The Gilgamesh Processor-in-Memory Architecture and Its Execution Model *

Hans P. Zima[a] 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, California, U.S.A.
E-mail: {zima,tron}@cacr.caltech.edu

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 paper, we describe key features of the *Gilgamesh* PIM architecture developed at Caltech and JPL, and introduce an object-based execution model for this architecture based on the notion of *macroservers*. Macroservers are associated with a state space in which a set of lightweight threads executes asynchronously; the location and distribution of objects and data in PIM memory can be dynamically controlled. We discuss the elements of a software architecture for Gilgamesh and illustrate some aspects of macroservers by outlining a sparse matrix algorithm.

1 Introduction

Over the past two decades, microprocessor and memory technologies progressed along different paths. Whereas microprocessor fabrication lines have been optimized towards providing fast transistors and many metal layers to accelerate communication and simplify power distribution, DRAM development has focused on achieving high density and low cost. A result of this separation is the *von Neumann bottleneck*, which is characterized by the significant divergence of processor performance (increasing at a rate of 60% per year) and DRAM access time (improving by only about 7% per year). The necessity to deal with the growing distance between processor and main memory has led to deep cache hierarchies and highly sophisticated microprocessor designs employing techniques such as super pipelining, superscalar execution, and sophisticated branch prediction schemes.

Processor-in-Memory (PIM) systems present a radically different approach. The essential idea is the integration of processor and memory on the same chip exploiting recent advances in semiconductor fabrication processes. Basic characteristics include:

1. integration of CMOS logic and DRAM memory on the same chip,
2. replication of processor/memory nodes across a chip,
3. elimination of cache hierarchies,
4. efficient memory operations and wide-word processing, and
5. simplification of the processor.

*The work described in this paper was partially supported by the Special Research Program F011 "AURORA" of the Austrian Science Fund (FWF) and the Center for Advanced Computation Research (CACR) at the California Institute of Technology in Pasadena, California.

The opportunity of PIM is primarily one of the bandwidth available to the processor. Processing logic has direct access to all the bits of a memory row (typically 2048); a new generation of very wide ALUs and corresponding instruction sets exploit this high internal bandwidth (on the order of 100 Gb/sec) yielding a potential performance of 10 Gips (32-bit operands) on a memory chip with a 16 MB capacity.

PIM favors a new processor design that simplifies operation, lowers development cost and time, and improves efficiency and performance over classical processor architecture. Some sophisticated mechanisms incorporated in today's processors are 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. Among the most important new capabilities are on-PIM virtual to physical address translation, message driven computation, and multithreading [25, 27].

Because of the efficiencies derived from staying on-chip, power consumption can be an order of magnitude lower than for comparable performance with conventional microprocessor-based systems. But dramatic advances in performance will be derived from large arrays of tightly coupled PIM chips.

The principal idea underlying PIM is not new. 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 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 since been undertaken to explore this new design and application space. PIM is being pursued as a means of accelerating array processing [17] and for providing a smart memory in conventional systems [13, 9]. It has also been considered for the management of systems resources in a hybrid technology multithreaded architecture (HTMT) for ultra-scale computing [19]. In 1999, IBM announced the *Blue Gene* project, a PIM-based development of a million-chip system expected to achieve Petaflops performance by 2004 [12, 1, 2]. More recently, Caltech and JPL have initiated the design of the *Gilgamesh* system, a PIM array targeted towards scalable spaceborne computing.

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 a large number of PIM processors to engage in concert in the solution of one or more parallel applications. 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. PIM favors data oriented computing where operations are scheduled and performed at the site of the data; 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 paper focuses on the requirements and the design of an execution model for PIM arrays that binds the actions of the independent processor/memory pairs distributed throughout the PIM array into a single coherent parallel/distributed computation. Major requirements for such a model include

1. Features for structuring and managing the global name space.
2. Control of object and data allocation, distribution, and alignment, with special support for sparse and irregular structures, as well as dynamic load balancing.
3. A general thread model, as a basis for expressing a range of parallel and distributed execution strategies, with a facility for binding threads to data.
4. Support for an efficient mapping of the model's features to the underlying low-level mechanisms and the operating system nucleus.

