $ww = 0$ **and** $rw = 0$.
  - W is the synchronization condition for the writers: $wr = 0$ **and** $ww = 0$.

  R and W are represented by the condition variables c_R and c_W, respectively.

A solution to the problem is given in Figure A.1. The creation of the macroserver for class *scheduler_template* is suppressed in the algorithm; reader and writer threads receive a reference to that macroserver as an argument.

A.2 Fine-Grain Scheduling of Resources

Assume a problem similar to that discussed in Section A.1, except that the data resource is structured and the access to each individual element has to be scheduled independently. Such fine-grain scheduling must be employed if the data resource is large and blocking of the whole resource is inacceptable when accessing a single element, as for example in a flight reservation system.

This scheduling problem can be handled in our framework by creating a separate instance of the scheduler class, *scheduler_template*, for each element to be protected, and including a corresponding macroserver variable into the data structure. This approach is outlined in Fig.7, using the monitor class introduced in Fig.A.1.

Note that in this example we align each element of the resource with the data used by its scheduler, an approach which fully exploits the locality of processing in a PIM array.

A.3 Sparse Matrix Vector Product

The Conjugate Gradient (CG) algorithm is a powerful iterative method for solving large sparse systems of linear equations [40]. A key element in the CG algorithm is a sparse matrix vector multiplication. In this section, we outline an approach for the parallelization of a sparse matrix vector product on a PIM array, based partly on concepts developed in [57].

```
MACROSERVER CLASS MONITOR scheduler_template   ! all methods are atomic
  INTEGER wr=0, ww=0
  CONDITION c_R, c_W
 CONTAINS
  METHOD begin_read()
   DO WHILE ((ww > 0) OR (NOT EMPTY(c_W))) WAIT c_R
   wr = wr + 1
  END begin_read

  METHOD end_read()
   wr = wr-1
   IF wr == 0 SIGNAL(c_W)
  END end_read

  METHOD begin_write()
   DO WHILE ((wr > 0) OR (ww > 0)) WAIT c_W
   wr = 1
  END begin_write

  METHOD end_write()
   ww = 0
   IF (NOT EMPTY(c_R))
     THEN SIGNAL_ALL(c_R)
     ELSE SIGNAL(c_W)
   END IF
  END end_write

END MACROSERVER CLASS scheduler_template

! core of reader method:
 METHOD reader(s)
  MACROSERVER(scheduler_template) s
    ...
  DO WHILE(TRUE)   ! forever
  ...
  CALL s%begin_read
      ! access the resource
  CALL s%end_read
  END WHILE
  ...
 END reader

! core of writer method:
 METHOD writer(s)
  MACROSERVER(scheduler_template) s
    ...
  DO WHILE(TRUE)   ! forever
  ...
  CALL s%begin_write
      ! access the resource
  CALL s%end_write
  END WHILE
  ...
 END writer
```

Figure 6: Readers/Writers Problem

```
MACROSERVER CLASS MONITOR scheduler_template

   ...
  CONTAINS
   METHOD begin_read()
     ...
   METHOD end_read()
     ...
   METHOD begin_write()
     ...
   METHOD end_write()
     ...

END MACROSERVER CLASS scheduler_template

 ! Main program
INTEGER :: N, I, K
TYPE flight_record          ! sketch of data structure for a flight record
  INTEGER :: date, time, flight_number, ...
  INTEGER :: create_status
   ...
  MACROSERVER (scheduler_template) my_scheduler    ! reference to the "private" scheduler for this element
   ...
END TYPE flight_record
   ...
TYPE (flight_record), ALLOCATABLE :: flights(:)
   ...
 ! determine N and a data distribution for flights, stored in dv.
ALLOCATE (flights(N))
DISTRIBUTE (flights,dv)
   ...
DO I=1,N
  flights(I)%my_scheduler = CREATE (scheduler_template, ON HOME (flights(I)), flights(I)%create_status)
    ...
END DO
...
 ! write access to element flight(K) in a thread:
K=...
CALL flights(K)%my_scheduler%begin_write()
     ! write
CALL flights(K)%my_scheduler%end_write()
...
```

