# A Case For Staged Database Paradigm Using CSP

Pranav Vaidya and Jaehwan John Lee
Department of Electrical and Computer Engineering
Purdue School of Engineering and Technology
Indiana University-Purdue University Indianapolis

**Abstract**

As millions and possibly billions of transistors become available on a single chip, most future processor architectures will be multicore processors with many concurrent execution units. Such multicore processor architectures present both novel challenges as well as opportunities for database system design. Recent research indicates that pipelined relational algorithms and staged databases might perform better than traditional databases on such multicore processor architectures. In this paper, we present a pipelined, partitioned hash-join algorithm developed using C++CSP – a threading library based on the process calculus of communicating sequential processes. Previous research done by Garcia-Korth in their paper titled "Pipelined Hash-Join on Multithreaded Architectures" has shown that the size of buffers is crucial to the performance of a pipelined hash-join algorithm. In this paper, we corroborate their result and show that deepening the buffers generally improves performance till the buffer size reaches a certain size, beyond which any increase in buffer size starts degrading performance. In our case, the best case performance is obtained when the total buffer size is approximately equal to the size of the L2 cache. We hypothesize that the maximum performance of our pipelined hash-join algorithm is obtained when the intermediate relations can fit entirely into the L2 cache. Our research also shows that the performance of the pipelined algorithm implemented with C++CSP threading library is comparable (within an order of magnitude), if not better, than the performance of a pipelined database algorithm implemented using Pthread threading library. Furthermore, our approach may be used to implement other pipelined algorithms in relational databases using a threading library based on the formal method of communicating sequential processes.

**Index Terms**

Pipelined Hash-Join Algorithm, Pipelined Database, Staged Database, Communicating Seq. Processes

# I. Introduction

Recent technology trends indicate that multicore processors are increasingly becoming the most common computing architecture. Software systems such as databases that have been shown to perform poorly on modern computing systems should be redesigned to optimally execute on modern multicore processors [1], [2]. Traditionally, database systems have followed a more conventional master-worker threading model. With the advent of multicore processors, research has shown that significant speedups can be obtained from pipelined relational algorithms engineered into staged databases [4]. Staged database systems exploit not only inter-query parallelism but also intra-query parallelism. In staged databases (as shown in Figure 1), the queries are broken down across different stages, which are connected by buffers. The stages can be further broken down into multiple threads each representing a single relational operator. As opposed to previous approaches, our goal in this series of research is not to compare piplined database algorithms with traditional database algorithms but to:

- Analyze and explore the formal processing model of Communicating Sequential Processes (CSP) [7] for future implementation of various pipelined database algorithms.
- Compare and contrast the performance of pipelined database algorithms implemented using CSP with implementations of pipelined database algorithms in the published literature.

Due to the time limitations at this step, we further limit the scope of our research to a specific implementation of pipelined-hash join. We compare the performance of our pipelined hash-join algorithm with the implementation of the pipelined hash-join algorithm done by Garcia-Korth [4]. Furthermore, we corroborate their result by showing that deepening the buffers generally improves performance till the buffer size reaches a certain size, beyond which any increase in buffer size starts degrading performance.

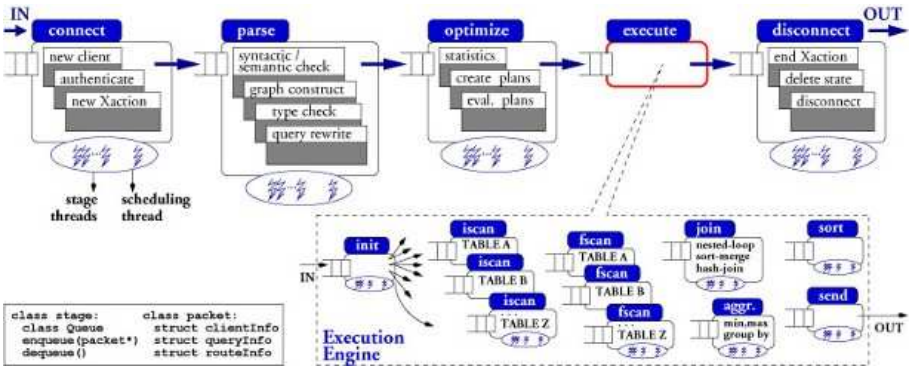



Figure 1: Pipeline stages in a staged database (courtesy of [2])

In our case, the best case performance is obtained when the total buffer size is approximately equal to the size of the L2 cache. We hypothesize that the maximum performance of our pipelined hash-join algorithm is obtained when the intermediate relations can fit entirely into the L2 cache.