Additional requirements, which will not be further discussed here, include the logical interface to I/O, protection issues, and the recovery from failed tasks and exceptional conditions.

This paper is structured as follows. The next section provides a short outline of the Gilgamesh hardware architecture. Subsequently, we present an intermediate-level execution model focusing on the concept of the macroserver (Section 3). The discussion will center around the facilities of the model supporting the control of data and work allocation in a PIM array. Section 4 illustrates some features of the model using a sparse matrix-vector multiply as an example. The paper ends with concluding remarks in Section 5.

2 A Short Outline of the Gilgamesh Hardware

The global structure of a **Gilgamesh** (*Giga Logic Gate Assemblies with Mesh Integration*) system is characterized by a set of identical *modules* connected by a network. Modules are hardware-implemented on a chip and contain a fixed number of identical PIM *nodes*. Each node integrates a region of physical DRAM memory with an independent processing facility, the node processing unit.

The Gilgamesh architecture does not provide data or instruction caches in the usual sense. However, each node has a large number of registers which can serve a variety of purposes including that of a programmed data cache.

Key hardware supported features of the Gilgamesh system include a global virtual shared address space, message-driven computation, and multithreading. The first two of these features distinguish Gilgamesh from the IBM Blue Gene system.

2.1 Virtual Shared Address Space

Gilgamesh provides a global virtual shared address space in the sense that virtual addresses can be mapped (with hardware support) to physical addresses across the whole system. A set of new techniques have been developed that enhance the efficiency of virtual-to-physical address translation employing set-associative techniques similar to some cache schemes and using wide registers of the node processing units as programmed TLBs [27].

It is important to note that nonlocal *direct* access (e.g., via a *read* or *write* instruction) from a given module to an address with a physical location in another module is not directly supported by the hardware but must be performed via *parcels*, a communication mechanism related to *active messages* [21, 8] (see below). Nonlocal access latency is expected to be at least one order of magnitude larger than local access latency, implying the importance of spatial locality for the programming model.

2.2 Message-Driven Computation – Parcels

A second important feature of the Gilgamesh architecture is message-driven computation. Like simple memories, PIMs acquire external requests to access and manipulate the contents of memory cells; however, PIMs may also have to perform complex sequences of operations in memory defined by user application or supervisor service routines. Message driven computation features fast on-module interpretation mechanisms that quickly identify both the operation sequence to be performed and the data rows upon which to be operated.

Support for message-driven computation focuses on the concept of parcels first proposed in the HTMT and DIVA projects [5, 9]. Parcels are messages that convey both data and command specifiers to direct work at remote sites and determine a continuation path of execution upon completion. Similar to active messages, parcels direct all inter-module transactions. A parcel may cause something as simple as a memory read operation to be performed or something as complex as a search through all PIM memory for a particular key field value. Parcel assimilation therefore becomes a critical factor in the effective performance obtainable from a given PIM. Parcels can be stored in row-wide registers and processed directly by the wide ALU of the node processing unit.

2.3 Multithreading

A third feature is the incorporation of multithreading into the Gilgamesh architecture. Multithreading permits rapid response to incoming service requests with low-overhead context switching and also enables overlapping of computation, communication, and memory access. Although counter intuitive, multithreading actually simplifies processor design by providing a uniform hardware methodology for dynamically managing physical processor resources and virtual application tasks.

3 Macroservers

Figure 1 provides an overview of the Gilgamesh global software architecture. Applications – operating in a virtual address space of their own – execute as asynchronous networks of dynamic objects called *macroservers*. Macroservers are associated with sets of data, methods, threads, and rules governing the relationship to their environment; their representation can be distributed across the system. Macroservers can dynamically adapt their *behavior* to changing conditions in the environment, thus offering a powerful mechanism supporting not only adaptive and dynamic numerical codes but also intelligent agents [22]. Macroservers have been described in detail in [25]; an overview is given in [26, 27].

