LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Exploiting Parallelism in the Implementation of AGNA, a Persistent Programming System

Computation Structures Group Memo 320
December 21, 1990

Rishiyur S. Nikhil *and* **Michael L. Heytens**

In *Proc. Seventh International Conference on Data Engineering*, Kobe, Japan, April 8-12, 1991.

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988. Support for Michael Heytens is provided in part by an Intel Graduate Fellowship.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Exploiting Parallelism in the Implementation of AGNA, a Persistent Programming System

Rishiyur S. Nikhil
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA

Michael L. Heytens
Massachusetts Institute of Technology
Room 36-667, 77 Massachusetts Avenue
Cambridge, MA 02139, USA

Abstract: We present our design for AGNA, a persistent object system that utilizes parallelism in a fundamental way to enhance performance. The underlying thesis is that fine-grained parallelism is essential for achieving scalable performance on parallel MIMD machines. This, in turn, implies a data-driven model of computation for efficiency. We present a complete design based on these principles. We start with a declarative source language because such languages reveal the most fine-grained parallelism. We describe how we compile transactions into an abstract, fine-grained parallel machine called P-RISC. The P-RISC virtual heap is implemented in the memory and disks of a parallel machine in such a way that paging is overlapped with useful computation. We describe the current implementation status, report some preliminary performance results, and compare our approach to several recent parallel database system projects.

1 Introduction

It is now widely recognized that the expressive power of conventional database systems is inadequate for application domains involving complex structures, such as integrated circuit CAD and computer-aided software engineering. To address this, many researchers are attempting to unify programming languages and databases into persistent programming languages—that is, full programming languages in which arbitrary user-defined objects may persist, augmented with transaction mechanisms for reliability and concurrency control [7, 6, 3, 20, 13, 8].

A major challenge in implementing such systems is performance. Conventional database systems achieve good performance in part because of flat, static data structures, which allow detailed planning of data layouts on disk, information that is used heavily in the construction of efficient indexes, retrieval algorithms and in query optimization. In persistent programming languages, where data structures are less regular and more dynamic, this pre-planning is less feasible and program optimization is more difficult. For these reasons, persistent programming languages have not yet achieved performance that is competitive with traditional database systems.

In this paper, we describe our approach to obtaining good performance in the AGNA persistent programming system by exploiting parallelism. We start with a *declarative* language, because such languages reveal abundant opportunities for parallel execution. SQL is an example of a declarative language, but we are working with a more general language that is based on complex objects instead of flat relations. Transactions are compiled into code for an abstract machine whose central feature is fine-grained parallelism with data-driven execution, both for normal computation as well as for disk I/O. Prior research into dataflow architectures indicates that this is an effective way to mask long-latency operations, which are inherent in both memory and disk accesses in parallel computers.

In Section 2, we give a brief description of our declarative persistent programming language, and discuss how it facilitates parallelism (the parallelism discussion carries over to other declarative languages, including SQL). In Section 3, we describe the AGNA implementation framework for exploiting parallelism, including compilation to fine-grained parallel threads, memory organization (both volatile and persistent), and file structures supporting persistent memory. In Section 4 we examine the compilation and execution of an example transaction. In Section 5, we compare our work to several recent database system projects. Finally, we conclude in Section 6 with a description of the current status of our prototype implementation, which runs on a network of Unix workstations, report some preliminary performance results, and describe our progress in porting the system to a real multiprocessor.

2 The Language

We model a database as an environment of bindings, *i.e.*, a table associating names with objects. The objects may be types, scalars, complex objects, lists, procedures, indexed mappings, *etc.* Objects may refer directly to other objects.

The following program fragments define two new structured-object types and some of their slots (we currently use a Lisp-like notation to avoid detailed syntax design).

```
(type STUDENT (extent)
 ((name         <=>  STRING)
  (courses   *<=>* COURSE)
  (total-units =>  NUMBER)
  (year        =>  NUMBER)
  ...))

(type COURSE (extent)
 ((name   *<=> STRING)
  (number <=> STRING)
  (units  =>  NUMBER)
  ...))
```

In addition to introducing type names **STUDENT** and **COURSE**, these declarations indicate that persistent extents containing all students and courses are to be maintained automatically, realizable as lists via the expressions **(all student)** and **(all course)**, respectively. The forms also define the name and type of each slot, whether they are single- or multi-valued, and whether they have single- or multi-valued inverses. For example, the student **name** slot records a single string value, and supports a unique inverse mapping strings to students. The **courses** slot is multi-valued, with a multi-valued inverse mapping a course to a collection of students. The basic operations available for manipulating structured objects, of both pre- and user-defined types, are allocation, slot selection, and slot definition.

