Processing-In-Memory Technology for Knowledge Discovery Algorithms

Jafar Adibi, Tim Barrett, Spundun Bhatt, Hans Chalupsky, Jacqueline Chame, Mary Hall
USC Information Sciences Institute
4676 Admiralty way, Marina del Rey, CA 90292
{adibi, jtb, spundun, hans, jchame, mhall}@isi.edu

ABSTRACT

The goal of this work is to gain insight into whether processing-in-memory (PIM) technology can be used to accelerate the performance of link discovery algorithms, which represent an important class of emerging knowledge discovery techniques. PIM chips that integrate processor logic into memory devices offer a new opportunity for bridging the growing gap between processor and memory speeds, especially for applications with high memory-bandwidth requirements. As LD algorithms are data-intensive and highly parallel, involving read-only queries over large data sets, parallel computing power extremely close (physically) to the data has the potential of providing dramatic computing speedups. For this reason, we evaluated the mapping of LD algorithms to a processing-in-memory (PIM) workstation-class architecture, the DIVA / Godiva hardware testbeds developed by USC/ISI. Accounting for differences in clock speed and data scaling, our analysis shows a performance gain on a single PIM, with the potential for greater improvement when multiple PIMs are used. Measured speedups of 8x are shown on two additional bandwidth benchmarks, even though the Itanium-2 has a clock rate 6X faster.

1. INTRODUCTION

Link discovery (LD) algorithms represent an important class of emerging knowledge discovery techniques being used to identify complex, multi-relational patterns. LD algorithms are data-intensive and highly parallel, involving read-only queries over large data sets. Developing link discovery algorithms presents a variety of difficult challenges. First, data ranges from highly unstructured sources such as reports, news stories, etc. to highly structured sources such as traditional relational databases. Unstructured sources need to be preprocessed first either manually or via natural language extraction methods before they can be used by LD methods. Second, data is complex, multi-relational and contains many mostly irrelevant connections (connectivity curse). Third, data is noisy, incomplete, corrupted and full of unaligned aliases. Finally, relevant data sources are heterogeneous, distributed and can be very high volume.

Proceedings of the Second International Workshop on Data Management on New Hardware (DaMoN 2006), June 25, 2006, Chicago, Illinois, USA.

In this paper, we evaluate the mapping of LD algorithms to a processing-in-memory (PIM) workstation-class architecture, the DIVA / Godiva hardware testbed developed by USC/ISI over the past eight years. PIM architectures integrate processing logic within memory devices to address the well-known processor-memory performance gap [3,4,5,6,7]. Many previous architectural solutions to the processor-memory gap such as multithreading, prefetching, and speculation, seek to reduce or tolerate memory latency, at the expense of increased memory bandwidth requirements [2]. Although recent system features like wide memory channels, multiple memory channels, and double-data-rate (DDR) DRAM have helped to increase the system bandwidth, the memory bandwidth into the processor is still severely constrained. Simply put, chip edge bandwidth, the product of the number of I/O signals per chip and chip I/O signal frequency, has increased at a rate much below that of Moore's Law. PIMs instead dramatically improve memory bandwidth, by orders of magnitude over conventional DRAM systems, because internal processors can be directly connected to the memory banks. Latency to on-chip logic is also significantly reduced because internal memory accesses avoid the delays associated with communicating off chip.

The second generation hardware testbed that is the subject of this study incorporates PIMs as replacement memories in an HP zx6000 Itanium-2 workstation. While several PIM-based architectures have been developed, DIVA represents the first smart-memory PIM device that is capable of executing ***independent threads of control*** (rather than lock-step execution) and designed to support ***in-memory virtual*** addressing (rather than having separate address spaces).

In designing this experiment, we hypothesized that for LD algorithms, parallel computing power extremely close (physically) to the data has the potential of providing dramatic computing speedups. To test whether the memory bandwidth provided by PIMs will accelerate link-discovery algorithms, we selected two representative data and graph mining algorithms, condensed them to their core functionality that exhibits most of the computational complexity, and implemented them on the hardware testbed. To compare with a conventional architecture, we measured the performance of the kernels on the Itanium-2 host. In addition to the raw performance measurements of the LD benchmarks, through appropriate scaling and modeling to reflect current and future technology, we derive an expected performance gain of 1.3X to 2.6X for a single PIM on the LD benchmarks, and predict speedups of as much as 10X for 8 PIMs.

We also developed an algorithm simulation in MATLAB to test out alternative parallel algorithms and the impact of data set properties without having to run a large number of cases on the

hardware testbed. These measurements demonstrate that there is potential for PIMs in speeding up graph clustering as the data scales, and particularly for highly connected graphs. We identified load imbalance, and also found that algorithms that drop communication sacrifice precision with just modest performance improvements.

The remainder of this paper is organized into six sections. The next two sections present background on the PIM system environment followed by our previous work on link discovery algorithms, which provided the foundation for this project. The subsequent section presents the bandwidth and LD benchmarks. The fifth section describes the MATLAB simulation of alternative parallel LD algorithms. The performance analysis and scaling results are presented in the sixth section, followed by a summary.

2. PIM SYSTEM ENVIRONMENT

Figure 1. DIVA DIMM with two PIMs.

The DIVA system architecture was specifically designed to support a smooth migration path for application software by integrating PIMs into conventional systems as seamlessly as possible. DIVA PIMs resemble, at their interfaces, commercial DRAMs, enabling PIM memory to be accessed by host software either as smart-memory co-processors or as conventional memory. In Figure 1, we show 2 DIVA PIMs on a standard memory DIMM board. These PIMs are connected to a host processor through *nearly* conventional memory control logic, with some differences due to how today's memories spread data and addresses across chips, as highlighted in Section 6.1.2. A separate memory-to-memory interconnect enables communication between memories without involving the host processor. Aside from memory bus access the host processor communicates with the PIM using parcels. A parcel is closely related to an active message as it is a relatively lightweight communication mechanism containing a reference to a function to be invoked when the parcel is received.