Node servers represent a level of the software system close to the hardware. They represent mini-objects with only a local life on a node. A distributed data structure contained in a macroserver is usually mapped to a set of node servers handling the local management of the associated distribution segments.

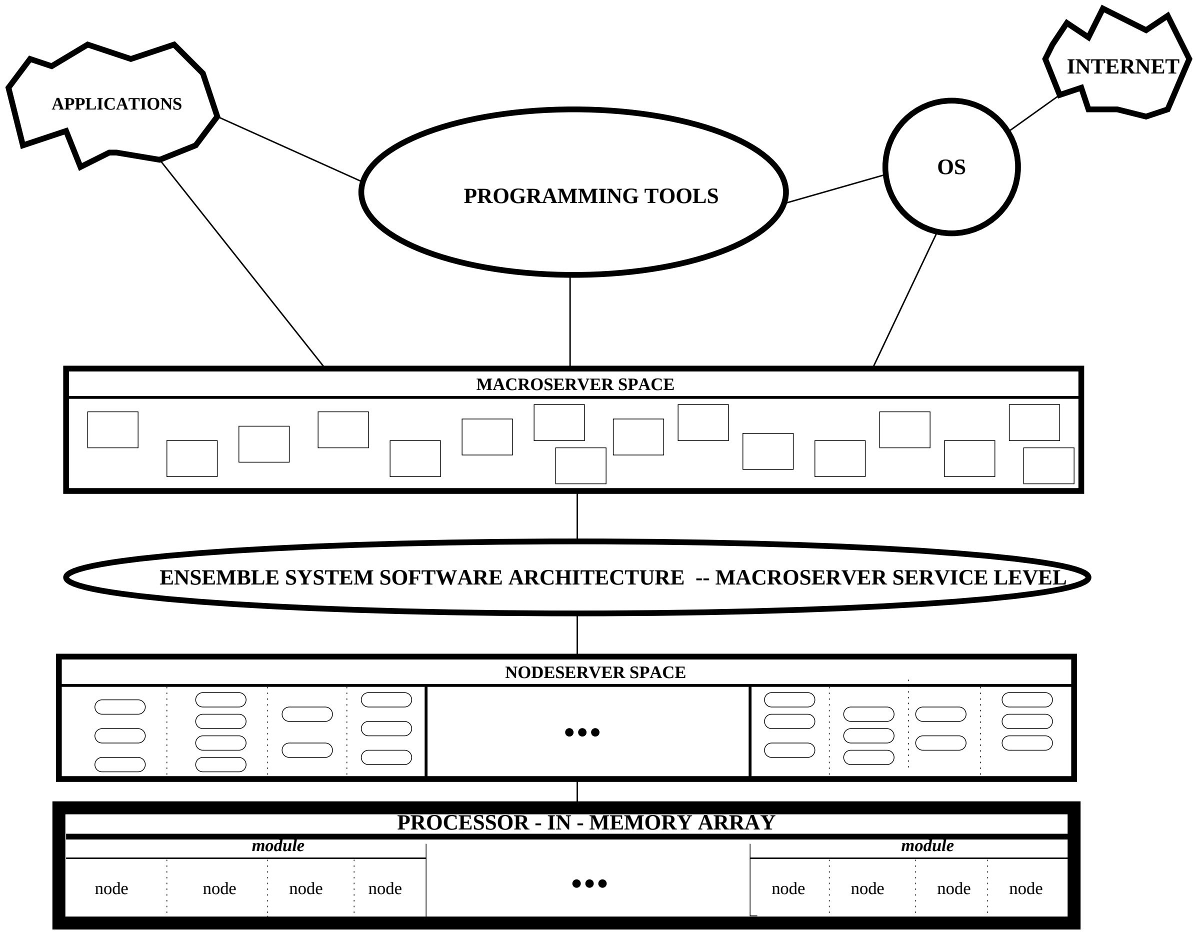

Figure 1: Global software architecture

3.1 A Brief Overview of Key Concepts

A macroserver is an object generated within a given application and operating in its address space. At any point in time, a macroserver is associated with a state space in which a set of asynchronous threads is operating, each of which being the result of the spawning of a method. The data structures of a macroserver can be distributed across the memory region allocated to it. We provide explicit functionality for specifying the initial distribution of data and their incremental redistribution depending on dynamically arising conditions. Whereas the basic ideas of this feature originate from data parallel languages [6, 11], we have generalized this concept to include arbitrary distribution functions and to apply to general data structures such as those provided by LISP lists. Furthermore, the model offers functionality for controlling the location in memory where a thread is to be executed. Such bindings can be established dynamically and are particularly important for linking threads to data on which they operate as well as for dealing with irregular computations.

Threads are lightweight and execute asynchronously. Recursive spawning allows the construction of thread systems with arbitrary depth and structure. Examples include systems of data parallel threads and multidisciplinary threads exposing a two-level hierarchy combining coarse-grain task parallelism with the underlying data parallelism of the individual discipline codes [14].

Mutual exclusion can be controlled via atomic methods; synchronization of threads can be achieved via condition variables [3] or futures. *Future variables*, which are related to the futures in Multilisp [10], can be bound to threads and used for implicit or explicit synchronization based upon the status of the thread and a potential value yielded by it.

3.2 Data Distribution and Alignment