Unlike the fixed schema of traditional databases, the environment is extensible, *i.e.*, the user can introduce new bindings. For example, the following transaction defines a function **courses-with-name** that maps a string **s** to a list of courses with that name (using the generic **invert** form for inverted slots):

```
(xact
  (define courses-with-name
    (lambda (s) (invert course name s))))
```

Read-only transactions (queries) are arbitrary expressions, including procedure calls, that refer to the objects of the database. A very concise and powerful notation for building such expressions is the "list comprehension." For example, here is a query to find the names of all first-year students taking software engineering:

```
(xact
  (all (student-name s)
    (s (students-with-year 1))
    (c (courses-with-name "Software Engineering"))
    (where (member c (student-courses s)))))
```

In words: for all first year students **s** and courses **c** with name "Software Engineering," such that **c** is a course taken by **s**, find the name of each such student **s**.

List comprehensions are common in functional languages (such as Haskell) and can be regarded as a generalization of SQL queries. In [26], Trinder shows how to translate any relational calculus expression into a list comprehension, and also that list comprehension queries are amenable to many of the standard optimizations (both algebraic and implementation-based) performed by conventional systems. List comprehensions are more general than SQL because (a) they allow arbitrary generator lists (that is, the lists over which identifiers such as **s** and **c** range may be arbitrary expressions, not just "base" extents), (b) they allow arbitrary predicates, not just arithmetic and string comparisons, and (c) the returned values can be of arbitrary type and computed using arbitrary functions, not just field projections.

For reasons of parallelism, update transactions in AGNA are also declarative. Conceptually, update specifications are collected during transaction execution, and occur instantaneously at transaction commit time. No ordering of updates is specified; to avoid nondeterminism, therefore, the new value of any component must be specified at most once— any multiple specification aborts the transaction. The update itself is not visible in the current transaction.

Here is an example of an update transaction that increases the number of units for course CS101, also updating the total units for each student that has taken the course.

```
(xact
  (deflocal c (course-with-number "CS101"))
  (update c course units (+ 3 (course-units c)))
  (foreach
    (students-in-course c)
    (lambda (s)
      (update s student total-units
              (+ 3 (student-total-units s))))))
```

The transaction binds local identifier **c** to the desired course object, updates its **units** slot, and updates the **total-units** slots of all students that have taken the course. Slot update is performed via the generic **update** form, which takes a structured object, type and slot names, and a new slot value.

A more complete description of the language may be found in [19].

3 Implementation

A Fine-grained, Data-driven model

Even if we begin with a language and compilation method that are capable of expressing fine-grained parallelism, there still remains the question of how this parallelism is best implemented on a real machine. Research into dataflow architectures indicates that an effective way to achieve high utilization in a parallel machine is to use a fine-grained, data-driven execution model (see [4, 5]). The analysis and experimental results may be summarized as follows.

Parallel MIMD machines are dominated by asynchronous events. Even on uniprocessors, it is already well recognized that asynchronous events are more efficiently handled by an interrupt-driven model rather than one that uses polling, because it avoids busy-waiting. An interrupt is a simple example of data-driven scheduling— when data becomes available, an interrupt occurs, and a *continuation* is specified (*via* an interrupt vector), which is the thread to be activated to accept the data.

Dataflow models of computation take this idea to the limit— *all* scheduling is uniformly data-driven. All long-latency (asynchronous) operations are structured as *split-phase* actions. Examples of long-latency operations include: memory reads across a parallel machine, disk transfers, procedure calls, requests for resources (such as heap allocation), *etc.* In the first phase, a request is sent from point A to point B in the machine, carrying with it three pieces of information:

(1) the continuation **contB** in B that will handle this request;

(2) the arguments for the request, and

(3) the continuation **contA** in A that will handle B's response.

When the request arrives at B, the continuation **contB** is activated (scheduled), which uses the arguments to perform the remote computation. When this is complete, the second phase occurs: B sends the result back to A, accompanied by **contA**. When the response arrives at A, the continuation **contA** is scheduled to compute using the result. Thus, a natural coroutining structure is inherent in the model.