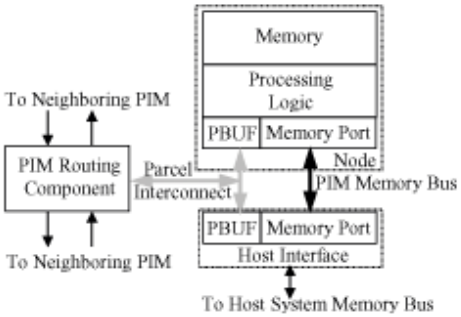

Figure 2. PIM architecture.

Figure 2 below shows two interconnects that span a PIM chip for information flow between nodes, the host interface, and the PIM routing co-Processor (PiRC). Each interconnect is distinguished by the type of information it carries. The PIM memory bus is used for conventional memory accesses from the host processor. The parcel interconnect allows parcels to transit between the host interface, the nodes, and the PiRC. The host interface also contains a parcel buffer (PBUF) for parcel communication between host and PIM. Each PIM node also has a PBUF, for node-to-node parcel communication.

The DIVA PIM node processing logic supports single-issue, in-order execution, with 32 bit instructions and 32-bit addresses. There are two datapaths whose actions are coordinated by a single execution control unit: a 32-bit scalar datapath that performs operations similar to those of standard 32-bit integer units, and a 256-bit Wide-Word datapath that performs fine-grain parallel operations on 8-, 16-, or 32-bit operands. The Wide-Word datapath is similar to multimedia extensions like AltiVec. Both datapaths execute from a single instruction stream under the direction of a single 5-stage pipeline, complete with register forwarding logic to resolve data dependence hazards. This pipeline fetches instructions from a small instruction cache, which is included to minimize memory contention between instruction reads and data accesses. Each datapath has its own independent general-purpose register file with 32 registers. Special instructions permit direct transfers between register files without going through memory.

The PIM devices used in the experiments represent our second-generation devices. As compared to the first-generation devices, discussed in [3], the Godiva PIMs include address translation and eight parallel floating-point units for concurrent use on the Wide datapath. In addition, their memory implements a Double-Data-Rate (DDRAM) interface for integration into the Itanium-2 zx6000 workstation.

The compiler for the DIVA PIMs automatically exploits the parallelism of the DIVA WideWord unit by generating SIMD instructions and using the wide register file as a compiler-controlled cache [9,10].

3. LINK DISCOVERY ALGORITHMS

The link discovery codes used in our experiment focus on a specific problem called ***group detection.*** The group detection task can be qualified into either (1) discovering hidden members of *known groups* (or group extension), or (2) identifying completely *unknown groups*. A known group is identified by a given *name* and a set of known members. The problem then is to discover potential additional hidden members of such a group given evidence of communication events, business transactions, familial relationships, etc. In our study, we only focus on *known groups*. To extend known groups we generally start with a set of known seed members (*e.g.*, a group of suspects) and then proceed in three phases. First, a function θ is applied to find likely new candidates for each group, producing an extended group. Second, the same function θ is used to rank these likely members by how strongly connected they are to the seed members. Third, the

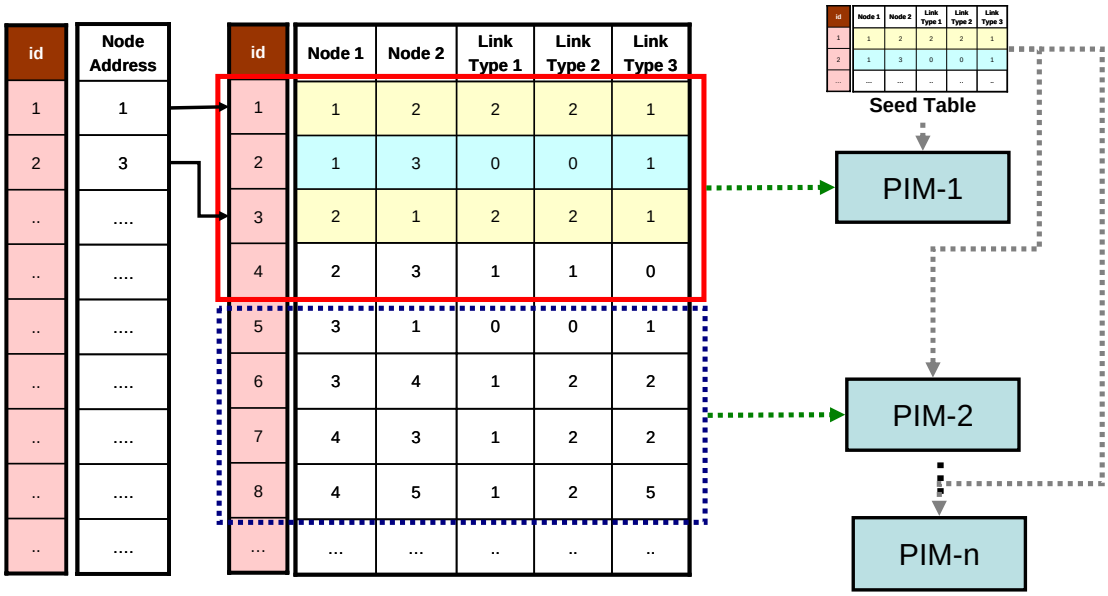

Table 1: Illustration of Node Table and Pair Table. Pair table splits among PIMs and each PIM also has the seed table.

ranked extended group is pruned using a threshold to produce the final output. To discover *unknown groups* we need to locate some initial points and treat them as seed members of a virtual group and apply a similar technique to *known groups*. The process of ***Seed Selection*** can use any number of techniques, but we are using seeds provided by the sample data sets discussed in the next section. Once we have a set of seeds, ***Group Detection*** discovers potential additional members of groups represented by each seed. Group detection looks for entities that are strongly connected with one or more of the seed members. To compute θ we transform the problem space into a graph in which each node represents an entity (such as a person) and each link represents the set of actions (*e.g.*, emails, phone calls etc.) in which the entities are involved. There are several choices for the function θ. For instance we can use ***mutual information (MI)*** between random variables representing individuals' activities which provides a measure of their dependence. To find two strongly connected entities, we aggregate the known links between them and statistically contrast them with connections to other candidates and the general population [1].