The approach to data distribution and alignment for macroservers generalizes previous work in the context of data parallel *Single-Program-Multiple-Data (SPMD)* languages, in particular Vienna Fortran [24, 6] and HPF [11, 15]. More recently, the problems with achieving satisfactory performance on distributed shared memory architectures such as the Origin2000 resulted in a number of proposals integrating HPF-like data distribution directives into the OpenMP language [18, 7, 4].

We first identify the atomic components of a data structure by it with a mapping, $D : \mathbf{I} \rightarrow \Omega$, where $\mathbf{I}$ is an *index domain*, and Ω is some "universal" set of values [1]. The idea here is that we can always decompose a data structure into its "atomic" components, which designate elementary values such as 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, 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.

When dealing with locality we have to make assertions about the mapping of variables to the physical memory of the system. Gilgamesh offers a global shared address space, but unlike the situation in an SMP or the Tera MTA (which essentially presents a PRAM to the user) the Gilgamesh architecture has a pronounced NUMA characteristic, with the latency ratio between module-local and nonlocal accesses ranging between 1 and 2 orders of magnitude. The explicit control of the mapping to physical space actually often forces the compiler or runtime system to break up a contiguous portion of virtual address space into smaller pieces consistent with the distribution strategy (see below).

In the following we will use an **abstract module set**, $\mathbf{Y}$. By this we mean a set of "virtual" modules which are in some way (how this is done is not specified here) 1-1 mapped to physical modules of the system.

Let D denote a data structure, $\mathbf{I}$ its index domain, and $\mathbf{Y}$ an abstract module set. A **data distribution**, δ^D, for D is defined as a total function $\delta^D : \mathbf{I} \rightarrow \mathcal{P}(\mathbf{Y}) - \{\phi\}$. Here we allow replication, i.e., the mapping of indices to more than one module. However, in most of what follows, we will only deal with *replication-free* distributions which can be interpreted as functions $\delta^D : \mathbf{I} \rightarrow \mathbf{Y}$. Also, we do not deal here with the mapping from the index domain to the *virtual* address space associated with a distribution, which often requires a rearrangement of virtual addresses. For a simple example, consider a *(block,block)* distribution of a 2D matrix, which creates an ordering of virtual addresses different from that in a standard contiguous representation.