The benefit of split phase actions is that processor A does not have to block while waiting for B's response— it is free to perform other computations in the interim.

Further, the benefit of passing continuations around is that a continuation directly identifies the thread to be woken up, making scheduling very efficient. Messages do not have be processed in any particular order. These features are invaluable in achieving high processor utilization.

Fine-grained threads are useful for scalability and load balancing. The performance of a parallel system should improve if we can provide more parallel resources (more processors, memories, and disks). Having small threads ensures that even with more processors, each processor still has enough threads to keep it busy while some threads are blocked. Further, small threads give more flexibility in the distribution of work across the machine.

How the Language Model Facilitates Parallelism

The above argument for fine-grained threads can be made independently of the language model. However, it is very difficult to compile such threads from traditional programming languages. Because they are usually tied inextricably to an imperative model, partitioning into parallel threads without introducing read-write races (due to side-effects) requires complex *dependency analysis*, which is difficult in all but the simplest programs. For declarative languages, on the other hand, the lack of side-effects makes it particularly easy to compile into very fine-grained threads. In the terminology of parallelizing compilers, there are no *anti-dependencies* in the language. We refer the reader to [4] for substantial evidence that compilers for declarative languages can effortlessly extract orders of magnitude more parallelism than is possible with traditional languages

Compilation of transactions

Compilation occurs in two major phases— translation of the transaction text into *dataflow program graphs* (DFPGs), and translation of DFPGs into code for a multi-threaded abstract machine called P-RISC. Substantial code optimizations are performed at each stage.

Our compiler has no knowledge of certain details of the parallel machine, such as the number of processors and memories, interconnection scheme, *etc.* The object code produced by our compiler is portable across machines with different configurations.

Dataflow Program Graphs

The translation to DFPGs largely follows the methods outlined in [25]. A DFPG is, roughly, a "data-driven" representation of the abstract syntax tree. For example, the expression `(students-in-course (course-with-number ''CS101''))` yields the program graph shown in Figure 1.

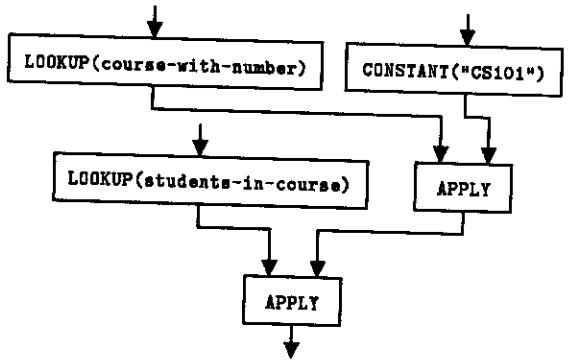

Figure 1: Example DFPG.

DFPGs have a precise operational semantics. Trigger tokens initiate the **LOOKUP** and **CONSTANT** instructions. All three of them may execute in parallel, and will produce result tokens on their output arcs. The **APPLY** instructions invoke the bodies of the incoming procedures, pass along any arguments, and send the results along their output arcs.

Unlike the **LOOKUP** and **CONSTANT** instructions, **APPLY** is not strict with respect to all its inputs—that is, it may begin executing before all its input tokens have arrived. As soon as the procedure input is available, **APPLY** may initiate execution of the procedure body. When argument inputs arrive, they are simply passed on to the procedure body. As soon as a result is available, it may be sent along the output arc. The non-strict operation of **APPLY** captures exactly the non-strictness of procedure calls in the AGNA language. This results in an important source of parallelism, one which allows the body of a procedure to execute in parallel with its arguments. Such parallelism is limited only by data dependencies.

DFPGs also capture all additional forms of parallelism in the source text. For lack of space, we refer the reader to [25] for details of how DFPGs can express conditionals, loops, and higher-order functions. DFPGs are largely architecture-independent and provide a convenient form for performing a variety of optimizations and transformations[1].

[1] Conventional optimizing compilers, for example, often use dataflow graphs to aid in program analysis.

P-RISC code

The final target of the compilation process is a multi-threaded RISC-like abstract machine called P-RISC [18]. The P-RISC instruction set includes the usual RISC primitives such as load/store, arithmetic, logic, and relational operations, and also primitives for the explicit management of parallel threads. Examples of the latter include a **fork** instruction for spawning a new thread and a **join** instruction for combining parallel threads.