Alternative approaches to MI are, for example, conventional social network techniques such as InOutRatio (IOR) for graph clustering. In this method the measure of membership is the ratio between the number of links within the group and the number of links from group members to nodes outside the group.

4. BENCHMARKS

Prior to the experiments with the LD kernels, we performed measurements on two common bandwidth benchmarks. We felt the addition of these benchmarks was important, as it tests out the hypothesis that PIMs really do offer a bandwidth advantage over conventional architectures. In addition, since the bandwidth benchmarks are less complex than the LD kernels, it provided an opportunity to test out our infrastructure and timing methodology. The following are all the benchmarks used in the experiment.

Bandwidth Benchmarks:

- **StreamAdd**: a subset of STREAM from the HPC Challenge benchmark, measures the performance of a stream of floating point additions and updates.
- **RandomAccess,** also from the HPC Challenge benchmark suite, is designed to stress the memory system by performing updates to randomly selected entries of a large table. Since the updates are to random memory locations there is effectively no spatial reuse, and each reference induces a memory access.

Link Discovery Codes:

- **Mutual Information**: The Mutual Information kernel was initially written in STELLA, an experimental object-oriented language aimed at symbolic programming applications. STELLA automatically generates C++, and the C++ code was used as the basis for the kernel [12].
- **Graph Clustering using In/Out Ratios**: The second link discovery kernel performs graph clustering as part of a group finding algorithm. The graph clustering component organizes the graph according to the strength of connections within a group as compared to the outside. This code was initially implemented in MATLAB, and was rewritten in C.

Parallel Graph Clustering Algorithm

A parallel PIM InOutRatio algorithm is shown in the appendix. This implementation was developed from the sequential code using MPI so that we could debug it on a conventional platform. We leveraged a working version of the computation phase of each PIM, which is the same as that of the MPI implementation, and then integrated it with the PIM-to-PIM communication phase, replacing the MPI constructs with parcels.

The data and computation are partitioned among PIM nodes as follows. Each PIM keeps a fraction of the PairTable, that is, a subset of the pairs of nodes in the graph, where a pair is a set of two nodes with at least one link between them. The PairTable is

partitioned so that, for a given node, all pairs containing that node are kept on the same PIM. Hence the whole graph is represented in a link table of *twice the number of links* times *the number of all link types* plus 2 (for two nodes). This duplication of information avoids unnecessary cross communication among PIMs during the IOR computation. In addition to a subset of the PairTable each PIM keeps a copy of the current group in its local memory (initially the current group is the set of seed members). Table 1 illustrates node table and link table.

The algorithm iterates until a desired number of new nodes is added to the group, finding a new member at each iteration, or until there are no more nodes to be added. At each iteration each PIM computes the node, among the nodes in its subset, with highest InOutRatio. After that, all PIMs communicate to find the node with highest InOutRatio across all PIMs. Finally, at the end of each iteration, each PIM adds the new "global" best node to its local copy of the current group.

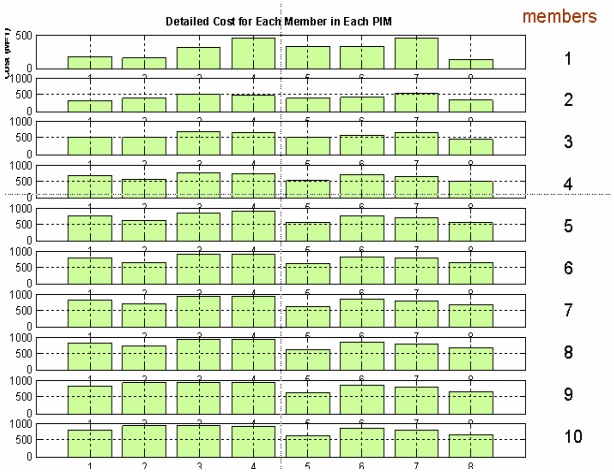

Figure 3: Load imbalance: 8 PIMs, Group Size = 10, M = 2

5. GRAPH CLUSTERING SIMULATION

Performance measurement on the PIMs takes a significant amount of work. To facilitate experimenting with alternative algorithms and algorithm parameters, we developed a MATLAB graph clustering framework to quickly prototype a variety of alternative parallel implementations. We simulated the parallel processing procedure in PIMs to achieve the following goals:

1. Compare the result of the simulated version vs. actual parallelization for correctness.

2. Measure and compare a set of performance-related indices to compare different algorithms and communication strategies.

We have developed a prototype of the specific parallel algorithm from the appendix and minor variations of it, and instrumented this implementation to gather performance-related indices. The data sets evaluated for group detection are derived from synthetic data developed by Information Extraction & Transport, Inc. The main design focus of these datasets is large amounts of relationships between entities. The artificial world consists of *individuals* that belong to *groups* and exploit *targets*. Groups can be *threat groups* or *non-threat* groups that exploit targets in threat and non-threat ways. Individuals are *threat* individuals or *non-threat* individuals. Every threat individual belongs to at least one threat group. Non-threat individuals belong to non-threat groups. Threat groups have only threat individuals as members. Threat individuals can belong to non-threat groups as well. Most of the simulations use Y3-DATASET_4028.

