

# Support for Irregular Computations in Massively Parallel PIM Arrays, Using an Object-Based Execution Model

*Hans P. Zima*<sup>1,2</sup> and *Thomas L. Sterling*<sup>1</sup>

<sup>1</sup> CACR, California Institute of Technology, Pasadena, CA 91125, U.S.A.

<sup>2</sup>Institute for Software Science, University of Vienna, Austria

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. Furthermore, large arrays of PIMs can be arranged into massively parallel architectures. In this paper, we outline the salient features of PIM architectures and discuss macroservers, an object-based model for such machines. Subsequently, we specifically address the support for irregular problems provided by PIM arrays. The discussion concludes with a case study illustrating an approach to the solution of a sparse matrix vector multiplication.

## 1 Introduction

Processor-in-Memory or PIM architecture couples processor cores and DRAM blocks providing direct access to memory row buffers to increase the memory bandwidth achieved by two orders of magnitude. Current generation PIMs are very basic, treating memory as a physical resource. But future generations of PIM architecture may be well suited to the processing of irregular data structures. PIM based system architecture may take a number of forms from a simple replacement of dumb memory chips with PIM chips to complete systems comprising an array of PIM chips. Manipulating irregular data structures favors specific system architecture characteristics that provide high inter-PIM chip communication bandwidth, efficient address manipulation, and fast response to light-weight service requests.

In this paper, we first characterize some properties of irregular problems (Section 2). The subsequent Section 3 provides a short overview of an object-based execution model for PIM arrays that we are currently developing, in parallel

with the design of a PIM array architecture. Section 4 will then discuss in more detail some of the features of future generation systems that are designed to support the efficient processing of irregular problems. We finish with concluding remarks in Section 6.

## 2 Irregular Problems

Many advanced scientific problems are of an irregular nature and contain a large degree of inherent parallelism. This includes sparse matrix computations, sweeps over unstructured grids, tree searches, and particle-in-cell codes; moreover, many relevant problems are of an adaptive nature. In order to execute such codes efficiently on a massively parallel array of PIMs, a viable tradeoff between the exploitation of locality and parallelism has to be found. As a consequence, the performance of irregular codes is largely determined by the decisions made relating to their data and work distributions.

First, a *data distribution* must be determined by partitioning large data structures and distributing them across the memories of the machine. Secondly, the choice of the *work distribution* has to be made in conjunction with the data distribution, taking into account dependencies and access patterns in the code.

Regular problems, which are characterized by regular, usually linear, data access patterns, can be efficiently implemented on conventional architectures exploiting compile-time knowledge [7]. In contrast, the data access patterns as well as the data and work distributions typical for irregular algorithms must be largely resolved at runtime. Such algorithms pose major problems for most existing parallel machines, mainly as a result of the memory access latency which makes the runtime translation of indirect references and the associated communication very expensive. The problem is compounded by the preprocessing that is often required to effectively organize bulk communication of data [5]. As will be discussed in more detail in Section 4, PIM arrays can offer significant support for this kind of problems.

Below we summarize some of the typical features of irregular applications.

- **Irregular Data Distributions:** The management of irregular data distributions is one of the key issues to be dealt with. Relevant topics include the generation of the data structure, its representation in memory, the implementation of access mechanisms, and (total as well as incremental) redistribution algorithms.
- **Thread Groups:** Irregular algorithms require sophisticated mechanisms for the generation of groups of cooperating threads. One example is a set of data parallel threads working on a problem in loosely synchronous SPMD mode; another one a thread structure arising from a tree search. Critical operations – all of which may be intra-chip as well as inter-chip include thread generation, communication, prefix and reduction operations, mutual exclusion, and condition synchronization.

- **Address Translation:** Address translation refers to the problem of mapping an indirect reference (such as  $A(X(I))$  or a pointer value) to a memory address. For irregular data structures, this translation must, in general, be performed at runtime, since neither the data distributions nor the value of index arrays need to be statically known.

### 3 Macroserver

We have defined an object-based model that provides an abstract programming paradigm for a massively parallel PIM array. Here we give a short outline of this model; for a more detailed specification see [8].