P-RISC threads are truly lightweight threads, requiring no operating-system support, and **fork** and **join** are instructions of no more complexity than conditional jumps. Each thread is described by a pair: an instruction pointer (IP) and a frame pointer (FP). Frames are like activation records, *i.e.*, they are allocated and deallocated as part of procedure invocations. There can be many simultaneously active frames, and each frame can have many simultaneously active threads.

A heap access is performed through a **load** instruction, which is a split-phase instruction because it has a potentially long latency (*e.g.*, the heap location may not be in a processor's local memory, or the location may not currently be paged in). A request to read the heap location is sent to the appropriate memory, along with the current continuation. The response comes back with this continuation, so that the processor knows exactly where to resume. In the interim, it can execute other threads from its pool of threads.

The data-driven framework also makes it easy for heap accesses to be synchronized. Each heap location has extra bits that indicate whether it is full or empty. The reader and the writer of that location may execute concurrently. If the read arrives first at the location (it is still empty), the accompanying continuation is simply queued there. When the write arrives, all continuations waiting at that location are then notified. Note: this relies crucially on the fact that each heap location is written exactly once, a natural consequence of the declarative programming model.

The idea of split-phase operations is also naturally extended to "manager calls" (or supervisor calls) for manipulating the persistent store, and managing different aspects of transaction execution. For example, we have managers for defining and looking up database bindings, allocating persistent objects, filtering type extents (with simple predicates), and coordinating transaction commit.

Translation from DFPGs into P-RISC code is straightforward. Our compiler also analyzes the graph to determine which pieces are worth doing in parallel. Roughly speaking, for connected subgraphs that do not involve long-latency operations, the compiler produces sequential code (for more details, see [16]).

Persistence

The compiler's model of the runtime environment consists of a P-RISC program containing many threads, operating off a single-level, global, virtual heap. Half of the heap is volatile, and the other half persistent, *i.e.*, its data are backed up in various disk files (to be described soon). During transaction execution, new objects may be allocated in the volatile and persistent parts of the heap, and new versions of persistent objects may be produced. At transaction commit time, newly allocated, reachable persistent objects are copied into the persistent heap, updates are installed, and the disk files are brought up-to-date. An object is identified by its heap address.

Physical machine model

We are targeting our implementation to MIMD multiprocessors consisting of processor-memory elements (PMEs), with or without attached disks, interconnected via a high-speed network (see Figure 2). The network may be a bus, in small multiprocessors, or a switching network in larger machines.

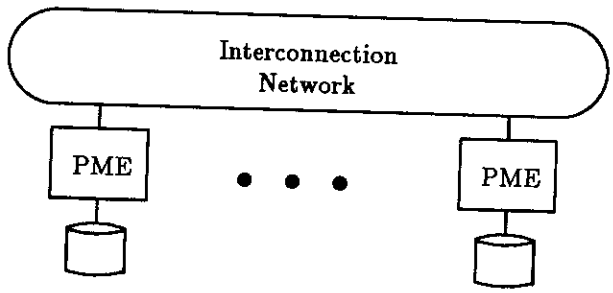

Figure 2: Machine organization.

Mapping the virtual heap to the physical machine

Apart from the usual requirement of fast translation from virtual heap addresses to physical memory addresses, we also require fast reconfigurability of a database to a machine with a different number of PMEs. For example, it should be easy to reconfigure a database originally constructed on a 16 PME machine to a 15 PME machine (perhaps a PME failed) or a 32 PME machine (perhaps the machine was upgraded).[2]

Our heap addresses are 42 bits wide. The heap is divided into *segments* of size 2^{32}, which is the unit of distribution across PMEs, *i.e.*, each segment is contained entirely within one PME. A segment-to-PME mapping table (many-to-one) is replicated on all PMEs. The first stage of heap address translation, therefore, involves consulting this table to decide which PME holds the target heap location.

Objects could be distributed across PMEs using various strategies, such as round-robin allocation, random allocation, "minimum object load," *etc.* This is a major topic that we plan to investigate soon.

Within a PME

Each segment is divided into *pages* of size 8K bytes. On each PME, physical memory consists of a collection of *page frames*, which caches a subset of all the virtual pages in that PME. The correspondence is maintained in a hash table:

(segment, page) $\longrightarrow$ page-frame

Thus, the second stage of heap address translation involves probing this table. If successful (*i.e.*, the page is in physical memory), then the heap location can be accessed.

