Experiences with Non-numeric Applications on Multithreaded Architectures

Angela Sodan*, Guang R. Gao+, Olivier Maquelin#, Jens-Uwe Schultz*, Xin-Min Tian$

*GMD FIRST
Rudower Chaussee 5
12489 Berlin
Germany
sodan@first.gmd.de
jus@first.gmd.de

+University of Delaware
140 Evans Hall
Newark, DE 19716
USA
ggao@eecis.udel.edu

#School of Comp. Scie.
McGill University
3480 University St.
Montreal
Canada, H3A 2A7
maquelin@cs.mcgill.ca

$IBM Toronto Lab.
1150 Eglinton Ave. East
Toronto
Canada, M3C 1H7
tian@vnet.ibm.com

Abstract

Distributed-memory machines have proved successful for many challenging numerical programs that can be split into largely independent computation-intensive subtasks requiring little data exchange (although the amount of exchanged data may be large). However, many irregular applications — e.g. in the AI field — have a fairly tight data coupling that often results from the use of shared data structures, making them in many cases not amenable to parallelization on distributed-memory machines. EARTH is an efficient multithreaded architecture that supports in particular large numbers of small data exchanges by means of low start-up times and the ability of tolerance of even small latencies. In this paper, we show the benefits provided by EARTH for applications of this sort by presenting experimental results from several AI applications run on the MANNA machine, which is a distributed-memory machine with a very high-performance communication network. EARTH-MANNA is shown to extend the range of programs that can be parallelized and run effectively on distributed-memory machines.

1 Introduction

EARTH-MANNA [13] is an efficient multithreaded architecture implemented on the high-performance distributed-memory machine MANNA [9]. EARTH is fine-grained in the sense that it provides communication by remote load/store operations and multithreading at a level below the function body with very fast thread switches allowing even small latencies to be hidden — considered the most difficult goal in Culler's cost estimations [7]. EARTH-MANNA additionally provides fast block transfer, thus allowing loosely coupled numeric applications (involving rarely occurring communications) to be run efficiently, too. EARTH's real benefits, however, are demonstrated in applications with a more critical computation / communication ratio, especially when there are very many small communications. The software overhead (especially the start-up time, which — for short messages — in conventional message passing is much larger than the real transfer time) is reduced in EARTH to a minimum, the potential for hiding latency being maximized. EARTH is, then, less sensitive to imperfect data distributions [21] and it provides benefits for numerical applications with complex irregular data structures that are hard to partition and in some cases may even evolve dynamically. Because start-up time is low and the number of communications is of limited significance, optimizations for combining several small messages to a large one become obsolete in many cases, thus also reducing the demands on the compiler with respect to communication optimization. In contrast, tests showed that it may even be advantageous to split large messages in order to increase the possibilities of overlapping computation and communication [21].

Furthermore, fine-grained communication and multithreading is of particular benefit for applications with dynamically created computational tasks, asynchronous mutual communications between tasks, irregular unpredictable data accesses, and — perhaps the most important aspect — frequent small communications, such as occured especially in the AI field in the following forms:

- tree-like dynamic task structures, appearing in many search problems

  They often have fairly small parallel computation steps, require dynamic task creation, asynchronous communication, and efficient dynamic load balancing, but by keeping overhead low and grouping several steps to threads / processes, there is potential for massive parallelism.

- computations with shared data structures, appearing in many reasoning- or transformation-based applications

  The data structures may potentially be linked, created dynamically, with possibly only part of them being accessed by each node. Because data is created dynamically, it is hard to be distributed by the compiler. Accesses are unpredictable, irregular, and highly asynchronous. The task-internal control structure is complex, and computation time may vary significantly for each parallel task. Thus, static assignment and static grouping or dynamic aggregation of communications to form larger messages (or prefetching) are difficult.

- fine-grained data-parallel and closely interconnected computations, as in artificial neural networks

Very low-cost communication / synchronization and subtle latency hiding are the only chance for parallelization.

- indeterministic application behavior with respect to computation time

  Some types of applications — and not only discrete optimizations for which this effect is well known — semantically permit arbitrary execution orders, but their execution time differs depending on them. This may lead to superlinear speedups in the best case, but in general it means a potentially wide spectrum of runtimes. Lower overheads minimize communication influences and thus should lead to more stable performance results, although the basic effect is inherently of an algorithmic nature.