The central concept of the model is called a *macroserver*. A macroserver is an autonomous, active object that comes into existence by being created based upon a *macroserver class*, which essentially establishes an encapsulation for a set of variables and methods as well as a context for threads and their synchronization.

At any point in time, a macroserver has a well-defined *home*, which is a location in virtual memory where the *metadata* needed for the management of the object can be found. Macroserver are associated with a set of variables whose values define its state. Variables may be distributed across the memories of the PIM; the underlying distribution mechanism can be thought of as a generalization of the corresponding concept in languages such as Vienna Fortran [2] or HPF [4], extended to LISP-type data structures and allowing arbitrary mappings as well as incremental redistribution.

The activation of a method in a macroserver gives rise to the generation of a (synchronous or asynchronous) thread. Threads can directly read and modify the state of the macroserver to which they belong. Each thread can activate methods of its own or of another macroserver accessible to it; as a consequence, the model offers intra-server as well as inter-server concurrency, reflecting the multi-level parallelism of the underlying architecture. Special support is provided for the operation of thread groups.

The model provides a simple mechanism for mutual exclusion (atomic methods) and allows synchronization via condition variables and futures.

Metadata includes information about the state variables (such as types and distributions), the signatures of methods, and representations of the threads currently operating in the macroserver. Most of the metadata will be stored at the home of the object, allowing an efficient centralized management by the associated PIM.

### 4 PIM Arrays and Their Support for Irregular Computations

PIM combines memory cell blocks and processing logic on the same integrated circuit die. The row buffer of the typical memory block may be on the order of 1K or more bits which are acquired in a single memory cycle delivering data at

a typical rate of 4 Gigabytes per second at the sense amps of the memory. A processor designed to directly manipulate this data simultaneously can typically operate at a performance of a Gigaops. Multiple memory-processor pairs can occupy the same die with a possible performance total today of 4 Gigaops, with higher rates for simple arithmetic or small field manipulations. A typical memory system comprising an array of PIM chips could provide a total throughput of 256 Gigaops or, for byte level operations, 1 Teraops peak throughput. While in some designs core processors developed as conventional microprocessors are “dropped” into the PIM to minimize development time and cost as well as exploit existing software such as compilers, in other cases much simpler processors may be designed explicitly for the PIM context, minimizing the die area consumed while optimizing for effective memory bandwidth. A typical PIM chip includes multiple modules of memory blocks and processors, memory bus interface, and shared functional units such as floating point arithmetic units. Also, addresses employed within the PIM by the on-chip processors are usually physical.

A system incorporating PIM chips may differ from the conventional structure outlined above in important ways to take advantage of the capabilities of PIM and employ them in the broader system. While in the most simple structure, the PIM chips may simply replace regular memory retaining the systems processor with its layered cache hierarchy, other structures diverging from this will deliver improved performance. Providing a separate highly parallel network for just inter-PIM communications and a second level of bulk storage as backing store for the PIMs are two examples. Beyond this is the “Gilgamesh” (billion logic gate array mesh) which is a multi-dimensional structure of PIM chips in a standalone system.

The processing and manipulation of irregular data structures imposes additional requirements than can be effectively handled with current generation PIM technology and architecture. However, many of the advantages that PIM has for dense contiguous data computing would also convey to metadata-organized irregular data structures if augmented with advanced mechanisms. An important advance that PIM enables is the exploitation of fine grain parallelism intrinsic to irregular structures. This is because many different parts of the distributed structure can be processed simultaneously by the many PIM processors throughout the memory and because the nodes of the structure can be handled efficiently in the memory itself.

Virtual memory management including address translation is central to the advanced architecture requirements of PIM for irregular data. Such data incorporates virtual user addresses within the data structure itself, which must be managed directly by the PIM. Conventional TLB based methods employed by microprocessors are likely to be poorly suited because the access patterns encountered by the PIM internal processors often will not experience the necessary level of temporal locality required to make them effective. An innovative approach employs “intrinsic address translation” in which the virtual to physical address mapping is incorporated in the data structures metadata. With bi-directional links, any page movement across physical memory can cause auto-

matic update of the translation data in other linked pages. A second method is a “set associative” approach that is a hybrid mapping; partly physical and partly virtual as in cache systems but in main memory instead. This allows efficient address translation between memory chips while permitting random placement and location of pages anywhere within a designated memory chip. Both methods when combined provide relatively general and highly scalable virtual to physical memory address translation.