The pages within a segment are partitioned into a persistent part and a volatile part; the high-order page bit distinguishes between the two. All volatile pages in (all segments in) a PME are mapped to a single paging file.

The persistent pages have more structure. We define an *extent* as a contiguous set of persistent pages (we use 512 pages). We place a further restriction that an extent can only contain objects of a single type. All extents of a particular type (*e.g.*, `STUDENT`) are mapped to a single file (`student.dat`). The correspondence is maintained in a hash table:

(segment, extent) $\longrightarrow$ (file, page-offset)

[2] For now, we are only considering static, or offline reconfiguration.

This grouping allows us to perform bulk operations on type extents efficiently (*e.g.*, finding all students with age 18), and to build indexes on files to support the invertible slot-functions specified in the source-language type declarations. The reason for mapping contiguous page groups (extents) to files rather than individual pages is to keep the mapping tables from becoming too large.

Transaction Execution

After compilation, a transaction is assigned an id, and its code is loaded into the ephemeral heap memory of a node in the machine. This initial node is the *coordinator* for the transaction. Conceptually, a transaction executes in three phases: a prologue, a body, and an epilogue. During the prologue, transaction-specific initialization is performed. In the second phase, the body of the transaction is executed and the result is printed. In the epilogue, the transaction's updates (if any) are installed in the persistent heap and a two-phase commit procedure is carried out by the coordinator.

The installation of updates includes locating and making persistent all objects reachable from the database. This is accomplished by computing the transitive closure of the transaction's *persistent root set*, which consists of new top-level bindings, new slot values in persistent objects, and new objects with automatic type extents. When reachable ephemeral objects are encountered during the closure traversal, they are migrated to the persistent heap. This migration process is analogous to that performed by a copying garbage collector.

4 Example

In this section we examine the translation and execution of the following AGNA transaction:

```
(xact
  (students-in-course
    (course-with-number ''CS101'')))
```

In the first phase of compilation, the transaction body is rewritten to a sequencing form that includes calls to library routines to begin the transaction, print the result, and end the transaction.

```
(xact
  (seq
    (begin-transaction)
    (print (students-in-course
             (course-with-number ''CS101'')))
    (end-transaction)))
```

Procedure **begin-transaction** implements the transaction prologue and **end-transaction** implements the epilogue.

The body of the transaction contains two calls to functions which perform slot inversions, one finding the course numbered "CS101" and the other finding all students in the course. Since the implementations of the two inversions (and other kinds of extent filtering) are similar, we will focus only on the latter one.

Let us assume that the first inversion returns `c`, a reference to a course object. Conceptually, the execution strategy for the second inversion is to dispatch `c` to all PMEs on which student objects reside, and for each PME to perform a local inversion, building a list of all local students taking the course. The final result is formed by appending all local lists. This last step is performed in an efficient manner by directly linking the end of each local list to the head of the preceding list (`nil`, the list terminator, is stored at the end of the last local list).

The execution strategy is selected by the compiler during compilation of **students-in-course**. The function body, **(invert student course c)** for course `c`, is translated by the compiler to the following call to **mv-inversion** ("mv" for "multi-valued"), the library routine which actually implements the inversion:

```
(mv-inversion (type-pme-list t) s c nil)
```

The first argument is the list of PMEs on which student objects reside, the second is the id of the student course slot, the third is the actual course object, and the fourth is the initial result list.

Function **mv-inversion** is defined as follows:

```
(define (mv-inversion pmes s v rest)
  (if (nil? pmes)
      rest
      (apply local mv-inversion (tl pmes) s v
        (apply pme (hd pmes)
          local-mv-inversion s v rest))))

(define (local-mv-inversion s v rest)
  (let ((res (mvinvert s v)))
    (if (nil? res)
        rest
        (par
          (set-cons-tl (tl res) rest)
          (hd res)))))
```

The function uses the **apply** form to express procedure application and to specify hints (*local*, *remote*, *random*, or *pme n*) on where the body of a procedure should be executed.

