Design of an In-Memory Database Engine Using Intel Xeon Phi Coprocessors

Michael Scherger
Department of Computer Science
Texas Christian University
Fort Worth, TX, USA
Email: m.scherger@tcu.edu

Abstract – *(PDPTA'14) This research presents the design and initial implementation of a database engine using an Intel Xeon Phi co-processor. The many integrated cores (MIC) of the Xeon Phi make this hardware accelerator a natural computing platform for an in-memory database engine or server. The database tables reside in the memory space of the MIC thus supporting fast in-memory database applications. This achieved by developing a coalescing parallel memory manager to allocate parallel variables in the same manner that fields are created in a table using a SQL CREATE TABLE command. The SQL interface was created using a database driver toolkit that provides an interface to the Xeon Phi server and client application. Once the basic framework was established, the algorithms for SQL select, insert, update, delete, and join were created to manipulate database information in the memory of the Xeon Phi.*

Keywords: parallel databases, parallel hardware accelerators, special purpose architectures

1 Introduction

Massively data parallel computers and the SIMD model of parallel computation can be a natural model of parallel computation to consider for massively parallel database servers. Since the cores are extremely close to the parallel memory, fast parallel memory searching make it a natural platform for data parallel computing intensive applications. As described in Potter in [9] and [10], data parallel models of computation such as the SIMD, ASC, or SITDAC model conform to the concept of a parallel database server since the data can logically and physically partitioned similar to the data organization of a database table or spreadsheet [2][8][9][10][11] and [12].

This research paper discusses the initial design of an in-memory database server using a Intel Xeon Phi co-processors. This research will discuss the design considerations and challenges for a database server and SQL engine that interfaces with the memory of a hardware accelerated data parallel computer. This system design can promote the use of massively parallel computers as database servers for use in embedded database systems, real-time database systems, and fast parallel associative search engines.

Database management systems (DBMS) provide a structured mechanism for storing, organizing, and retrieving data in a way that is consistent with the database's format [14]. System software will allow data storage and access to a database without the user's knowledge about the internal data representation either in persistent storage or in the computer's memory. A DBMS usually has but is not limited to the following components [14]:

- Processors and main memory – the hardware of the DBMS for data selection and computation
- Secondary storage – disks for data persistence and offline storage
- Database manager – software for creating and maintaining databases, tables, fields, and relations
- Utilities – software for database maintenance, data integrity and security, and database repair
- Application development tools – software for database application development integrated into the DBMS
- Report writers – software modules for presentations and reports based on tables and queries from database information
- Design aids – software to assist in the design of databases, tables, fields, indexes, and relationships

The organization of this research paper is as follows. Section 2 will use the tracking and correlation problem in air traffic control as a motivating example. Section 3 present an overview of the Intel Xeon Phi co-processor and system software. Section 4 will present the hardware and physical design of the database server including the mapping of table records into the memory of the parallel computer. Section 5 will discuss the techniques of sequential and parallel database query processing. Section 6 will present the system software design of the parallel SQL engine and the algorithms for the basic database server operations. Section 7 will discuss the conclusions and future research.

2 Example Application: ATC Tracking and Correlation

Consider a real world and real time application of air traffic control. The following example is an extremely simplified version of the air-traffic tracking and control problem, but provides enough detail to illustrate the system software design for a parallel database server and SQL interface [11]. Some of the basic tasks of air traffic control are:

- Tracking and correlation – The radar will generate reports of flights of returns that must be correlated to tracks of flights currently in the system.
- Conflict detection – The computer system must then determine if there are any tracks/flights that will conflict/collide with a time look ahead of a predetermined number of minutes or miles.
- Flight plan update – Based on the tracking and correlation information and combining it with the conflict detection, the flight plan information will be updated.

There are many other important tasks in air traffic control, but this is an example of a real-time processing problem [6],[7], and [8]. There are also hard deadlines for computation imposed on the above tasks. They must be completed prior to the next hard deadline in this real time system.