Our results indicate that the performance of our pipelined implementation is comparable (within an order of magnitude), if not better, than the performance of the pipelined hash-join implementation by Garcia-Korth [4]. This shows that the C++CSP library based on the formal method of communicating sequential processes can be a viable alternative for developing pipelined database algorithms.

## II. Related Work

Pipelined database algorithms and staged databases have been explored previously in [3], [4], [6]. Software prefetching in pipelined hash-join was explored in [3] using simulation. This research was then validated on real hardware [4], [6]. Zhou *et al.* [6] examined a single hash-join operation on a Simultaneous Multi-Threaded (SMT) processor. Garcia-Korth [4] considered pipelined hash-join using Uniformly-Heterogeneous threading Model (UHM). They evaluated the performance of the pipelined hash-join algorithm using a novel buffer management system and the Pthread threading library. This unique buffer management system sets up processes in producer-consumer pairs, which are scheduled according to the availability of data to be processed. This technical report differs from the above approaches in that we use a thread library based on Communicating Sequential Processes (CSP) [9], [10]. CSP is a formal process calculus, developed by C. A. R. Hoare [7], which describes the interaction between concurrent processes. This process model is still being actively explored in research to overcome the limitations of the traditional thread model [8]. Researchers have argued that traditional threading models such as Pthreads do not inherently display key properties of sequential computation namely predictability and determinism. Therefore, it becomes the job of the programmer to remove this inherent non-determinism and bring about predictable computation. Hence, there is a strong case made for concurrent coordination languages based on sound, composable formalisms such as CSP to be used as threading models [8]. In this technical report, we show how pipelined hash-join can be implemented using one such formal coordination language called C++CSP – a CSP based C++ threading library.

## III. Problem Definition

In this technical report, we choose to examine the implementation of a pipeline of two hash-joins and three selection operators. Figure 2 shows the pipeline of relational operators that correspond to *Query1*. In this figure, operators $O_1, O_2$ and $O_3$ are the range-select operators used to perform pre-selection on

the relations A, B and C. The operators $O_4$ and $O_5$ are the hash-join operators. Our hash-join operator is similar to the hash-join operator implemented in [21]. Operator $O_6$ is the final post projection operator used to produce query results.

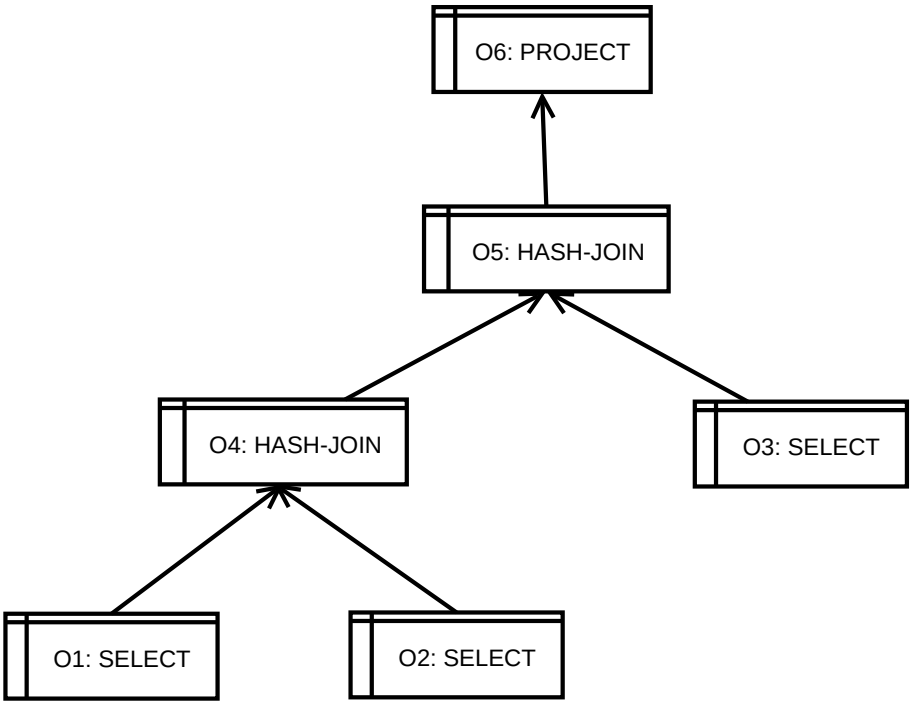



Figure 2: Pipeline of Hash-Joins (Bottom to up pipeline)