The implementation of **mv-inversion** offers many opportunities for parallel execution. Since the language

supports non-strict procedure calls (*i.e.*, the body of a procedure may begin executing before all of its arguments are ready) the recursive call to `mv-inversion` may proceed, even though the result of the call to `local-mv-inversion` may not be available. Thus, `mv-inversion` can immediately dispatch all local inversions, which may then proceed in parallel.

Local result lists are produced in `local-mv-inversion` by the primitive manager `mvinvert`, which takes a slot identifier and a slot value. The object returned by `mvinvert` is either `nil`, if no local student objects were found, or an *open list*. Open lists, which are closely related to *difference lists* in logic programming, can be represented as a cons cell containing two references A1 and A2, as shown in Figure 3. Internally, the structure consists of a list in which the tail of the last cons cell is empty, denoted via the symbol $\perp$. (Note: $\perp$ is not the same as `nil`! That is, reading the tail of a cons containing `nil` simply returns `nil`, while reading the tail of an empty cons results in deferral of the reader until the tail is written.) A1 points at the first cell and A2 points at the last cell (these may in fact be the same cell).

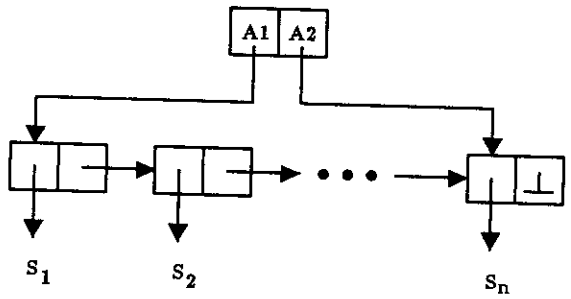

Figure 3: The structure of an open list.

The characteristic feature of open lists is that they are easy to append. In `mv-inversion`, the local lists are appended by threading an intermediate result list through each invocation of `local-mv-inversion`. The first invocation receives the initial result of `nil`, which it stores in the tail of its local result list. It then returns a reference to the head of the local list (*i.e.*, its local A1). The second invocation receives this reference, stores it in the tail of its local result list, and returns the new intermediate result list. This process continues until the last invocation is reached, which appends its local list and returns the final result to `mv-inversion`. If a local inversion produces no result (*i.e.*, `mvinvert` returns `nil`), then the incoming intermediate result list is simply passed along.

This example illustrates another important source of parallelism in the language, one which enables the producer and consumer of a data structure to execute concurrently. The `print` function, for example, may begin traversing the result list (to print each element) as soon as the first segment of the result list is available (*i.e.*, as soon as the corresponding invocation of `mv-local-inversion` produces a result). It may do this even though the entire result may not be ready. If an attempt is made to read an unwritten cons tail, then the read operation is simply deferred (by enqueuing the continuation). When the write finally arrives, all deferred readers will be notified, allowing the list traversal to continue.

Lastly, the example also illustrates the way in which execution is driven by the availability of data. For example, when the PME list is available, `mv-inversion` may begin traversing it, dispatching `local-mv-inversion` functions. When the slot identifier and slot value are available within `local-mv-inversion`, `mvinvert` may be invoked to build the local result list. Finally, when an intermediate result list is available, `local-mv-inversion` may append its local result (propagation of the new intermediate result may be performed as soon as a non-nil local result is available).

5 Related Work

There have been many attempts to exploit parallelism in the implementation of persistent systems ([10, 15, 9, 2, 24, 23, 17] and see [11, 12] for summaries of earlier work). Recently, commercial products and high-performance research prototypes have demonstrated that parallelism can be utilized effectively in the implementation of the relational model. Tandem's Nonstop SQL exploits inter-transaction parallelism to achieve a linear speedup from 14 to 208 DebitCredit transactions [1] per second as the hardware is increased from 2 to 32 nodes [23]. Gamma exploits intra-transaction parallelism to achieve roughly linear speedup for single-user execution of relational queries as the hardware is increased from 1 to 30 nodes (*i.e.*, for a constant-sized database, doubling the number of PMEs roughly halves the response time) [14].

AGM [9], Gamma, and Bubba [10] all use data-driven execution and a machine architecture similar to ours. AGM supports a language based on the Entity-Relationship data model, and memory is viewed as an active graph of tokens, which are used to represent both entities and relationships. Tokens are mapped to PMEs of the machine by a hashing function, and transaction

processing consists of injecting and propagating special query and/or update tokens through the graph.