The flight plans and tracks from radar can be stored in a simplified tabular format (flat table) in a database table similar to illustration in Figure 1.

AID	LAT	LON	ALT	HDG	AS
CO56	40.55	81.26	270	0	450
AA123	39.9	84.31	60	225	275
UA722	41.24	81.51	150	0	525
AA1223	40.0	82.53	290	315	305
…	…	…	…	…	…
CA2341	39.54	84.13	70	90	330

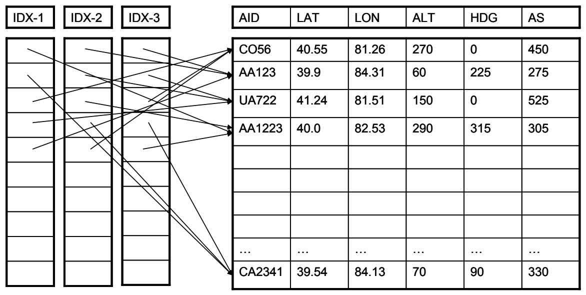
Figure 1: Sample database to store flight plan information.

The ATC system software will/may have to perform the following operations when receiving a new set of track information.

- Insert a new flight into the table. As aircraft enter the airspace, they need to be stored into the flight table. This involves searching for an open/free record in the table and then copying the flight information into that newly created record.
- Deleting a flight from the table. As aircraft leave the airspace, they need to be deleted from the flight table. This involves searching for the record of the flight to delete and marking that the record is inactive.
- Selecting a flight from the table. Selection involves identifying one or more flights for further processing. The selection must scan the data in the fields for this table and then return that information back for further processing.
- Updating the flight information. Updating a flight begins with a search followed by a copy of new information into the selected record from the track information.

Each of these frequent operations (insert, select, update, delete) requires some type of a parallel memory search. In the case of insert, the search operation is for an open record in the table. In the case of the select, update, and delete operations, the search required is based on the data stored in the records of the table. This is a contextual search or associative based search.

The most common method to improve search performance in a database server is to use index tables. This is illustrated in the Figure 2.



AID	LAT	LON	ALT	HDG	AS
CO56	40.55	81.26	270	0	450
AA123	39.9	84.31	60	225	275
UA722	41.24	81.51	150	0	525
AA1223	40.0	82.53	290	315	305
…	…	…	…	…	…
CA2341	39.54	84.13	70	90	330

Figure 2: Flight table illustrating the use of index tables to improve performance.

An index table stores the record indexes based on some ordering criteria or sorting functions. In Figure 2, an index table may store pointers to the indices for the aircraft sorted by aircraft id. Another index may store pointers to indices for the aircraft based on altitude. Finally another index may store pointers to the indices for the aircraft based on airspeed.

In theory, a database table can have one or more indices for each field. However, this dramatically reduces the performance of the insert, update, and delete operations at the benefit of doing fast searches [1][3][4][5]. As new records (flights) are entered into the table, the index tables need to be updated and maintain their sorted order. The constant resorting of each index table becomes increasingly computational demanding. The same is true for the delete and update operations when the flight information changes. The performance degradation is further amplified when multiple index tables must change.

3 Overview of the Intel Xeon Phi

The Intel Xeon Phi co-processors have 60 in-order Intel MIC architecture cores running at 1 GHz. The Intel MIC architecture is based on the x86 ISA, extended with 64-bit addressing and 512-bit wide SIMD vector instructions and registers. Each core supports 4 hardware threads. In addition to the cores, there are multiple on-die memory controllers and other components.

As shown in Figure 3, each core has a newly designed Vector Processing Unit (VPU). Each VPU unit contains 32 512-bit vector registers. To support the new vector processing model, a new 512-bit SIMD ISA was introduced. The VPU is a key feature of the Intel MIC architecture based cores. Fully utilizing the vector unit is critical the best performance. The Intel MIC architecture cores do not support other SIMD ISA's such as MMX, SSE, or AVX.