Let $\mathbf{i} \in \mathbf{I}$ be arbitrarily selected, and $\mathbf{y} \in \delta^D(\mathbf{i})$. This is equivalent to an assertion that the memory representation of the data structure element $D(\mathbf{i})$ has a representation in module $\mathbf{y}$. The pair (D, δ^D) is called a **distributed data structure**.

Assume D, $\mathbf{I}$, and $\delta^D : \mathbf{I} \rightarrow \mathcal{P}(\mathbf{Y}) - \{\phi\}$ are given as above. Further, let $\mathbf{y} \in \mathbf{Y}$ be arbitrarily selected. The set of all indices mapped to $\mathbf{y}$ is called the **distribution segment**, $\lambda^D(\mathbf{y})$, of $\mathbf{y}$: $\lambda^D(\mathbf{y}) := \{\mathbf{i} \in \mathbf{I} \mid \mathbf{y} \in \delta^D(\mathbf{i})\}$.

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

As we will see later (Section 4), the mechanism introduced here is general enough to also deal with highly irregular data structures, such as sparse matrices.

Distributions of a data structure will in general be created at the time the data structure is allocated; furthermore, macroservers provide special methods which allow a full or partial (incremental) redistribution of distributed data structures.

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 is established between their distributions. This can be formally defined in a similar way as distributions [6].

In addition to providing features for the control of data distribution and alignment, macroservers also offer a functionality to specify work distribution, in particular to bind a set of parameterized threads to the distribution segments of a data distribution. For details see [25].

3.3 Other Methods to Control Virtual-to-Physical Memory Mapping

The previous section has illustrated how the mapping from virtual to physical memory can be controlled at a high level of abstraction via declarative as well as imperative constructs for the distribution, redistribution, and alignment of data. Questions arising in this context include the following:

1. what precisely is the information the hardware and the lowest levels of the software need in order to be able to take advantage of distribution and alignment information, and

2. how can that information communicated via the hierarchical levels of the software system to the machine/assembly language.

A number of authors have applied page migration based on page access statistics. A recent paper describes a page migration engine, used in an OpenMP context, which is controlled by execution traces and simulates data distribution and redistribution in iterative parallel codes [16]; see also [7].

4 Sparse Matrix Vector Multiply

This case study is based on a specific data parallel approach to a sparse matrix vector multiplication originally proposed in [20]. We outline the sequential algorithm, discuss the distribution and representation of the matrix in the memory of a Gilgamesh system, and subsequently formulate a parallel algorithm based on macroservers.

4.1 Sequential Algorithm

Consider the operation $S = A.B$, where $A(1:N, 1:M)$ is a sparse matrix with q nonzero elements, and $B(1:M)$ and $S(1:N)$ are vectors. We assume that the nonzero elements of A are enumerated using row-major order; the k-th element in this order is called the k-th nonzero element of A. An example for A is given in Fig. 2: the non-zero elements ($q = 16$) are explicitly indicated (for simplicity, we assume their values identical with their order in the enumeration); all other fields are assumed 0.

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;

- the **column vector**, $C(1:q)$, contains in position k the column number, in A, of the k-th nonzero element in A; and

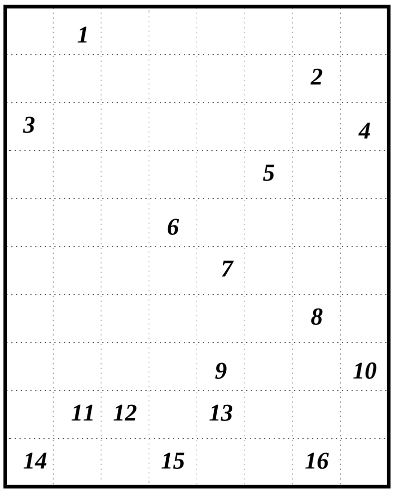

Figure 2: Undistributed sparse matrix A

- the **row vector**, $R(1:N+1)$, contains in position i the number of the first nonzero element of A in that row (if any); else the value of R(i+1). $R(N+1)$ is set to $q+1$.