Gamma and Bubba use a "shared-nothing" [22] memory organization in which PMEs do not share main memory or disk drives, and communicate solely via message-passing. Gamma implements the basic relational data model, while Bubba implements an extended relational model that includes such things as nested relations, user-defined functions, object identity, and iterative and conditional constructs. Both systems partition relations horizontally across PMEs. Also, both utilize coarse-grained parallelism; for example, a relational query compiles to a dataflow graph containing operators such as *select*, *project*, and *join*. An operator is implemented via one or more processes (or process threads) running on each PME which participates in the operator. For relational queries, the query processing strategy to be used and the process-to-PME mapping are determined largely at compile-time.

In contrast to the Gamma/Bubba approach, we use a much finer parallel grain size and a memory model consisting of a global heap and procedure frames (or activation records). Another major difference is the way in which parallel tasks are mapped to PMEs. In our system, tasks are often mapped to PMEs at run-time, based primarily on considerations of load balancing and object location. For an SQL-like subset of our language, however, the compiler is often able to select a query processing strategy which provides hints on where certain computations should be performed, exploiting the locality of data. While such methods work well for relatively simple operations on regularly structured data, it is not clear that they can be easily generalized to more sophisticated operations on complex data.

The final significant difference is the way in which parallel tasks are synchronized and controlled. In Gamma/Bubba, a separate control process is used to inform an operator process of the identity of the processes to (from) which it is to send (receive) data. When the processes which implement an operator complete, they inform the control process, which in turn may initiate other operator processes. Consumer processes block while waiting for data from producer processes. In our approach, synchronization is performed explicitly via the `join` instruction (which combines parallel threads) or implicitly through heap access (readers of an empty location are automatically deferred until the location is written). These mechanisms make minimal use of OS synchronization primitives and are entirely data driven, involving no busy-waiting.

Control is handled uniformly through the use of continuations, which can be thought of as the ultimate in lightweight threads. For operating systems which support non-blocking I/O, *all* threads on a PME, possibly from different transactions, may execute within the same OS process.

6 Concluding Remarks

We have presented our design for AGNA, an expressive persistent object system that utilizes parallelism in a fundamental way to enhance performance. We use a declarative transaction language which includes such features as list comprehensions (a concise notation for expressing nested iteration), complex objects, user-defined types and procedures, and indexed mappings. Transactions are first compiled to dataflow graphs, and then to an abstract machine whose central feature is fine-grained parallelism with data-driven execution. By compiling to fine-grained threads and using a data-driven model of execution, useful computation may be overlapped with long latency operations such as disk I/O and remote memory accesses.

A prototype implementation of AGNA is already operational, running on Unix machines interconnected via a local area network. PMEs are implemented via P-RISC emulator programs written in C. A single machine may host several emulator programs, or each emulator may be mapped to its own physical machine. The compiler is written in Common Lisp. This initial environment (Unix, C and Lisp) was chosen because of its widespread availability, the flexibility it offers, and the many program development tools such as profilers and debuggers that it supports.

We have just recently tested the uniprocessor performance of AGNA against Ingres (version 6.2) on the benchmark of simple operations described in [21]. The tests were conducted on a Sun 4/490 with 128 Mb of main memory and an HP 97549 disk. The Ingres version was coded in EQUEL/C. The "small" version of the benchmark database was used (20,000 person objects, 5,000 document objects, and 15,000 author objects). The results, tabulated below, are expressed in milliseconds.

Operation	AGNA	INGRES
Name Lookup	10	17
Range Lookup	61	106
Group Lookup	18	37
Reference Lookup	8	17
Record Insert	14	35
Sequential Scan	.733	.395

Notes on benchmark operations: Name Lookup is the time to fetch a record given its unique id; Range Lookup is the time to fetch ten records which have field values in a specified range; Group Lookup is the time to find three records of one type which reference a given record of another type; Reference Lookup is the time to fetch a record that is referenced by a field of another record; Record Insert is the time to insert one record; and Sequential Scan is the time to scan all objects of a particular type divided by the number of objects scanned. A complete description of the benchmark operations is given in [21].

While these preliminary results for AGNA are very encouraging, we must keep in mind that we have not yet implemented concurrency control (across transactions) and recovery, which we plan to do in the near future. Undoubtedly, this will slow our system somewhat, although it is too early to say by how much. After this, and after completing our implementation of optimizations, we will perform a complete analysis of the performance of both AGNA and Ingres on the benchmark.