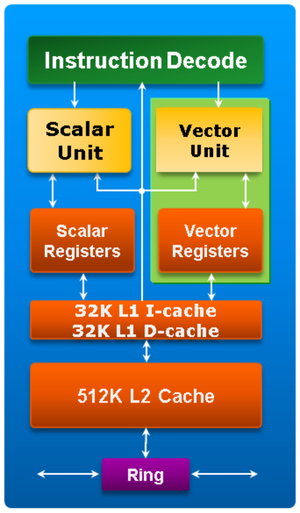


Figure 3: Intel Xeon Phi MIC core block diagram.

Each core has a 32KB L1 data cache, a 32KB L1 instruction cache, and a 512KB L2 cache. As shown in Figure 4, The L2 caches of all cores are interconnected with each other and the memory controllers via a bidirectional ring bus, that effectively creates a shared last-level cache of up to 32 MB. The design of each core includes a short in-order pipeline. There is no latency in executing scalar operations and low latency in executing vector operations. Since the in-order pipeline is short, the overhead for branch misprediction is low.

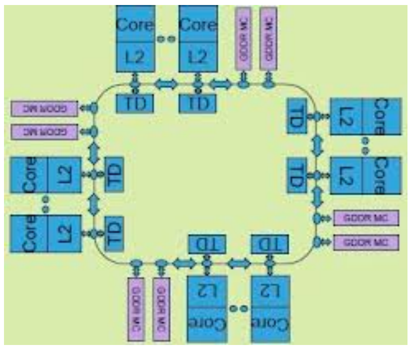


Figure 4: Logical MIC core layout and ring communication bus.

4 Database Engine Hardware Design and Architecture

There are a few assumptions regarding the design of the parallel database server [11].

1. The database, tables, and records in the parallel database server are memory resident. Storage is completely volatile and there is no persistent storage in the cells or array memory implemented in this design. For real-time computation, storing data and record information in in secondary storage is costly in terms of access time. Having the data reside in memory, close to the processing elements is more conducive for real-time applications.

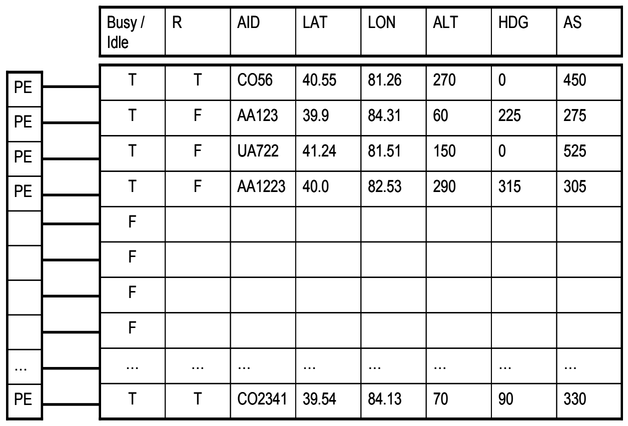
2. The data parallel memory map is similar to the field layout planned of a database table. If TABLE_A has fields F_1, F_2, F_3 created in that order, then the parallel memory map will have parallel variables F_1[$], F_2[$], and F_3[$] located in lower to higher parallel memory addresses.

3. The number of actual processing elements is fixed during the execution of the parallel database server. This is not an unrealistic assumption since the Intel Xeon Phi has a fixed number of cores (or hyper-thread processors).

4. The amount of memory per processing element is fixed. Again, the memory in the Intel Xeon Phi separate "parallel memory space" than the memory of the host computer. Albeit the parallel memory space is often smaller than the host memory, for most database applications, the amount of parallel storage is adequate.

Since this model is using massively parallel search and responder processing as a model of data parallelism, database index tables are no longer required. Each database field (column) can be searched for the desired value in constant time. Data parallelism can also support efficient software for associative searches.