Figure 7: Fine-grain scheduling

```
! The vectors D, C, and R represent the sparse matrix A(1:N,1:M) in CRS representation:
  REAL :: D(q)
  INTEGER :: C(q), R(N+1)
  REAL :: B(M), S(N)
  INTEGER :: I, K
   DO I = 1,M
       S(I)=0.0
       DO K = R(I), R(I+1)-1
           S(I) = S(I) + D(K)*B(C(K))
       ENDDO K
   ENDDO I
```

Figure 8: Sparse matrix-vector multiply: core loop of the sequential algorithm

We first take a look at the sequential algorithm. Consider the operation $S = A.B$, where $A(1:N, 1:M)$ is a sparse real matrix with q nonzero elements $A(i_k, j_k), 1 \leq k \leq q$, and $B(1:M)$ and $S(1:N)$ are real vectors. The enumeration of the nonzero elements of A is based on row-major order.

In the **Compressed Row Storage (CRS)** format, A is represented by three vectors, D, C, and R:

- the **data vector**, $D(1:q)$, stores the sequence of nonzero elements of A, in the order of their enumeration: $D(k) = A(i_k, j_k)$ for all k.
- the **column vector**, $C(1:q)$, contains in position k the column number of the k-th nonzero element in A: $C(k) = j_k$.
- the **row vector**, $R(1:N+1)$, contains in position i the number of the first nonzero element of A in that row, if such an element exists; otherwise $R(i) = R(i+1)$. Furthermore, $R(N+1)$ is set to $q+1$.

Based upon this representation, the core loop of the sequential algorithm for computing the sparse product S=A.B can be formulated in Fortran as shown in Figure 8.

The first step in developing a parallel version of the algorithm consists of defining a *distributed sparse representation* of A. This essentially combines a data distribution (Section 5.3) for the "virtual" array A with the given sparse format (CRS) in the following sense: a data distribution is determined for A as usual by defining a replication-free distribution function $\delta^A : \mathbf{I} \rightarrow \mathcal{M}$, where $\mathbf{I} = [1:N] \times [1:M]$ is the index domain of A. This determines a local distribution segment for each memory unit u. The distributed sparse representation is then obtained by representing the submatrix constituting the local distribution segment using the CRS format.

A number of data distributions have been discussed for this purpose, including cyclic distributions and *Multiple Recursive Decomposition (MRD)* [57]. We focus here on the MRD distribution, which recursively subdivides the matrix along rows and columns to determine an irregular partition of contiguous rectangular segments, guided by the objective of achieving load balancing by allocating approximately the same number of nonzero elements in each segment.

Assume that the MRD distribution of A creates NN distribution segments, $\lambda^A(u), 1 \leq u \leq NN$, where the memory units in $\mathcal{M}$ to which A is being distributed are numbered from 1 to NN (Def. 3). Each distribution segment, $\lambda^A(u)$, is represented in CRS format in memory unit u.

Based upon this distribution, a parallel algorithm for the sparse matrix vector product can now be derived, as outlined in Figure 9. In order to keep this algorithm relatively simple, we focus on the local submatrix-subvector product and omit the actual partitioning algorithm as well as details such as dynamic array allocation and the computation of the final global sum.

The local matrix-vector product is computed in the method *mat_vec_loc*, which is activated as a separate thread, $\mathbf{t}^u$, in each memory unit, u. For each u, the distribution segment, A^u, is given as $A^u = A(L1(u) : U1(u), L2(u) : U2(u))$, based on its global bounds, and $q(u)$ provides the number of nonzero elements in

A^u. Furthermore, we assume that the components of the CRS representation for A^u are given by the array sections $D(u, 1 : q(u))$, $C(u, 1 : q(u))$, and $R(u, L1(u) - U1(u) + 1)$. Finally, each thread $\mathbf{t}^u$ stores the partial sum computed by it in the temporary vector $TS(u, 1 : N)$. All these structures are set up in the memory of the *my_sparse* macroserver by the partitioning routine *mrd_partition*, which is left unspecified.

The algorithm begins by creating the macroserver *my_sparse*. In the next step, the sparse matrix is generated and distributed, creating a distributed sparse CRS format. As a result of this step, the local representations $D(u, :)$, $C(u, :)$, and $R(u, :)$ as well as all the auxiliary data structures such as the arrays L1,U1,L2,U2, and q are set up. Once this is done, the NN threads $\mathbf{t}^u$ can be generated. They compute partial vectors which are stored in $TS(u, :)$. Finally, the $TS(u, :)$ have to be combined in a global sum to determine the final result vector, S.