A port of our software is also well under way to an Intel iPSC/2 Hypercube with 32 processors and 32 disk drives. When the port is finished, we plan a thorough assessment of the performance and scalability of the system, both on traditional and non-traditional benchmarks. The results of this analysis will be the subject of a future paper.

A major issue that we plan to investigate is the extent to which I/O latency—a major obstacle to good performance in both uniprocessor and multiprocessor database systems—can be masked through parallelism. If enough parallelism is available, then a processor can avoid idling while waiting for I/O to complete by executing other threads that are ready to run. Finally, although it is presently only a goal, our fine-grained parallel model will allow us to go down to the level of scheduling disk requests to reduce head movement.

Acknowledgements

We thank David DeWitt for giving us access to the Intel iPSC/2 at the University of Wisconsin, and for assisting us in the actual use of the machine. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988. Support for the second author is provided in part by an Intel Graduate Fellowship.

References

[1] Anon et al. A measure of transaction processing power. *Datamation*, 31(7):112–118, April 1985.

[2] P. Apers, M. Kersten, and H. Oerlemans. PRISMA database machine: A distributed, main-memory approach. In *Proceedings of the International Conference on Extending Database Technology*, Venice, Italy, 1988.

[3] Proceedings of the workshop on persistence and data types, appin, scotland, August 1985.

[4] Arvind, D. E. Culler, and G. K. Maa. Assessing the Benefits of Fine-grained Parallelism in Dataflow Programs. *International Journal of Supercomputer Applications*, 2(3), 1988.

[5] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany, Springer-Verlag LNCS 295*, June 25-29 1987.

[6] M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[7] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. PS-algol: An Algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1981.

[8] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, and F. Velez. The Design and Implementation of O_2, an Object-Oriented Database System. In *Advances in Object-Oriented Database Systems, Springer-Verlag*, September 1988.

[9] L. Bic and R. L. Hartmann. AGM: A dataflow database machine. *ACM Transactions on Database Systems*, 14(1):114–146, March 1989.

[10] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, March 1990.

[11] H. Boral and D. J. DeWitt. Database machines: An idea whose time has passed. In *Proceedings of the 1983 International Workshop on Database Machines*, August 1983.

[12] H. Boral and S. Redfield. Database machine morphology. In *Proceedings of the 11th International Conference on Very Large Databases*, Stockholm, Sweden, August 1985.

[13] G. Copeland and D. Maier. Making smalltalk a database system. In *Proc. ACM SIGMOD*, page 325, 1984.

[14] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[15] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In *Proceedings of the 1986 Conference on Very Large Data Bases*, August 1986.

[16] R. A. Iannucci. *A Dataflow/von Neumann Hybrid Architecture.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1988.

[17] M. Kitsuregawa, H. Tanaka, and T. Moto-oka. Architecture and performance of parallel relational database machine GRACE. In *Proceedings of the International Conference on Parallel Processing*, 1984.

[18] R. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, May 28 - June 1 1989.

[19] R. S. Nikhil. The semantics of update in a functional database programming language. In *Proceedings of the ALTAIR-CRAI Workshop on Database Programming Languages*, Roscoff, France, September 1987.

[20] Proceedings of the ALTAIR-CRAI workshop on database programming languages, roscoff, france, September 1987.

[21] W. Rubenstein, M. Kubicar, and R. Cattell. Benchmarking simple database operations. In *Proceedings of the 1987 ACM SIGMOD Conference*, San Francisco, CA, May 27-29 1987. ACM.

[22] M. Stonebraker. The case for shared nothing. *Database Engineering*, 9(1), 1986.

[23] Tandem Performance Group. A benchmark of non-stop sql on the debit credit transaction. In *Proceedings of the 1988 ACM SIGMOD Conference*, Chicago, IL, June 1988. ACM.

[24] Teradata. Dbc/1012 database computer system manual release 2.0. Document C10-0001-02, Teradata Corp., Nov. 1985.

[25] K. R. Traub. A compiler for the mit tagged-token dataflow architecture. Master's thesis, Massachusetts Institute of Technology, 1986. Technical Report LCS TR-370.

[26] P. Trinder. *A Functional Database System.* PhD thesis, Oxford University, Cambridge, England, May 1989.