The cores, or processing elements (PE) of the Intel Xeon Phi will be used to assist in the basic database operations and searching. This is illustrated in Figure 5. In this figure, the database table is superimposed on the memory and processing elements of a SIMD computer. Two additional fields have been prefixed to the table: a busy-idle flag to indicate if the PE or record is active and a responder flag used for search operations. Using this approach, each individual record is located in the memory of a PE. Using massive parallel searching, processing elements can scan their individual memories and set the responder flag or turn their busy-idle flag on or off.



	Busy / Idle	R	AID	LAT	LON	ALT	HDG	AS
PE	T	T	CO56	40.55	81.26	270	0	450
PE	T	F	AA123	39.9	84.31	60	225	275
PE	T	F	UA722	41.24	81.51	150	0	525
PE	T	F	AA1223	40.0	82.53	290	315	305
	F							
	F							
	F							
	F							
...	...	...	...	...	...	...	...	...
PE	T	T	CO2341	39.54	84.13	70	90	330

Figure 5: Flight table superimposed onto the PEs and memory of a SIMD computer.

Database tables are dynamic objects; there is typically no *a priori* knowledge of the number of table records. If the number of records in a table exceeds the number of physical PEs in the system (parallel memory overflow) the database server will use a cyclical data placement strategy when inserting new records. This is a form of virtual parallelism that is maintained by the parallel database server and not the operating system. This cyclical placement will manage multiple tables with multiple folds in an interleaved fashion as determined by the amount of data in the tables. For example, in Figure 6, Table A utilizes only 4 PEs, while the number of records in Table B has exceeded the number of PEs resulting in multiple folds.

An insert operation for Table A will add a new record into the area occupied by fold 1 for table A. For table B, the next insert operation will be in fold 2. If enough records are added to Table A to exceed the capacity of fold 1, a new fold will be created in the free memory space to the right of fold 2 of Table B [11].

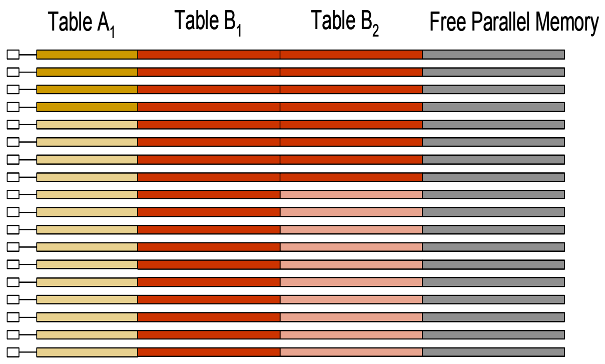


Figure 6: Multiple database tables, table folds, and unused parallel memory.folds, and unused parallel memory.

By necessity, data memory management becomes the responsibility of the parallel database server instead of the parallel compiler or other system software [11]. A coalescing parallel memory manager (CPMM) was developed to keep track of table, field and fold addresses. Figure 7 illustrates some of the administrative data structures that must be maintained for folded tables.

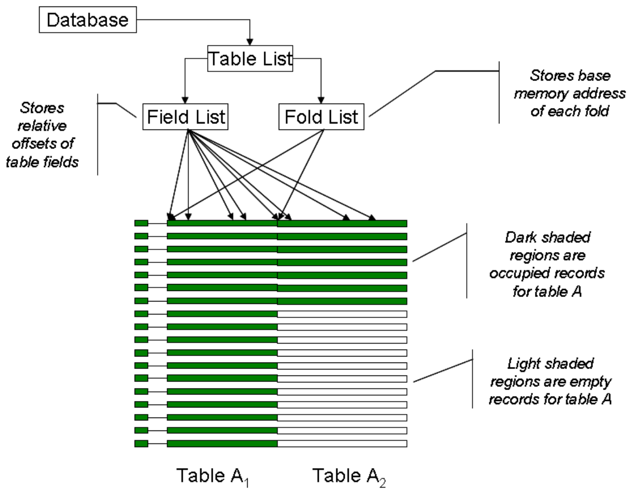

Database
Table List
Stores relative offsets of table fields
Field List
Fold List
Stores base memory address of each fold
Dark shaded regions are occupied records for table A
Light shaded regions are empty records for table A
Table A_1
Table A_2

Figure 7: Data structures for logical database tables, folds, records, and fields.