To run the IOR over a set of PIMs we split the Link Table and distribute the load over all PIMs, as described in the previous section. In addition each PIM has the seed link information. In each iteration, each PIM finds the node with the highest IOR and reports the result to other PIMs. Using this basic approach, we identify a couple of variations on the algorithm in Section 4. First, we can evaluate the impact of reporting one or more members at each iteration **(M)**. Obviously, if the number of members reported by each PIM is more than one, the overall accuracy of IOR decreases, but parallelism increases and communication costs are reduced. As with other such algorithms, small differences in results can be tolerated. We can also vary the group size (**N**) and the number of PIMs (**A**). We assume PIMs communicate with each other hierarchically. Each PIM is located in a leaf of a balanced tree and reports the best node to its parents. The results of these variations are shown in this section. We ran our simulation of IOR on a group of 1, 2, 4, 8 and 16 PIMs.

Estimating Cost

Because we are prototyping the algorithm in MATLAB, any direct execution time measurements of the MATLAB prototype would not necessarily be representative of a production parallel implementation. For this reason, we estimate performance of the PIM computations as the sum of *processing* and *communication cost*. We represent *processing cost* with the total number of links accessed by all PIMs. We represent the *communication cost* with the total number of links passed from one PIM to another.

***Processing cost*:** In each step In/Out ratio treats a node as a potential new group member. The *processing cost* is calculated through the following steps:

1. Count number of all edges connected to this node
2. Count "in" edges between the new node and nodes in current seed group
3. The "out" edges are the differences between 1 and 2
   a) can be calculated once in O(F) time (F = avg. fan out)
   b) calculated in each iteration in O(F) time

***Communication Cost*:** *Communication cost* represents the aggregate message passing among PIMs when the best In/Out ratio is calculated (reporting up) and when the best ratio is reported back to all PIMs (reporting down).

- *Reporting Up*: Report the best node from each PIM plus relevant rows from the link table. The communication cost is the number of rows associated with the best local node.
- *Reporting Back*: Reports the selected node and relevant rows from the link table are sent to all PIMs. The cost is the number of rows associated with the best global node.

With these costs, we estimate performance as we vary the number of PIMs (from 1 to 16), following a hierarchical policy for communication. We also vary the number of members, M, (M=1 and M=2) reported at each iteration. Table 2 summarizes the results. In general, increasing the number of PIMs reduces the cost of overall computation. However, the benefits depend on the actual cost of communication relative to processing.

PIM		1	2	4	8	16
Processing Cost	M=1	35373		10866	6675	4602
	M=2	33923	18427	9251	5078	2770
Communication Cost	M=1	0	123	369	861	1845
	M=2	0	138	369	861	2250

Table 2: Computation and Communication cost. M: number of members reported at each round

Load Balance

Figure 3 breaks down the work performed at each PIM, using 8 PIMs. For this example, M=2. The, total number of new members to be discovered is 10. Thus, each row in the figure shows a step of the ten-step computation, and each PIM is represented horizontally.

From Figure 3, we see that the amount of work at each PIM varies significantly, and also varies across different time steps. However, the trend is toward similar behavior at each time step, related to connectivity of the subgraph allocated to a PIM. Thus, there is load imbalance, suggesting that various strategies for managing load might be useful for this algorithm, such as virtualization and dynamic scheduling. In addition, as the number of seed members increases, each PIM will have a somewhat similar link table (especially in a scale free network where there are a couple of nodes with extremely high degree) and load imbalance reduces consequently. Another important observation is that when the amount of load imbalance increases there is enough time for some of the PIMs to communicate with each other. Table 3 summarizes lessons from this experiment.

1	Seed Member Size	↑	Load Imbalance	↓
2	M (members reported at each round)	↑	Load Imbalance	↑
3	Uniformity in the data	↑	Load Imbalance	↓
4	Load Imbalance	↑	Communication Cost	↓
5	Load Imbalance	↓	Computation Cost	↓

Table 3: Summary of Load Balance Experiment

Connectivity Effect

Connectivity is an important feature of a graph which refers to the average number of edges between nodes in the graph. In this experiment we measured the effect of connectivity on the processing cost. As is illustrated in Figure 4, the processing cost increases when the connectivity increases. However this rise in cost is smaller for 16 PIMs than for smaller numbers of PIMs. Figure 5 illustrates the effect of connectivity on speedup. As it shows, 16 PIMs perform even better as connectivity increases.

6. PERFORMANCE ANALYSIS ON PIMs

Figure 6 illustrates our approach to performance evaluation of codes on the PIM hardware testbed. The process has many steps. Initially, scalar code for a single PIM is developed that

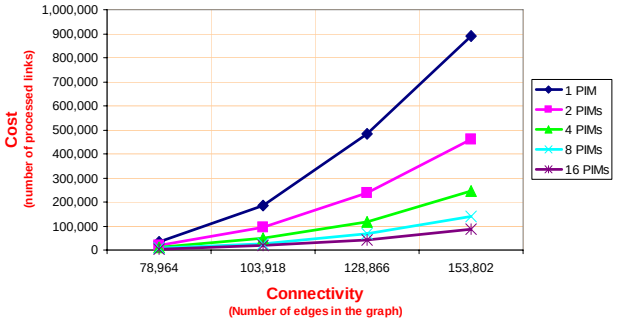

Figure 4: Cost vs. Connectivity

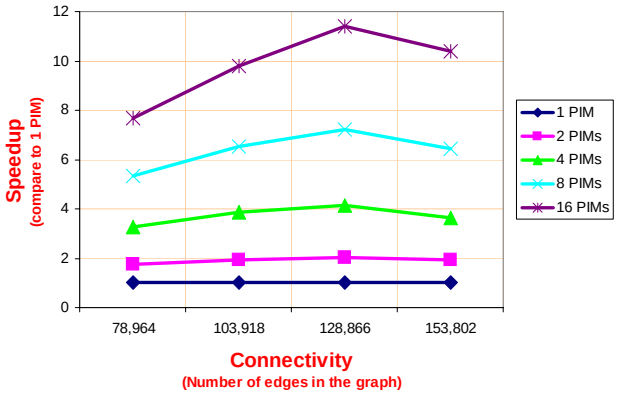

Figure 5: Speedup vs. Connectivity

implements a specific algorithm. As an initial test, we compile and assemble the code, without any optimization (*i.e.*, gcc –O0), and validate its execution on the hardware, using very small TEST INPUT data. To collect performance measurements, the process is far more complex.