Our pipeline can also be easily extended to a query pipeline consisting of *n*-joins and *n*-selection operators. For our work, we compare the performance of this query pipeline implemented by us with the pipeline implemented by Garcia-Korth [4]. Furthermore, similar to the work done by Garcia-Korth, we also examine the performance of our threading model as a function of the size of the buffers used to share data among the threads.

**Query 1** SQL query corresponding to pipelined hash-join implementation

```
SELECT A.OID, B.OID, C.OID
FROM  A, B, C
WHERE A.bkey = B.bkey
AND B.ckey = C.ckey
AND A.bkey >= MIN_VAL_1 and A.bkey <= MAX_VAL_1
AND B.ckey >= MIN_VAL_2 and B.ckey <= MAX_VAL_2;
```

Additionally, our algorithm makes the following assumptions:

- All input, intermediate/temporary and final output relations can be held in the main memory. This also includes the hash-tables that are constructed as part of our hash-join algorithm.
- As a corollary to the above assumption, we also assume that our join will be performing an in-memory join with no significant disk or I/O traffic. This is done to ensure that our results are applicable to the modeling of main memory databases and modern computing hardware that consists of large cache and main memory.

## IV. CSP: Threading and Buffer Management System

### A. Overview of CSP and C++CSP Multithreading Library

This section provides a brief overview of the mathematical terminology behind CSP and indicates corresponding constructs available in C++CSP library to help readers better understand the following sections.

*1) CSP and C++CSP:* Nearly three decades ago, C. A. R. Hoare introduced Communicating Sequential Processes (CSP) [7], a programming model for dealing with parallelism and concurrency that has a strong mathematical basis. CSP avoids most of the fundamental problems inherent in concurrent design (such as synchronization, deadlocks, livelocks, racing and sharing of resources) without the need of any special mathematical knowledge. Although yet no mainstream operating systems or programming languages have built-in CSP mechanisms, extensions such as CCSP [16], JCSP [17], CTJ [18] and C++CSP [9], [10] have been suggested for language kernels of C, Java as well as C++. Out of these, C++CSP is a highly promising library developed in C++. C++CSP runs in native mode in either Windows or Linux and is highly portable across computing platforms that support these operating systems. Typically, CSP kernels are evaluated using the Commstime benchmark [19]. Results from this benchmark indicate that C++CSP is only slower than its OCCAM counterpart [19] by about 4X. However, it is up to 40-50X faster than the corresponding Java implementation in JCSP [17].

*2) Terminology and corresponding C++CSP Constructs:* In CSP, concurrent activities are modeled as processes that communicate and coordinate their activities using channels. These channels represent mediums through which messages are sent and received.

- Processes: Processes represent concurrent modules that do not share variables but communicate by sending and receiving messages. In C++CSP library, processes are modeled by inheriting from the CSProcess or the ThreadCSProcess class. Processes derived from the CSProcess class run in the process space of the process that created them. As a result, they represent user-level threads.

Processes derived from ThreadCSProcess are kernel-level threads and are scheduled for execution by the OS. Although they are scheduled by the OS, they are synchronized from each other by the C++CSP kernel. As a result, these processes can exploit the multiprocessing power of most Commercial-Off-The-Shelf (COTS) machines. For example, in our initial investigations, we have modeled individual pipeline stages using kernel level thread granularity.

- Channels and Ports: A port on Process X is connected to a port on Process Y using a channel. Each process can read and write to the channels using the reading or writing end, respectively. C++CSP provides both of these types of channels in the form of the following:

  a. One2OneChannels: These represent channels between a pair of processes.

  b. Any2OneChannels: These represent channels from a given set of processes to an individual process. That is, multiple writers compete for writing to the channel end and only one succeeds.

  c. Any2AnyChannels: These represent channels between two groups of processes.