The parallel database server will maintain the controlling data structures to manage the database, tables, folds, records, and column addresses. These data structures reside in the sequential memory of the control unit or front-end computer. Dark shaded regions in this figure represent active records in the table. Note that there are two folds for this table and the field list is replicated for each fold. The table fold is an absolute parallel memory address while the field address is a relative parallel memory address. By adding the two memory addresses together, the physical memory address for a database field within a fold can be determined. The parallel memory manager also created extra hidden table fields used for basic database operations (described in a later section). These hidden table fields included several responder bits, a busy/idle flag, and timestamp fields for record insertion, selection, and update.

5 Sequential and Parallel Query Processing

Query processing refers to the range of activities involved with extracting data from a database. The activities include translation of queries in high-level database languages into expressions that can be used at the physical (storage) level. The fundamental steps a database server must perform when processing a database query appear in Figure 8:

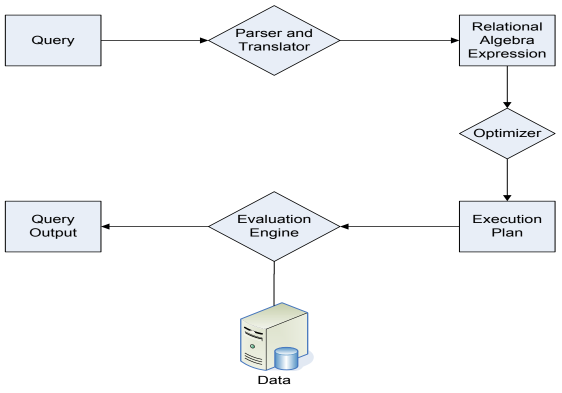


Figure 8: Major functional components of database query processing.

Before query processing can begin, the system must translate the query into a usable form. A language such as SQL is appropriate for software application development, but is not amenable to be the system's internal representation of a query. As shown in Figure 3, the first step the system must take in query processing is to translate a given query into its internal form. This translation process is similar to the work performed by the parser of a compiler. In generating the internal form of the query, the parser checks the syntax of the user's query and verifies that the query names appear in the database. The system then constructs a parse tree representation of the query, which it then translates into a relational algebra expression.

The sequence of steps in query processing is representative. Not all databases exactly follow these steps. However, the concepts that have been described form the basis of query processing in databases.

5.1 Sequential Query Processing Algorithms

There are several sequential query processing algorithms defined in the literature [14] and [15]. Each algorithm has a particular use when the query processing evaluation takes place.

The most relevant query processing algorithm related to this research is the A1 – Linear Search algorithm, which is now described. In a linear search, the system scans each file block and tests all records to see whether they satisfy the selection condition. An initial seek is required to access the first block of the file. The cost of linear search, in terms of number of disk operations, is one seek plus b_r block transfers, where b_r denotes the number of blocks in the file. Equivalently, the time cost is $t_S + b_r * t_T$.

Although the A1 – Linear Search algorithm may be slower on sequential computers than other algorithms for implementing selection and other query processing tasks, it is the most natural algorithm in terms of conversion to a massively parallel equivalent since the linear search on a parallel variable can be accomplished in constant time on SIMD (or MASC) computers assuming the database can be held entirely in memory.

6 System Software Design and Architecture

Now that the basic parallel memory management issues have been addressed, the system software design of the database server is described.

A client application will use the database driver manager to interface with the client database driver. The client database driver communicates with the SQL Engine. The SQL Engine will call process these instructions and then call the appropriate parallel database server, where there will be a corresponding function call to perform an operation in the memory of the parallel computer. The parallel database server will then receive the request from the database driver and control the databases, tables, records, and columns in the parallel memory.