The code must be highly tuned. Also, the data sets should be as representative of realistic data sets as possible within the constraints of the system (referred to as SCALE INPUT). It is important that the SCALE INPUT be large enough to stress the Itanium-2 memory system, since this is the regime where PIMs

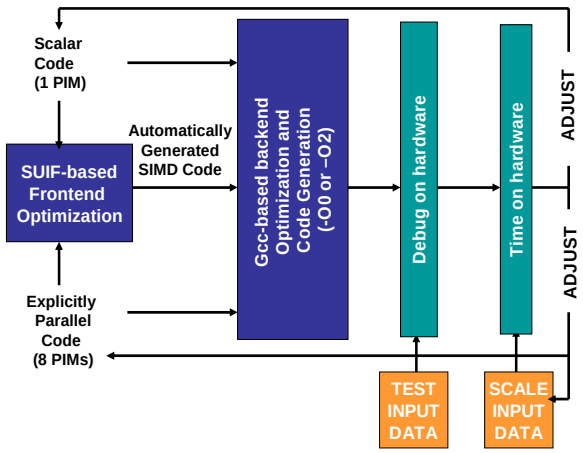

Figure 6. Performance Tuning Process on Hardware.

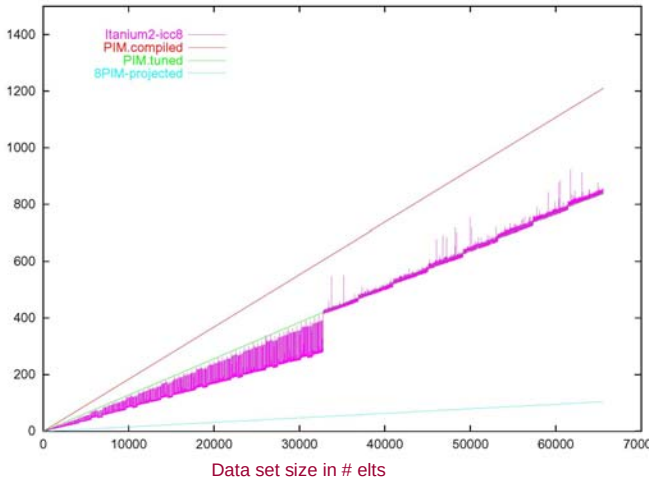

Figure 7. StreamAdd Performance Measurements. (exec time in second vs. dataset size in elts)

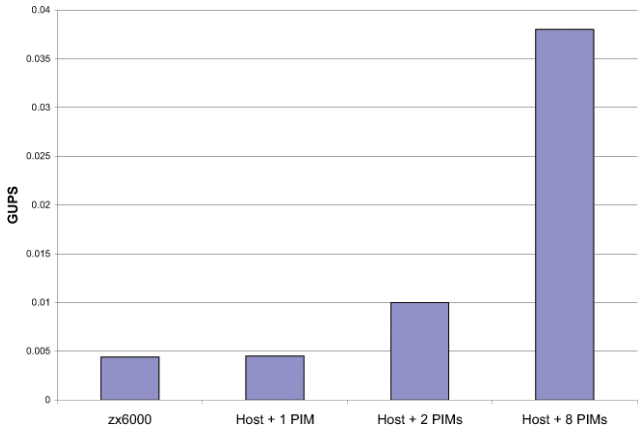

Figure 8. G-updates/second (GUPS) for RandomAccess

will be advantageous. To optimize for single PIM performance, we use the frontend compiler described in Section 2 that converts sequential code to SIMD code to exploit the hardware's WideWord capability. We also use the optimization within the gcc backend (*i.e.*, gcc –O2) to further optimize the code, performing register allocation, constant propagation, strength reduction, and other standard optimizations.

Our performance measurements are compared against the Itanium-2. For this experiment, the host-only versions were compiled with the icc 8.0 C compiler available from Intel with -O3 options, running under Linux version 2.4 on the single processor 900MHz zx6000 workstation. To develop parallel code, we add explicit parallel constructs to the code to communicate between PIMs, and optimize the data/computation partitioning by hand, as in the parallel code of the appendix.

Due to the complexity of the code to be analyzed, it was our goal to minimize any manual intervention in the generation of code. Only very minor hand coding was performed on compiler-generated code – to enhance instruction scheduling in StreamAdd and to optimize control flow in MI.

6.1 Bandwidth Benchmarks

6.1.1 StreamAdd

Performance results for StreamAdd are shown in Figure 7. The y-axis represents execution time, and the x-axis shows performance as a function of data set size. There are three curves. The Itanium2-icc8 curve represents the code executing on the Itanium-2 host. The one labeled PIM.compiled shows execution time of the same data set on 1 PIM, and is generated by the compiler. The one labeled PIM.tuned represents the hand-tuned version of the code. The fourth curve projects execution time on 8 PIMs. As there is no communication in this code, the PIM performance speeds up linearly with the number of PIMs.

The PIM floating-point performance for StreamAdd shows deterministic performance monotonically increasing as the problem size increases. The Itanium2 result shows performance that varies as a function of problem size. It is better for smaller problem sizes, but as the problem sizes get larger, the way in which the system allocates memory leads to worse performance. We observe in the figure that the single PIM execution time is comparable to that of the Itanium2 execution time. With eight PIMs running in the memory system, there is an 8x speedup over the single Itanium 2 processor.