Based upon this representation, the core loop of the sequential algorithm can be formulated in Fortran as shown in Figure 3.

```
INTEGER :: C(q), R(N+1)
REAL :: D(q), B(M), S(N)
INTEGER :: I, J
 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 3: Sparse matrix vector multiply: core loop of sequential algorithm

4.2 Distributed Sparse Representation

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 with a sparse format such as CRS. More specifically, a data distribution is interpreted as if A were a dense array. The distributed sparse representation is then obtained by representing the submatrices constituting the distribution segments in the CRS format.

A number of data distributions have been used for this purpose, including *Multiple Recursive Decomposition (MRD)* and cyclic distributions [20]. MRD partitions A into NN rectangular distribution segments, $A\_u, 1 \leq u \leq NN$, where NN is the number of available modules. These segments are constructed by a recursive algorithm aiming at associating approximately the same number of nonzero elements with each segment. Fig.4 illustrates an irregular distribution of our example matrix, A, into four rectangular distribution segments, and their associated representations.

4.3 Macroserver-Based Parallel Algorithm

Based upon a distributed sparse representation using MRD, we derive a parallel algorithm for the sparse matrix vector product, as outlined in Figure 5. 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, t_u, in each module u.[2] The distribution segments, A_u, are given as $A_u = A(L1(u) : U1(u), L2(u) : U2(u))$, based on their 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 t_u stores its contribution to the partial sum in the temporary vector $TS(u, 1 : N)$.

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 t_u can be generated. They compute partial vectors which are stored in $TS(u, :)$. Finally, the $TS(u, :)$ are combined in a global sum to determine the final result vector, S. Note that each thread t_u is by itself an activity that can be subdivided into a set of threads, one on each node of the module involved in the computation. This will not be made explicit in the algorithm.

The vector B is distributed in some appropriate way which is not further specified and about which no knowledge is assumed.

5 Conclusion

In this paper we first discussed the salient features of PIM systems, focusing on the recently initiated Gilgamesh project. We then presented the highlights of macroservers, an object-based programming and execution model for Gilgamesh. A revision and first implementation of this model is being developed in parallel with the design of the Gilgamesh ISA and a simulator for the system.