6.1 Parallel SQL Insert Algorithm

The task of the parallel SQL insert operation is to insert new data into a free record located anywhere in the table in any fold. An example of the SQL INSERT statement is the following:

```
INSERT INTO FLIGHTS( AID, LAT, LON, ALT, AS )
VALUES( 'CO128', 43.39, 83.67, 190, 450 )
```

This insert statement will insert a new record into the FLIGHTS table (reference the database table in Figure 4) and assign the respective values to the AID, LAT, LON, ALT, and AS fields.

For inserting a record into a parallel memory space, the basic parallel insert algorithm is the following:

```
Algorithm Par_SQL_Insert( RecordData )

  open_record_found = FALSE

  For each table fold

    Perform associative search on the
    table’s BI field where BI field is
    false (i.e. record is empty – there
    may be multiple records returned)

    if ( idle records found )
       select one record;
       BI = TRUE
       open_record_found = TRUE
       break

    // no open record is found
    // in any fold
    if ( open_record_found == FALSE )
      create a new fold
      select first record in the new fold
      BI = TRUE
      break

  Copy the data into the parallel memory record
  Return success or failure
```

Figure 8: Parallel SQL insert algorithm.

The algorithm Figure 8 begins by searching for an open or idle record in each of the table folds in turn. If idle records are found, then PE identification number and the fold select one record and field addresses are used to copy the data into the parallel memory record. If no idle record is found, then the parallel memory manager must create a new fold. This can be accomplished by allocating space from the unused space in parallel memory the same width as previous folds and recording the new base address in sequential memory. Since a new fold is created, the parallel memory manager can select any PE for the insertion; e.g. the first PE (lowest PE id number) can be used. The basic parallel search can be done in *O(1)* time. However, since each table fold may have to be scanned, the running time is *O(#folds)* which is typically small and normally still *O(1)* since the number of folds normally constant and not a function of higher complexity.

6.2 Parallel SQL Delete Algorithm

The task of the parallel SQL delete operation is to delete records according to some searching or selection criteria. An example of the SQL DELETE statement is the following:

```
DELETE FROM FLIGHTS
WHERE AID = ‘NW 545’     /* delete criteria */
```

This delete statement will delete all records where the AID (aircraft ID) is ‘NW 545’. For deleting a record from the parallel memory space, the parallel delete algorithm is the following:

```
Algorithm Parallel_SQL_Delete( DeleteCriteria )
returns Boolean

  For each table fold

    Perform associative search where the Delete
    Criteria is TRUE and set responders
    appropriately

    If (the responder is TRUE)
       Reset the Busy-Idle flag

    If all records in the current fold are idle
       CPMM marks the current fold as free

  Return success or failure
```

Figure 9: Parallel SQL delete algorithm.

The algorithm in Figure 9 begins by looping through each table fold and having each cell evaluate the appropriate fields as specified in the delete criteria clause. For those cells where the delete criteria clause is True, the responder busy-idle flag is reset. If all the records in a given fold have their busy-idle flag reset, the coalescing parallel memory manager (CPMM) marks that fold as completely unused and returns it to the free pool of parallel memory. The basic parallel delete can be done in *O(1)* time. However, since each table fold will have to be scanned, the running time is *O(#folds)*.

7 Conclusions and Future Work

This research paper has presented an initial design of an in-memory database engine utilizing Intel Xeon Phi co-processors. Also presented was the system software design and interface for sequential programs and applications to interface with the server. This was achieved by designing and developing the set of algorithms for common database operations that would support the functionality of a parallel database server. The SQL operations presented include insert and delete and execute in O(#folds) steps. The update and selection operations are similar. The design of a coalescing parallel memory manager was also developed to manage large tables and virtual parallelism.

An area of future research explores how the parallel database handles virtual parallelism. The present design uses a coalescing parallel memory manager to control the table folds in the memory of the parallel computer. This parallel memory manager is a built in component of the parallel database server because the Xeon Phi environment assumed that the number of processing elements was fixed at runtime and could not change.