  The channels described above are not *"Buffered"*. This means that the writer is blocked until the reader has read the channel. All of the channels described above also have *"Buffered"* counterparts that model buffered channels. In this case, the writer can continue writing to the channel till the point when the buffer becomes full. After that, the writer is blocked until the reader has read from the buffer. In our work, a channel between any two pipeline stages is modeled using BufferedOne2OneChannel. The reader end is obtained via a call to the channels "*reader()*" method while the writer end is obtained via a call to the channels "*writer()*" method.

- Assignment: Communication via channels is done using overloaded redirect operators. Thus, "*chan » a*" represents reading from the channel into variable $a$. Similarly, "*chan « a*" represents writing to the channel.

- Sequential and Parallel Composition: CSP provides sequential as well as parallel composition of processes. The operator ';' represents the "in sequence" operator. Thus, the notation $[S_i; S_j]$ is used to represent sequential composition. To represent parallel composition of processes, CSP provides the '$\|$' operator. Hence, $[S_i \| S_j]$ indicates that processes $S_i$ and $S_j$ are run in parallel. In C++CSP library, these operators are modeled as RunInParallel(processes) and RunInSequence(processes) helper functions.

- Extended Rendezvous: In cases where the reader controls the duration of synchronization, an extended rendezvous is used. This is represented in C++CSP as ScopedInputs classes.

- In CSP, selection is represented as $[G_1 \rightarrow S_1 \| G_2 \rightarrow S_2 \| ... \| G_n \rightarrow S_n]$ where, $G_1, G_2, ..., G_n$ denote the predicates or "guards" and $S_1, S_2, ..., S_n$ denote concurrent processes. Thus, the statement above

can be interpreted as "wait until guard $G_i$ are true and then execute corresponding processes Si in parallel". Repetition is represented in CSP as "$*[S_i]$". This statement is interpreted as "execute $S_i$ forever". In C++CSP library, selection, wait and repetition are modeled using alternative class, guard class and simple "while" loops in the "*run()*" function, respectively.

### *B. Representing Query Pipeline Using C++CSP*

This section explains how each query pipeline stage shown in Figure 2 is declared in C++CSP. Each pipeline stage/query operator is a derived class of the *CSProcess* class. Program 1 shows the interface of the range-select operator derived from the CSProcess class. Objects of this class scan the entire input table and produce tuples containing row-IDs and values that represent the keys on which hash-join is being performed. Relation A, B and C contain 100,000 tuples each on which we have implemented a range operator with selectivity equal to 50%.

**Program 1** Range Select Operator Implementation

```
class    PSelect:public csp::CSProcess
{
        public:
                PSelect(
                                UINT64 min,
                                UINT64 max,
                                CTable* tbl,
                                const csp::Chanout<UINT64>\& voidChannel,
                                const csp::Chanout<UINT64>\& vvalChannel
                        );
                        ~PSelect();
        protected:
                                virtual void run();
        private:
                                UINT64          _min;
                                UINT64          _max;
                                CTable*         _tbl;

                                csp::Chanout<UINT64>    oidchannel;
                                csp::Chanout<UINT64>    valchannel;
};
```

Program 2 is a partitioned hash-join operator derived from the CSProcess class. Objects of this class create hash-tables on both input relations. However, as opposed to the partitioning algorithm used in [4], we have used radix-clustering to partition the input relations into clusters that can fit into the cache.

Program 3 is a post-projection operator derived from the CSProcess class. Objects of this class project row-IDs of the respective tables from the results of the join.

**Program 2** Radix Cluster-Join Operator Implementation

```
class           PRadixJoin:public  csp::CSProcess
{
        public:
                PRadixJoin(
                        csp::AltChanin<UINT64>& oidChan1,
                        csp::AltChanin<UINT64>& valChan1,
                        csp::AltChanin<UINT64>& oidChan2,
                        csp::AltChanin<UINT64>& valChan2,
                        csp::Chanout<UINT64>& oidChan,
                        csp::Chanout<UINT64>& valChan
                          );

                ~PRadixJoin();
        protected:
                virtual void run();
        private:
                csp::AltChanin<UINT64> oidChan1;
                csp::AltChanin<UINT64> valChan1;
                csp::AltChanin<UINT64> oidChan2;
                csp::AltChanin<UINT64> valChan2;
                csp::Chanout<UINT64> oidChan;
                csp::Chanout<UINT64> valChan;

                TCluster        clusterArrA[256];
                TCluster        clusterArrB[256];

                void AddToCluster(TCluster* clusterArr,
                                  UINT64 oid, UINT64 val
                                 );
                void ClusterJoin();
};
```

These individual processes are then connected using BufferedOne2OneChannels provided by the C++CSP threading library. The C++CSP kernel is used to set up the various processes into producer-consumer relations. The C++CSP manages the concurrency control between threads and buffers. Due to this, the C++CSP kernel manages imbalances in the workload between the various operations.