Many questions remain open at this time. The behavior of the Gilgamesh system will have to be studied under benchmarks from a range of applications including the domains of numerical computation, artificial intelligence (in particular, systems of autonomous agents), and monitoring and control. The results of such studies will allow the fine tuning of macroserver semantics and its implementation. Also, the mapping from high-level languages to macroservers will be analyzed, leading to associated compilation and runtime mechanisms as well as a set of supporting software tools.

Finally, the question of the future role of PIMs and their relationship to other developments in hardware and software is yet to be decided. Important current directions include smart caches in conventional systems, the fast growth of memory system sizes, systems and SMPs on a chip and the approach of the billion-transistor chip. In summary, it seems safe to say that the new paradigm provided by PIMs offers exciting challenges for parallel systems architecture of the future.

References

[1] G.S.Almasi,C.Cascaval,J.G.Castanos,M.Denneau,W.Donath, M.Eleftheriou, M.Giampapa, H.Ho, D.Lieber, J.E.Moreira, D.Newns, M.Snir, and H.S.Warren,Jr. Demonstrating the Scalability of a Molecular Dynamics Application on a Petaflop Computer. Research Report RC 21965, *IBM T.J. Watson Research Center*, Yorktown Heights, NY 10598, February 2001.

[2] G.S.Almasi,C.Cascaval,J.G.Castanos, D.Lieber, and J.E.Moreira. Developing System Software for Blue Gene. Research Report RC 21999, *IBM T.J. Watson Research Center*, Yorktown Heights, NY 10598, March 2001.

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

[4] J.Bircsak,P.Craig,R.Crowell,Z.Cvetanovic,J.Harris,C.A.Nelson, and C.D.Offner. Extending OpenMP for NUMA Machines. *Proc.SC2000*, November 2000, Dallas, Texas.

[5] 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.

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

[2] We use u as a shorthand for module $MD(u)$.

[7] 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.

[8] D.E.Culler,J.P.Singh, and A.Gupta. Parallel Computer Architecture. A Hardware/Software Approach. *Morgan Kaufman Publishers*, 1999.

[9] 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.

[10] 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.

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

[12] IBM Corporation. *http://www.research.ibm.com/bluegene*

[13] Y.Kang,W.Huang,S.-M.Yoo,D.Keen,Z.Ge,V.Lam,P.Pattnaik, and J.Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. *Proc.International Conf.on Computer Design (ICCD)*, Austin, Texas, October 1999.

[14] 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).

[15] 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*, Vol.24,No.3-4,pp.325–354 (1998)

[16] D.S.Nikolopoulos,T.S.Papatheodorou,C.D.Polychronoupoulos,J.Labarta, and E.Ayguade. Is Data Distribution Necessary in OpenMP? *Proc.SC2000*, November 2000, Dallas, Texas.