Another area of future research could explore how the tables, records, and fields are physically mapped to the memory of the parallel computer. Presently, the parallel memory map as shown in Figures 6 and 7 indicate that the parallel variables are allocated for all processing elements in a given fold regardless of the number of records actually storing information. This leads to a waste of processing elements for tables with only a few records.

8 References

[1] Babb, E, "Implementing a Relational Database by Means of Specialized Hardware", *ACM Transactions on Database Systems*, Vol., 4, No. 1, 1979, pp. 1-29.

[2] Batcher, Kenneth, "The STARAN Series E, *Proceedings of the International Conference on Parallel Processing*, 1977, pp. 140-143.

[3] Berra, P. Bruce, "Some Problems in Associative Processor Applications to Database Management", *National Computer Conference and Exposition*, Vol. 43, 1974, pp. 1-5.

[4] DeFiore, Casper R, P. Bruce Berra, "A Quantitative Analysis of the Utilization of Associative Memories in Data Management", *IEEE Transactions on Computers*, Vol. c-23, No. 2, February, 1974, pp. 121-132.

[5] Hsiao, D., and M. J. Menon, "Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Growth (Part 1)", *Technical Report, OSU-CISRC-TR-81-7*, The Ohio State University, Columbus, Ohio, July, 1981.

[6] Jin, Mingxian, Johnnie Baker, and Kenneth Batcher, "Timings for Associative Operations on the MASC Model", *Proc. of the 15th International Parallel and Distributed Processing Symposium (Workshop in Massively Parallel Processing)*, abstract on page 193, full text on CDROM, April 2001.

[7] Lockheed Martin Company - formerly Loral Defense Systems, *ASPRO-VME Parallel/Associative Computer: Technical Overview*, Oct. 1993.

[8] Meilander, Will, Johnnie Baker, and Mingxian Jin, "Importance of SIMD Computation Reconsidered", *Proc. of the 17th International Parallel and Distributed Processing Symposium (Workshop on Massively Parallel Processing)*, abstract on page 266, full text on CDROM, April 2003.

[9] Potter, Jerry L., *Associative Computing: A Programming Paradigm for Massively Parallel Computers*, Plennum Press, New York, NY, 1992.

[10] Potter, Jerry, Johnnie Baker, Stephen Scott, Arvind Bansal, Chokchai Leangsuksun, and Chandra Asthagiri, "ASC: An Associative Computing Paradigm", *Computer*, Nov. 1994, pp. 19-25.

[11] Scherger, Michael, "An Object Model Framework, Runtime Environment Support, and Database System Software for a Multiple Instruction Stream Associative Model of Parallel Computation", *PhD Dissertation*, Department of Computer Science, Kent State University, Kent, Ohio 2005.

[12] Michael Scherger, Johnnie Baker, and Jerry Potter, "Multiple Instruction Stream Control for an Associative Model of Parallel Computation", *Proc. of the 16th International Parallel and Distributed Processing Symposium*, abstract on page 266, full text on CDROM, April 2003.

[13] Scherger, Michael, Johnnie Baker, and Jerry Potter, "An Object Oriented Framework for and Associative Model of Parallel Computation", *Proc. of the 16th International Parallel and Distributed Processing Symposium*, abstract on page 166, full text on CDROM, April 2003.

[14] Scherger, Michael, "On the Design of a Massively Parallel Database Server for In-Memory and Real Time Database Applications", Proceedings of the Int'l Conf. on Parallel and Distributed Processing Techniques (PDPTA), Las Vegas, June, 2007.

[15] Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan, *Database System Concepts, 4th ed.*, McGraw-Hill, Boston, MA, 2002.

[16] Su, Stanley Y. W., *Database Computers: Principles Architectures and Technologies*, McGraw-Hill, New York, NY, 1988.

[17] Intel Corporation, Intel Xeon Phi Coprocessor Developer's Quick Start Guide, Version 1.7, 2013.

[18] Intel Corporation, Intel Xeon Phi Coprocessor Architecture, 2013.

[19] Intel Corporation, Intel Xeon Phi Coprocessor System Software Developer's Guide, 2013.