6.1.2 Random Access

RandomAccess results are shown in Figure 8. This benchmark requires host and PIM collaboration at a fairly fine grain, and stressed the host-to-PIM interface. The process of writing from host to PIMs is complex. To send a 256-bit parcel to a PIM requires reordering bits and a stream of writes due to the spreading of data across multiple memory chips and bit interleaving by the Itanium-2 memory controller. We write a subset of the bits on each memory operation, and must rearrange the bits on arrival at the PIMs. While this works in our system, we envision a system where this sort of reorganization is not required, and we do not include the cost of this reorganization in our timings.

We measure performance of the host and PIM portions of the implementation on the hardware, and combine these measurements. On the PIM side, the code was instrumented and measured from the time a parcel is sensed through the update until the time it is waiting again for another parcel. Measurements are in terms of clock cycles. The performance of the PIM across timing measurements has proven to be deterministic and consistent. On the host side, we replaced the writes to the host parcel buffer on the PIMs with a conventional write. To force the write to occur, we flush the cache line in which the PIM payload resides just prior to the point where the parcel should be sent. We verified with a logic analyzer that this implementation had the appropriate number of memory operations.

In the performance measurements of Figure 8 the host sends parcels at the rate of 2.5M parcels per second (we discuss below how to increase this rate). A single PIM can process parcels at the rate of 1.2M parcels per second. In a system with two PIMs, the PIMs and host can process parcels at the same rate. Beyond this, the rate at which PIMs process parcels is much faster than the host's ability to produce parcels, and the host system bus becomes a performance bottleneck.

	Execution Time	Cycles	Instructions Per Clock
Itanium-2	5.5ms	4.9M	1.588
Single PIM (scalar)	113.0ms	16M	<1
Single PIM (superword, compiler-generated)	94.6ms	13M	<1, but includes SIMD ops
Single PIM (superword, hand tuned)	32.1ms	4M	<1, but includes SIMD ops

Table 4. Performance comparison for MI on TEST input.

	Execution Time	Cycles	Instructions Per Clock
Itanium-2	0.26 ms	233K	0.806
Single PIM (scalar)	1.11 ms	155K	<1
2 PIM parallel	1.65 ms	231K	<1

Table 5. Performance comparison for Graph Clustering on TEST input.

Using these results, we can project end-to-end performance as shown in Figure 8. The first bar is the measurement of the host-only implementation on the Itanium-2. The performance is 0.0044 giga-updates per second (GUPS), the unit of measure for this benchmark. If the Host+PIM code is used with just one PIM, the performance is comparable, 0.0045 GUPS. With a second PIM in the system, we can achieve a 2X speedup, achieving 0.01 GUPS.

The 8 PIM results rely on sending parcels to 8 PIMs in parallel. This is possible by exploiting what is in some sense a challenging aspect of the host-to-memory interface. A write to a particular PIM must be done 4 bytes at a time, with the remainder of the bytes ignored by the memory system. If the address space is arranged appropriately, the remainder of the bits can go to distinct PIMs, four bytes at a time. This would require a slightly different implementation, such that when the buckets are full for one PIM, an aggregate communication would be performed for all PIMs. This complex arrangement of bytes/addresses could be transparent to the user by implementing it in the boot and communication libraries. With such an implementation, we could achieve 0.038 GUPS. Further improvements are possible with PIMs running at a higher clock rate; a clock rate improvement of just 2X would make each PIM's bandwidth on par with the Itanium-2, leading to a speedup of 16X.

6.2 Mutual Information on a Single PIM

For the MI results, we first present numbers on the TEST data set of 520 nodes and 4768 links, which was derived by deleting nodes and links from a larger data set, and its purpose was to derive a test input small enough to be conveniently loaded onto a single PIM. It is roughly 500Kbytes. Performance results for the Itanium-2 alone and for a single PIM on the TEST input are shown in Table 4. The first column identifies whether the code was run on the Itanium-2 or the PIMs. The execution time is shown in the next column. Recall that the Itanium-2 clock is operating at 900 MHz while the PIM clock rate is 140MHz, and the Itanium-2 is 6-way VLIW while the PIM processor is a single-issue, in–order processor. Thus, to facilitate comparison between the two systems, the number of cycles is shown in the next column. There are three versions of code for the PIMs in Table 4. The first uses the scalar processor, and does not exploit the PIM's WideWord unit. The second is automatically generated from sequential code by the SUIF compiler frontend to perform SIMD operations in the WideWord unit. The third was a result of hand tuning the second version. There is a modest speedup of 1.2X in the Wide version of the code over scalar code. However, with minor tuning having to do with control flow optimizations as discussed in Section 2, there is a speedup of 3.5X over the scalar version of the code.

In comparing the Itanium-2 performance to the PIMs, the PIM code runs slower, but it is executing 18% fewer cycles than the Itanium-2. The performance difference is largely due to the Itanium-2's 6X faster clock rate. In addition, the Itanium-2 memory system is performing well for this relatively small data set. There is plenty of work to keep the processor busy, achieving an Instructions-Per-Clock of 1.588. This means that the Itanium-2 is retiring more than one instruction per cycle, while the single-issue, in-order PIM processor is retiring less than one instruction per cycle.

The TEST input was very small, so we were interested in how the MutualInformation kernel would perform on the larger data sets, . as shown in Table 6. The first column describes the data set, with TEST included for reference. The second column shows how many nodes and links are in the graph for each dataset, followed by size in bytes in the third column. Itanium-2 execution time for each data set is in the fourth column, and Itanium-2 IPC is shown in the fifth column. The sixth and seventh columns show L1 and L2 miss rates, respectively.

Overall, the table reveals that this kernel behaves similarly regardless of execution time or data set size. While there is some variation in IPC across the different data sets, it is generally high, ranging from 1.478 to 1.588. L2 miss rates are extremely low, less than one percent. L1 miss rates are somewhat high, but the L2 accesses comprise over 80% of memory references across all the different data set sizes.