[17] D.PattersonT.Anderson,N.Cardwell,R.Fromm,K.Keeton,C.Kozyrakis,R.Tomas, and K.Yelick. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, pp.33-44, April 1997.

[18] Silicon Graphics Inc. MIPSPro Fortran 90 Commands and Directives Reference Manual. *Document Number 007-3696-003.*

[19] 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.

[20] 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).

[21] T.Von Eicken,D.E.Culler,S.C.Goldstein,and K.E.Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. *Proc.19th Int.Symposium on Computer Architecture*,Gold Coast, Australia, ACM Press (1992).

[22] G.Weiss (Ed.). Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence. *MIT Press*, 1999.

[23] 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.

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

[25] H.Zima and T.Sterling. Macroservers. An Execution Model for DRAM Processor-In-Memory Arrays. *Technical Report CACR-182, Center for Advanced Computing Research, California Institute of Technology*, Pasadena, CA, February 2000. Also: *Technical Report TR 00-01, Institute for Software Science, University of Vienna, February 2000.*

[26] H.Zima and T.Sterling. Support for Irregular Computations in Massively Parallel PIM Arrays, Using an Object-Based Execution Model. *Proc.Irregular'2000*, Cancun, Mexico, May 2000.

[27] H.Zima and T.Sterling. Macroservers: An Object-Based Programming and Execution Model for Processor-in-Memory Arrays. *Proc.International Symposium on High Performance Computing (ISHPC2K)*, Tokyo, Japan, October 2000.

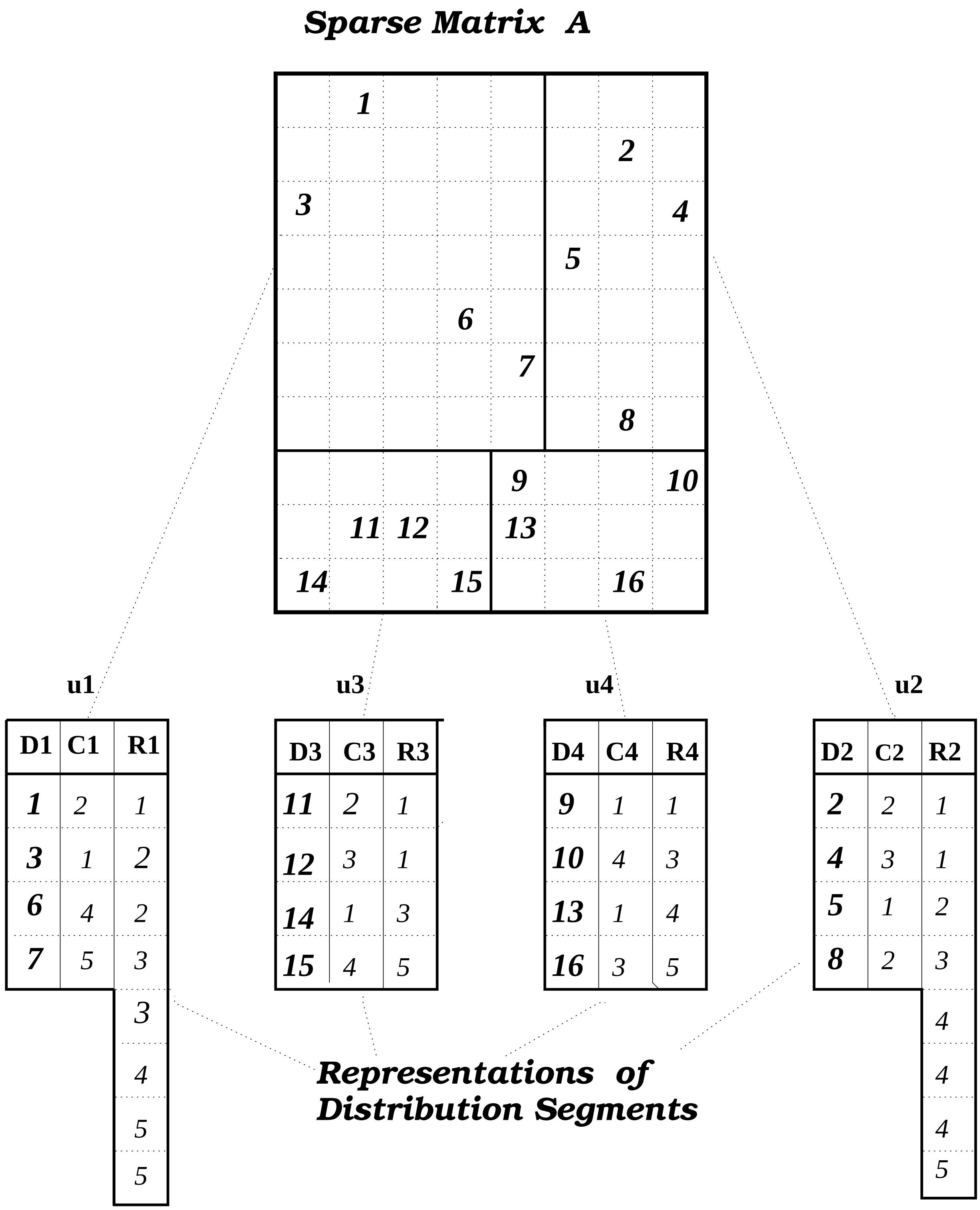

Figure 4: Distributed representation of sparse matrix A

```
INTEGER NN = number_modules()    ! number of modules available for this application
MODULES MD(NN)                   ! abstract module array
MACROSERVER CLASS sparse_template
  INTEGER :: u
  REAL , SPARSE (CRS(L1,U1,L2,U2,D,C,R,q,...),MD) :: A(N,M)
  INTEGER :: L1(NN), U1(NN), L2(NN), U2(NN), q(NN)
  REAL :: D(NN,:), C(NN,:), R(NN,:)
  REAL , DISTRIBUTE (...,:) TO (MD) :: TS(NN,:)
  REAL , DISTRIBUTE (...) TO (MD) :: B(M)
  REAL :: S(N)
  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)+1)
   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,:).
     ! This is a parallel algorithm that uses the future values returned in the matrix_vector routine for synchronization.
   END METHOD global_sum

   METHOD matrix_vector()
         ! Create on each module a thread executing a parameterized version of mat_vec_loc. Provide
         ! arguments that point to the local distribution segments.
      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,:))
   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 5: Parallel sparse matrix-vector multiply using macroservers