We finish this discussion by outlining a number of topics that illustrate the support of the PIM array architecture for this kind of algorithm:

- The CRS representation of the local data segments can be stored and processed locally in each PIM node by microservers.
- The indirect references involving D and B can be resolved in the memory; making the implementation of an inspector/executor scheme [52, 61] much more efficient than for distributed-memory machines.
- The PIM array network offers efficient support for spawning a large number of "similar" parallel threads and for executing reduction operations.

```
MACROSERVER CLASS sparse_template
  INTEGER :: NN, u
  REAL, SPARSE(CRS(L1,U1,L2,U2,D,C,R,q,...) :: A(N,M)
  REAL :: B(M), S(N)
  REAL :: TS(NN,:)
  INTEGER :: L1(NN), U1(NN), L2(NN), U2(NN), q(NN)
  REAL :: D(NN,:), C(NN,:), R(NN,:)
  FUTURE :: F(NN)

  CONTAINS

   METHOD generate_and_partition()
     ! Generates matrix, determines MRD partition, and sets up the distributed data structures and auxiliary data
   END METHOD generate_and_partition

   METHOD mat_vec_loc(u,D,C,R)
   REAL :: D(q(u))
   INTEGER :: C(q(u)), R(L1(u):U1(u)+2)
   INTEGER :: I, K
    DO I = L1(u),U1(u)
        TS(u,1:N) = 0.0
        DO K = R(I), R(I+1)-1
            TS(u,I) = TS(u,I) + D(K)*B(C(K)+L2(u))
        ENDDO K
    ENDDO I
   END METHOD mat_vec_loc

   METHOD global_sum()
     ! Performs global reductions to compute final result vector, stored in S, from the temporary vectors TS(u,:)
   END METHOD global_sum

   METHOD matrix_vector()
      FORALL THREADS(u=1:NN, ON HOME(A(L1(u):U1(u),L2(u):U2(u)))
                F(u) = SPAWN(my_sparse%mat_vec_loc(u,D(u,:),C(u,:),R(u,:))
      WAIT(ALL(F))
   END METHOD matrix_vector

END MACROSERVER CLASS sparse_template

 ! Main program
MACROSERVER(sparse_template) my_sparse = CREATE(sparse_template)
CALL my_sparse%generate_and_partition()
CALL my_sparse%matrix_vector()
CALL my_sparse%global_sum()
```

Figure 9: Sparse matrix-vector multiply: parallel algorithm

B Abstract Machine Interface

B.1 Global System Issues

B.1.1 Global Name Management

1. Global names: classes, methods, macroservers, threads — Sections 4, 5, 6
2. Acquaintance relation — Section 4.3

B.1.2 Global Memory Management

1. Supervisor memory
2. Application memory — Section 4.4.1
3. Memory regions — Section 4.4.1
4. Macroserver memory — Section 5
5. Alignment relationship — Section 4.3

B.2 Macroservers

B.2.1 Macroserver Creation and Management

1. **CREATE** $(C, a_1, \ldots, a_n, cstr, sv)$ — Section 4.4.2
2. **MIGRATE** (*ms*, *cstr*, *sv*) — Section 4.5.1
3. **DESTROY** (*ms*, *sv*) — Section 4.5.2

where,

- C: macroserver class
- $a_1, \ldots, a_n$: argument expressions
- *cstr*: region constraint
- *sv*: status variable
- *ms*: macroserver variable

B.2.2 Method Components and Attributes

Methods are discussed in Sections 4.2 and 5.4. Their components and attributes include:

1. *name*
2. *result type*
3. *formal parameters*
4. *private variables*
5. *code*
6. *attributes*
   (a) access (*public*, *private*)
   (b) *atomic*
   (c) *pure*
   (d) *purest*
   (e) *non-preemptive*

B.2.3 Macroserver Object Structure

Macroserver: $\mathbf{S} = (C, reg, h, \mathcal{V}, M, A, \mathcal{T})$ — Section 5