## V. Experimental Methodology

In this section, we describe our hardware platform, our software platform and the performance monitoring tool used to measure the performance of our partitioned hash-join algorithm on real hardware.

**Program 3** Post Query Project Operator Implementation

```
#include        <cppcsp.h>
#include        <cppcsp.h>
class           Project:public csp::CSProcess
{
        public:
                                Project(
                                        const csp::Chanin<UINT64>& voidChannel,
                                        const csp::Chanin<UINT64>& vvalChannel
                                        );
                                ~Project();
        protected:
                                virtual void run();
        private:

                                csp::Chanin<UINT64>     oidchannel;
                                csp::Chanin<UINT64>     valchannel;
};
```

### A. Hardware Platform

As part of this work, we have evaluated the performance of a query pipeline shown in Figure 2 on an AMD Athlon 64 X2 4200+ dual core processor with each core running at 2.2 GHz. Table 1 describes in detail the characteristics of our hardware platform.

The AMD Athlon microarchitecture consists of a nine macro-op issue, out-of-order superscalar processor pipeline. The processor has three integer units, three address generation units, three floating point units and one load-store unit. There can be as many as 72 macro-ops in flight at one time. The AMD Athlon processor chip has an integrated 64-bit, dual-ported split L1 cache with a separate snoop port and two-levels of Translation Lookaside Buffers (TLB). Each processor core has a discrete integrated 512 Kbyte L2 victim cache. The data cache has eight banks to support concurrent access by two 64-bit loads or stores [22].

The Athlon microarchitecture uses a two-level TLB to determine physical addresses from virtual addresses generated by the processor. The L1 Instruction TLB (ITLB) and Data TLB are both thirty-two entry fully-associative TLBs each. The L2 TLB is a 512-entry, four-way set-associative TLB. Each entry in the TLB maps to a 4K page in memory.

### B. Software Platform

We run our experiments on Microsoft Windows XP, Professional Edition 64-bit (Service Pack 2). For performance measurement, we use AMD Code Analyst 2.76 freely available from the AMD website [23].

Table 1: Hardware platform.

| Hardware Feature | Description |
|---|---|
| Number of CPU Cores | 2 in an AMD Athlon X2 4200+ processor |
| Execution Units | Three integer execution units, three floating point units, three address generation units and one load-store unit |
| Branch Predictor | 2048 two-way set-associative branch predictor |
| L1 Cache | Discrete L1 cache per core, Split 64 Kbyte 2-way set-associative Instr. and Data cache |
| L1 Cache Line Size | 64-byte line size |
| L2 Cache | Discrete L2 cache per core, 512 Kbyte 16-way set-associative cache |
| L2 Type | Victim cache |
| L2 Cache Line Size | 64-byte line size |
| Translation Lookaside Buffer (TLB) | Two-level TLB where each entry maps to a 4K page |
| L1 Instruction TLB (ITLB) | 32 entry fully-associative TLB |
| L1 Data TLB (DTLB) | 32 entry fully-associative TLB |
| L2 TLB | 512 entry four-way set-associative TLB |
| Chipset | nVidia GeForce 6100/NF410 |
| Memory | 4GB DDR2 SDRAM at 3486.34 MHz |
| OS | Microsoft Windows XP, Professional Edition 64-bit (Service Pack 2) |
| HD size | 80 GB |

The AMD code analyst is used to gather event counter data related to the overall performance (such as CPI, data accesses, TLB performance) as well as thread-specific data (such as CPI, data accesses, TLB performance of each executing thread) to evaluate the performance of our multithreaded, pipelined hash-join algorithm.