Dataset	Nodes	Links	Size bytes	Exec. Time	Inst. Per Clock	L1 Miss Rate	L2 Miss Rate
TEST	520	4764	499KB	5.5ms	1.588	20.37%	0.51%
4027	987	3.368M	350MB	1.719 s	1.478	21.50%	0.66%
4028	9820	387668	40MB	332.4ms	1.540	22.52%	0.57%
4036	9960	384240	40MB	408.2ms	1.561	31.84%	0.48%

Table 6. Mutual Information Data Scaling on Itanium-2

Dataset	Nodes	Links	Pairs	Size (bytes)	Exec. Time	Inst. Per Clock	L1 Miss Rate	L2 Miss Rate
TEST	9820	23656	9704	273KB	2.59ms	0.806	19.6%	14.2%
4027	987	3.368M	769298	15MB	7.234s	0.489	15.7%	39.9%
4028	9820	387668	157928	3MB	219.7ms	0.653	31.5%	26.5%
4036	9960	384240	196544	4MB	592.5ms	0.464	31.8%	25.1%

Table 7. Graph Clustering Data Scaling on Itanium-2

In summary, this kernel performs well on the Itanium-2's memory system, and there is little room for improvement. The reduction in PIM cycles as compared to the Itanium-2 cycles is a result of optimizations for the PIM's WideWord unit.

6.3 Graph Clustering, Sequential and Parallel

For graph clustering, we present numbers on a TEST data set of 9820 nodes, 23656 links, and 9704 pairs. Note that this data is organized more compactly than for MI. It is represented by pairs of nodes, rather than by links. As in the previous example, this data set was derived by deleting nodes and links from a larger data set, and its purpose was to derive a test input small enough to be conveniently loaded onto a single PIM. It is roughly 273Kbytes. Performance results for the Itanium on the Itanium-2, for a single PIM running sequential code and 2 PIMs executing the parallel code on the TEST input are shown in Table 5. (A version of the code with SIMD instructions for the Wideword unit was generated by the compiler and validated in simulation, but not measured on the hardware.)

The columns are as in Table 4. For this code, the IPC on the Itanium-2 is much lower at 0.806. As a result, the scalar PIM code (without WideWord code) is only 4.26X slower than on the Itanium-2, and executes a third fewer cycles than the Itanium-2. This improvement in cycle count indicates that the memory behavior of this code on the Itanium-2 limits its performance.

We developed the parallel version of the algorithm as described in Section 3 and show preliminary performance results for two PIMs. The performance is slower than the scalar code and we suspect that the overhead of the parallelization is too high for this data set size and only 2 processors. Usually speedups can be improved by scaling up the data set size, but we are limited by the 1MByte storage on each PIM.

To investigate how data scaling impacts performance of the graph clustering benchmark, we performed measurements on the same three data sets as in Table 6, shown in Table 7. The data has been translated to use the compact pair representation. We see from Table 7 that performance of graph clustering is greatly affected by data set size. The data sets range from the 273KB of the TEST input, to 3 and 4MB for the intermediate sizes and 15MB for the large size. IPC is as low as 0.464. One interesting observation from this table is the difference in performance behavior of the 4028 and 4036 data sets. While they have a comparable number of nodes and links, the number of pairs is about 24% larger in the latter case, resulting in a 33% increase in data set size. This difference leads to a more than doubling of execution time, and a much lower IPC. While not shown here, the data set in 4036 has a significant increase in L3 misses which appears to be the cause of the performance difference.

Our results on graph clustering are more preliminary than the others, but these results demonstrate a potential for PIMs in graph clustering algorithms. Our scalar PIM code is not at all tuned, and yet there are fewer cycles than on the Itanium-2.

6.4 Scaling Analysis & Projected Performance

The previously described measurements of these kernels on the prototype hardware are very valuable, but as it is an academic hardware implementation that was developed on a short time schedule, it does not achieve performance levels that we could expect from a commercial-quality PIM implementation. Table 8 below summarizes the differences between the Itanium-2 and PIM processors. The key differences are that the Itanium-2 has a 6x faster clock rate, and it is a 6-way issue EPIC processor rather than the single-issue, in-order processor of the PIM chip. A further constraint on our PIM implementation is that the on-chip memory is just 1 Mbyte. For these reasons, in this section we evaluate the measurements of the previous section in a broader context, to account for the constraints of an academic project, consider architectural variations and look to the impact of future technology trends.

Future devices will have denser memories and multi-core processors, not higher clock speeds or more complex processors. Thus we can expect that production PIM chip processor speeds can be on par with IBM Cell (which is multi GHz in 90nm). In the

	Clock	CPU Info	Area	Transistor	Power
Itanium-2	900 MHZ	EPIC, 6-way	421mm^2	221M	~100W
1 PIM	140 MHZ	single-issue, in-order, pipelined	121mm^2	56.6M 55M (memory)	~1W
8 PIMs	140 MHZ	single-issue, in-order, pipelined	8 x 121mm^2	453M 440M (memory)	~8W

Table 8: Comparison of Itanium-2 and PIM processors.

	Speedup Raw 1-PIM	Clock Scaling (1) above	Data Scaling (2) above	2 PIMs	Projected 8 PIMs
Mutual Information	0.171X	1.225X	1.316X	~2.632X (projected)	~10.528X
Graph Clustering	0.234X	1.503X	2.611X	1.752X	unknown

Table 9: Projected performance accounting for clock and data scaling.

end, the best performance/watt may be achieved by slower clock speeds, and in fact, it may be best to have PIM processors operate no faster than the embedded DRAM frequency. Clock rate could up to 1 GHz, memory size could increase to 512 MB/chips and processor eDRAM cores could be 32 per chip by 2010.