A second requirement is direct inter-PIM message driven communication without system processor intervention. This is being explored by the DIVA project [3] and HTMT project [1]. When a pointer indicates continuation of a structure on an external chip, the computation must be able to “jump” across this physical gap while retaining logical consistency. The “parcel” model [1] permits such actions to follow distributed data structures across PIM arrays. A parcel is a message packet that not only conveys data, but also specifies the actions to be performed. The arrival of a parcel causes a PIM to respond by instantiating the specified thread, carrying out the action on the designated local data, and providing a necessary response, possibly by returning another parcel or by continuing to move through the data structure.

A third requirement is architecture support for thread management. Because of the potentially random distribution of the data structure elements or substructures, the order of service requests may be unknown until runtime. A PIM chip may be required to service multiple parcels concurrently. A PIM comprises several subsystems that can operate simultaneously. Finally, some PIM threads may require access to remote resources such as other PIM chips and such accesses will impose delays. One mechanism not yet incorporated in past PIM implementations is multithreading, a means of rapidly switching among multiple concurrent thread contexts. Multithreading provides an efficient low level mechanism for managing multiple active threads of execution. This is useful while waiting several cycles for a memory access or the propagation delay for a shared ALU to permit other work to be performed by the remaining resources. Each time a new parcel arrives, the active thread can be suspended until the incident parcel is at least stored, thereby freeing the receiver hardware resources for the next arrival. Multithreaded architecture will greatly facilitate the manipulation and processing of irregular data structures because it provides a runtime adaptive means of applying system resources to computational needs as determined by the structure of the data itself.

## 5 Case Study: Sparse Matrix Vector Multiply

In this section, we outline an approach for the parallelization of a sparse matrix-vector multiplication using our model, based partly on concepts developed in [6].

We first take a look at the 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  $A$

```

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 J = R(I), R(I+1)-1
    S(I) = S(I) + D(J)*B(C(J))
  ENDDO J
ENDDO I

```

Figure 1: Sparse Matrix-Vector Multiply: Core Loop

are enumerated using row-major order; the  $k$ -th element in this order is called the  $k$ -th nonzero element of  $A$ .

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
- 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);  $R(N + 1)$  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 1.

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 [7, 8] is interpreted as if  $A$  were a dense array, specifying a local distribution segment for each PIM node. The distributed sparse representation is then obtained by representing the submatrix constituting the local distribution segment in the sparse format (CRS in our case).

A number of data distributions have been used for this purpose, including *Multiple Recursive Decomposition (MRD)* and cyclic distributions [6]. MRD is a method that partitions  $A$  into  $nn$  rectangular distribution segments, where  $nn$  is the number of memory nodes. These segments are the result of a recursive construction algorithm that aims at achieving load balancing by having approximately the same number of nonzero elements in each segment.

Based upon a distributed sparse representation using MRD, a parallel algorithm for the sparse matrix vector product can now be formulated by applying (a slightly modified version of) the algorithm in Fig. 1 in parallel to all distribution segments, and then combining the partial results for each row of the original matrix in a reduction operation. Because of lack of space, we do not

discuss further details of the parallel algorithm here (see [8]). However, we outline 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 [5, 7] 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.

## 6 Conclusion

In this paper, we have discussed the design of massively parallel PIM arrays, together with an object-based execution model for such architectures. An important focus of this work in progress is the capability to deal effectively with irregular problems.

## References

- [1] 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.
- [2] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.
- [3] 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.
- [4] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0*, January 1997.
- [5] 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),pp.303-312, 1990.
- [6] 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).
- [7] 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.
- [8] H.Zima and T.Sterling. Macroservers. An Object-Based Model for Massively Parallel Processor-in-Memory Arrays. *Caltech CACR Technical Report*, January 2000 (in preparation).