### *C. Storage Model*

We use a simple column-oriented storage model along the lines of the Decomposed Storage Model (DSM) described in [21]. Here, an n-ary relation is mapped to 'n' binary relations called Binary Association Tables (BATs). Figure 3 describes how a relation 'A' from *Query1* (see Figure 3a) is broken into two BATs namely A_Key and A_Value. The OID is the object identifier for a row. We do not explicitly store OID for each row in our storage model but use a concept of virtual-OIDs where the OIDs are increasing in value and densely packed. Hence, we can determine the OID of a specific row by adding the offset of the row to the OID of the first row. For more details, an interested reader should refer to [21].

| OID | Key | Value |
|---|---|---|
| 0 | 1 | 100 |
| 1 | 2 | 200 |
| ... | ... | ... |

(a) Row Oriented Storage - Relation A

| OID | Key |
|---|---|
| 0 | 1 |
| 1 | 2 |
| ... | ... |

(b) Decomposed Relation - A_Key

| OID | Value |
|---|---|
| 0 | 100 |
| 1 | 200 |
| ... | ... |

(c) Decomposed Relation - A_Value

Figure 3: Decomposed Storage Model

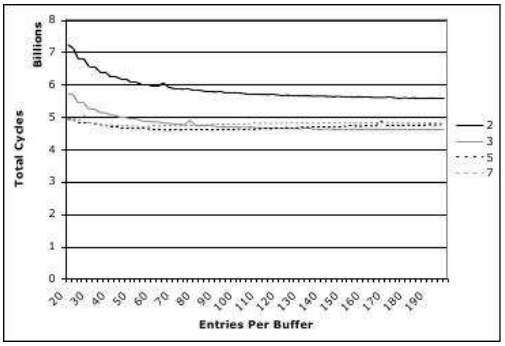
## VI. Results

The results shown in Table 2 are comparable (within an order of magnitude), if not better, than the performance of the pipelined hash-join described in [4]. This can also be seen from Figure 4a and Figure 4b. Furthermore, these results corroborate the findings of Garcia-Korth [4], which indicate that deepening the buffers generally improves performance till the buffer size reaches a certain size beyond which any increase in buffer size starts degrading performance. In our case, the best case performance is obtained when the buffer size is around 500 tuples (where each entry is of 128 bytes size each). This seems to match closely with the size of L2 cache, which is 512 KB. We hypothesize that the maximum performance is obtained when the intermediate relations can fit entirely into the L2 cache. Note that although our implementation takes fewer number of cycles than that of the implementation done by Garcia-Korth [4], we hesitate to claim that the algorithm implemented using C++CSP is faster because of two reasons:

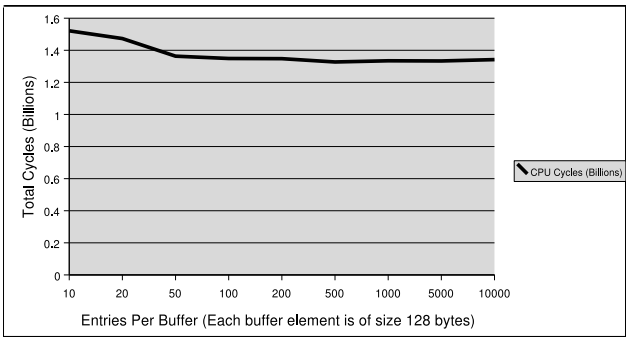
- We have used a relatively simple column-oriented storage model for storing our test data (as described in Section V-C) while Garcia-Korth [4] used a traditional row-oriented storage model.

Table 2: Buffer size vs. execution cycle results (the larger, the slower)

| Buffer Size | 10 | 20 | 50 | 100 | 200 | 500 | 1,000 | 5,000 | 10,000 |
|---|---|---|---|---|---|---|---|---|---|
| CPU Cycles (Billions) | 1.52 | 1.47 | 1.36 | 1.35 | 1.35 | 1.33 | 1.33 | 1.33 | 1.34 |




(a) Pentium 4 results (courtesy of [4])




(b) AMD dual core 64 X2 results (our experimentation)

Figure 4: Performance of pipelined hash-join as a function of buffer size

- We need to examine performance of the pipelined database algorithm on different hardware platforms before we can make a comprehensive claim that our algorithm is faster. We have used an AMD dual core 64 X2 hardware machine while Garcia-Korth [4] used a Pentium 4 machine. Due to time limitations, we have not performed our experimentation on any other hardware platform. We will perform such experimentation as part of our future work in order to strengthen our claim that the implementation using C++CSP performs better than that of Garcia-Korth.