1. C: macroserver class — Section 4
2. *reg*: region — Section 4.4.1
3. h: home — Section 5.1
4. $\mathcal{V} = (V, state, \delta)$: variable specification — Section 5.2
   (a) V: variable set
   (b) *state*: state of V
   (c) δ: distribution of V
5. M: set of methods — Sections 4.2 and 5.4
6. A: acquaintance relation — Section 4.3
7. $\mathcal{T}$: set of threads — Section 6

B.2.4 Distributions and Alignments

1. **Distribution object**: $\delta^D : \mathbf{I} \rightarrow \mathcal{P}(\mathcal{R}) - \{\phi\}$ — Definition 2
2. **Alignment object**: $\alpha : \mathbf{I}_1 \rightarrow \mathcal{P}(\mathbf{I}_2) - \{\phi\}$ — Definition 4
3. **Distribute statement**: **DISTRIBUTE** (v, dv, sv) — Section 5.3.4
4. **Incremental redistribution**: **INC_REDISTRIBUTE** (v, dv, sv) — Section 5.3.4
5. **Align statement**: **ALIGN** (v_1, v_2, av, sv) — Section 5.3.6
6. **Inquiry functions** — Section 5.3.5

where,

- D: data structure
- $\mathbf{I}$, $\mathbf{I}_1$, $\mathbf{I}_2$: index domains
- $\mathcal{R}$: virtual memory region
- $v, v_1, v_2 \in V$: variables
- dv: distribution variable
- av: alignment variable
- sv: status variable.

B.3 Threads

B.3.1 Thread Specification

1. **Thread specification**: $\mathbf{t} = (mid, h, f, \mathbf{a}, status, result, \mathcal{V}_\mathbf{t})$ — Section 6.1

where,

- *mid*: global method identification
- h: home of the thread
- f: future variable
- $\mathbf{a}$: input vector
- *status*: thread status
- *result*: container for the result value
- $\mathcal{V}_\mathbf{t}$: specification of the private variables

B.3.2 Thread Status

Thread status is discussed in Section 6.1.1.

1. *thread attributes*: *atomic*, *non-preemptive*
2. *completion_status*: *pending*, *completed*
3. *blocking_status*: *blocked*, *active*
4. *error_status*
5. *scheduling_status*
6. *detachment_status*

B.3.3 Pure Thread Functions

Pure thread functions are discussed in Section 6.2.

1. **logical function completed** (fex)
2. **logical function blocked** (fex)
3. **logical function error** (fex)
4. **logical function detached** (fex)
5. **logical function non-preemptive** (fex)
6. **integer function priority** (fex)
7. **logical function equal** (fex, fex')
8. **future function who_am_I** ()

where,

- *fex*, *fex'*: future expressions

B.3.4 Thread Manipulation

1. $f =$ **SPAWN** $(ms, m, arg_1, \ldots, arg_n, status\_input, wv)$ Section 6.3
2. **TERMINATE** (f) Section 6.4

where,

- f: future variable
- ms: macroserver variable
- m: method identification
- $arg_1, \ldots, arg_n$: argument list
- *status_input*: status information
- wv: work distribution variable

B.3.5 Work Distributions

1. **Work distribution object**: $\omega^{\mathbf{T}} : \mathbf{T} \rightarrow \mathbf{R}$. Section 6

where,

- $\mathbf{T}$: set of threads
- $\mathcal{R}$: region

B.3.6 Schedules

1. **Schedule function**: $\sigma : \mathcal{R} \rightarrow \mathcal{P}(\mathbf{I})$ Definition 7
2. **Data structure schedule**: (D, δ, σ, d) Definition 8

where,

- $\mathcal{R}$:region
- **I**: index domain
- (D, δ): distributed data structure with index domain **I**
- σ: schedule function with domain $\mathcal{R}$ and range $\mathcal{P}(\mathbf{I})$
- d: direction: "R" (gather schedule), or "W" (scatter schedule)

B.4 Synchronization Operations

B.4.1 Condition Synchronization

Section 7.2.2

1. **WAIT** (c)
2. **SIGNAL** (c)
3. **SIGNAL_ALL** (c)
4. **EMPTY** (c)

where,

- c: condition variable

B.4.2 Future Synchronization

Section 7.3

1. **WAIT** (fex)

where,

- fex: future expression