Based on these projections, we scale the performance measurements from the previous section, and the result of this exercise is shown in Table 9 below. In the first column, we show the raw 1-PIM speedup over Itanium-2, which is in both cases less than one. The next column shows the impact of clock scaling, based on the following formula:

Clock Scaling = IT2 Cycles / PIM Cycles (1)

That is, we assume the PIM and Itanium-2 have the same clock rate. In the next column, we also try to account for the impact of scaling to larger and more representative data set sizes. We normalize the speedup by adjusting the IPC from the TEST input set up to the SCALE input set, using the following formula:

DataScaling=IT2 Cycles*(IPC_{test} /IPC_{scale}) / PIM Cycles (2)

In examining the results, since the graph clustering code started out closer to the Itanium-2 performance, it yields a 1.5X speedup in terms of reduction in cycles, as compared to 1.225X for Mutual Information. Further, since the IPC for the graph clustering code varies significantly with larger data sets, it shows an additional gain up to a 2.6X speedup as compared to a minor improvement for Mutual Information. We also considered parallel performance. The MI kernel is embarrassingly parallel as written, performing for the most part independent computation, and thus we project linear speedups for 2 and 8 PIMs. For graph clustering, the 2 PIM result, even after clock and data scaling, does not perform as well as the scalar version, as discussed in the previous section. Given the complexity of the parallel implementation, we did not attempt to project 8 PIM performance for this kernel.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented the first performance measurements on the Godiva hardware testbed, a PIM-based architecture implementation developed at USC/ISI over a 7 year period. An important result of this work was that we were able to demonstrate the effectiveness of PIMs at delivering memory bandwidth on the bandwidth benchmarks StreamAdd and RandomAccess. For both kernels, our PIMs were able to deliver comparable bandwidth to the Itanium-2 with just a single PIM, and 8X more bandwidth with 8 PIMs, in spite of the Itanium-2's far more powerful and 6X faster processor. Further, RandomAccess demonstrated the advantage of supporting sharing of data between host and PIM, a unique feature of our PIM architecture as compared to others. We expect these kernels would be at least 6 times faster on a single commercial PIM than a conventional platform. For the link discovery kernels, the results for the graph clustering kernel are far more promising for PIMs than the mutual information kernel. As the graph gets larger, graph clustering exhibits poor memory-system behavior on the Itanium-2. As a result, the number of cycles on the PIM is a third lower than on the Itanium-2 even with our 273KB TEST input. Our analysis suggests that much larger gains would be obtained for memory sizes in a commercial PIM implementation. The real open question is how to perform graph clustering in parallel and manage the communication overhead. Our parallel PIM results are preliminary, and did not show performance improvement for the small TEST data set on 2 PIMs. We augmented these parallel results with the simulations described in Section 4. We found potential for performance gains in graph clustering with PIMs, but also some load imbalance, and sensitivity to data set features.

Looking to the future, there is a need to develop parallel algorithms for link discovery and identify the appropriate algorithms and programming models to manage communication in graph data structures.

8. ACKNOWLEDGEMENT

This material is based on research sponsored by AFRL and NSA under agreement number FA8750-04-1-0265. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL and NSA or the U.S. Government.

9. REFERENCES

[1] J. Adibi, H. Chalupsky, E. Melz and A. Valente. The KOJAK Group Finder: Connecting the Dots via Integrated Knowledge-Based and Statistical Reasoning. Innovative Applications of Artificial Intelligence Conf., IAAI 2004.

[2] D. Burger, J. Goodman and A. Kagi, "Memory Bandwidth Limitations of Future Microprocessors", Proc. of the International Symposium on Computer Architecture, May, 1996.

[3] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, G. Daglikoca, "The Architecture of the DIVA Processing-In-Memory Chip," In Proc. of the International Conference on Supercomputing, June, 2002.

[4] D. Elliott et al., "Computational RAM: Implementing Processors in Memory", *IEEE Design and Test of Computers*, January - March, 1999.

[5] M. Gokhale and B. Holmes and K. Iobst, "Processing In Memory: the Terasys Massively Parallel PIM Array", *IEEE Computer*, April, 1995.

[6] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, W. Athas, A. Srivastava, J. Shin, J. Park, "Mapping Irregular Computations to DIVA, a Data-Intensive Architecture," In Proc. of SC99, Nov. 1999.

[7] P. Kogge, "The EXECUBE Approach to Massively Parallel Processing", Proc. of the International Conference on Parallel Processing", Aug, 1994.

[8] S. Lin and H. Chalupsky. Unsupervised Link Discovery in Multi-relational Data via Rarity Analysis. In Proc. of the 3rd IEEE International Conference on Data Mining (ICDM '03).

[9] J. Shin, J. Chame and M. W. Hall, "Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures," Journal of Instruction-Level Parallelism, 2003.

[10] J. Shin, M. Hall and J. Chame, ``Superword-Level Parallelism in the Presence of Control Flow," In Proc. of tCode Generation and Optimization (CGO), March, 2005.

[11] H. Chalupsky and R.M. MacGregor. STELLA - a Lisp-like language for symbolic programming with delivery in Common Lisp, C++ and Java. In Proc. of the Lisp User Group Meeting, Berkeley, CA, 1999. Franz Inc.

Appendix

```
algorithm  ParallelInOutRatio ( {seed members}, numNewMembers) : Group
{
Group = {seed members}
LocalOutList = {nodes in local PIM memory connected to Group}
localBestRatio = 0.0
while (numNewMembers > 0) {
     foreach node1 in LocalOutList {
            newInLinks = newOutLinks = 0
          foreach pair (node1, node2) in PairTable {
               if (node2 is in Group)
                 newInLinks += •( links of pair (node1, node2) )
               else
                 newOutLinks += •( links of pair (node1, node2) )
               InOutRatio = (inlinks+newInLinks)/(outLinks – newInLinks + newOutLinks);
              if (InOutRatio > localBestRatio) {
                  localBestRatio = InOutRatio;
                  localBestNode = node1;
              }
          }
     }
  ParallelReduction(localBestNode, localBestRatio, newInLinks, newOutLinks,
                       globalBestNode, inLinks, outLinks)
     Group = Group U {globalBestNode}
     update  LocalOutList
     numNewMembers = numNewMembers  - 1
}
return Group
}
```