## VII. Conclusion

In this technical report, we described the implementation of a pipelined hash-join algorithm using a threading library based on the formal method of Communicating Sequential Processes (CSP). Furthermore, we corroborated the results of our implementation with a previously published result of Garcia-Korth [4]. These results are quite encouraging and seem to indicate that the performance of a pipelined hash-join algorithm implemented using C++CSP is comparable (within an order of magnitude), if not better, than the performance of a pipelined hash-join algorithm implemented using the Pthread threading model by Garcia-Korth [4]. Furthermore, we showed how C++CSP can be used to develop a pipelined database algorithm. Other pipelined algorithms and eventually a staged database can be similarly implemented using C++CSP. However, further development of such algorithms and the staged database is left as future work.

## VIII. Acknowledgment

We would like to thank IUPUI RSFG internal grant that enabled the research performed here.

## References

[1] A. Ailamaki, D. DeWitt, M. Hill and D. Wood, "DBMSs on a modern processor: where does time go?", In Proc. of the VLDB'99, pp. 266-277, Sept. 1999.

[2] S. Harizopoulos and A. Ailamaki, "A case for staged database systems," In Proc. of 1st Conference on Innovative Data Systems Research, 2003.

[3] S. Chen, A. Ailamaki, P. B. Gibbons and T. C.Mowry, "Improving hash join performance through prefetching," In Proc. of Intl. Conf. on Data Engineering, 2004.

[4] P. Garcia and H. F. Korth, "Pipelined Hash-Join on Multithreaded Architectures," In Third International Workshop on Data Management on New Hardware (DaMoN 2007), 2007

[5] J. Cieslewicz, J. Berry and B. Hendrickson *et al*, "Parallelism in database operations: insights from a massively multithreaded architecture," In Proc. of the 2nd international Workshop on Data Management on New Hardware DaMoN '06, June 2006.

[6] J. Zhou, J. Cieslewicz, K. A. Ross and M. Shah, "Improving database performance on simultaneous multithreading processors," In Proc. of the VLDB'05, pp. 49-60, 2005.

[7] C. Hoare, *Communicating Sequential Processes*, Prentice Hall. ISBN 0-13-153289-8

[8] E. Lee, "The Problem with Threads," In IEEE Computer, May 2006.

[9] N. Brown and P. Welch, "An Introduction to the Kent C++CSP Library," In Communicating Process Architectures 2003.

[10] N. Brown, "C++CSP networked," In Communicating Process Architectures 2004.

[11] T. MacKenzie and T. Dix, "Object-oriented Ease-based parallel primitives in C++," In International Conference on Parallel and Distributed Systems (ICPADS'98), pp. 623, 1998.

[12] A. Roscoe and C. Hoare, "The laws of Occam programming," In Theor. Comput. Sci. 60, 2, pp. 177-229, 1988.

[13] G. Theodoropoulos, *Strategies for the modelling and simulation of asynchronous computer architectures*. PhD Thesis, Department of Computer Science, University of Manchester, 1995.
Available at: ftp://ftp.cs.man.ac.uk/pub/amulet/theses/theo95-phd.ps.Z.

[14] C. Chien, M. Franklin, T. Pan and P. Prabhu, "ARAS: asynchronous RISC architecture simulator," In Proc. of the 2nd Working Conference on Asynchronous Design Methodologies, 1995.

[15] E. Lee and A. Vincentelli, "Comparing models of computation," In Proc. of the International Conference on Computer-Aided Design, pp. 234-241, 1996.

[16] J. Moores, "CCSP - a portable CSP-based run-time system supporting C and Occam," In Architectures, Languages and Techniques for Concurrent Systems, volume 57 of Concurrent Systems Engineering series, pp. 147-168, 1999.

[17] P. Welch, B. Vinter and J. Pascoe, "Cluster computing and JCSP networking," In Communicating Process Architectures, 2002.

[18] G. Hilderink, J. Broenink and A. Bakkers, "Communicating threads for Java," In Communicating Process Architectures, 1999.

[19] R. Peel, "A reconfigurable host interconnection scheme for Occam-based field programmable gate arrays," In Communicating Process Architectures, 2001.

[20] P. Welch and D. Wood, "The Kent retargetable occam compiler," Parallel Processing Developments, In Proc. of WoTUG 19. 1996.

[21] P. A. Boncz, *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.D. Thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[22] AMD-Athlon Technical Brief, www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22054.pdf, visited Dec. 2007.

[23] AMD CodeAnalyst Performance Analyzer for Windows, http://developer.amd.com/cawin.jsp, visited Dec. 2007.