The types of applications mentioned above may appear as isolated problems or as subproblems in more complex programs. Pure search problems naturally involve massive parallelism with large numbers of independent subtasks. If overhead is kept low, we can expect close-to-optimal speedup, and for some applications of this sort it has already been obtained in other systems [16]. Applications based on complex reasoning or transformations and using shared data structures are more challenging — especially on distributed-memory machines — because the control structure inherently contains many dependencies and the shared data easily becomes a bottleneck. Although sharing does not necessarily mean central management and a partly distributed organization is often possible, the inherent degree of parallelism is usually limited, i.e. it is not massive, exploiting up to several thousand processors. However, such programs are often very time-consuming (in the range of hours), and even a speedup of 5 or 10 is therefore a great achievement. Furthermore, in complex programs several such centers of parallel code may be nested in a tree-like manner, thus ultimately allowing sufficient overall parallelism to be obtained. Nevertheless, what is needed are systems that keep thread / process management and communication overhead to a minimum.

We present applications of the classes mentioned above and explore the usability and benefits of multithreading. More specifically, we demonstrate

- the influence of overhead and the difference made by using multithreading as compared with existing message passing or abstract-data-structure libraries
- that some applications become amenable to parallelization only if overhead is kept extremely low.

Two of the applications presented (Eigenvalue and Gröbner Basis) are taken from Multipol [22], which is an interesting library supporting some important shared-data abstractions related to scheduling structures or partial/final result structures. Multipol is implemented using the TAM multithreaded system and was first run on a CM-5 — also a distributed-memory machine. However, providing abstract data structures at library level naturally involves an overhead that may be too high for this critical type of application. We make direct use of EARTH's thread and communication operations, investigating the benefits of this implementation. The third application is a feed-forward artificial neural network with a very critical ratio of computation and communication, thus being at best parallelizable if communication overhead is kept extremely low, as in EARTH. We show that these irregular applications can be well supported by multithreaded architectures, and that the range of applications that are parallelizable on distributed-memory machines can be extended in this way.

In Section 2, we describe in detail the EARTH-MANNA system. Section 3 presents performance results for three applications: Eigenvalue, Gröbner Basis, and feed-forward neural networks. Section 4 discusses the relation to other work, and Section 5 gives a summary.

2 The EARTH-MANNA System

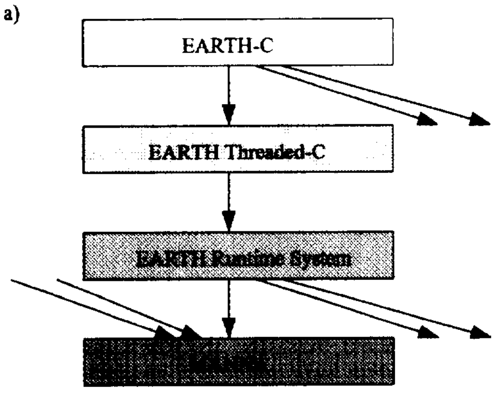
EARTH (Efficient Architecture for Running THreads) is a runtime system supporting a fine-grained multithreaded program-execution model. The code within a function body is subdivided into threads that are scheduled using dataflow-like synchronization operations. Threads are non-preemptive, i.e. once started, they run to completion. Threaded functions and threads in the threaded functions can be invoked remotely; threads are the schedulable entities in the EARTH system. The system maintains a global address space, and communication is performed by remote load/store operations and is thus at a very fine-grained level, too. Overheads for thread scheduling and communication start-up are in the range of a few microseconds (i.e. in the range of a few tens of instructions in the implementation of the corresponding operations). Thus, overlapping even small latencies is possible and worthwhile. EARTH also provides an efficient dynamic load-balancing facilities (using a work-stealing mechanism). For details, see [13].

EARTH was first implemented on MANNA, but it is currently being ported to other machines like the IBM SP2 and a cluster of SUN machines connected via a Myrinet switch. MANNA is a high-performance and low-cost distributed-memory machine. Each node contains two Intel i860 XP RISC CPUs, 32 MB of local memory, and a 50 MB/s bandwidth communication network realized on the basis of hierarchically organized crossbars. Detailed descriptions can be found in [9]. EARTH comes in two versions: a two-processor configuration per node, with one processor (Synchronization Unit) performing the specific EARTH operations and the other (Execution Unit) executing the basic application code; and a single-processor version executing all the code together. Both versions were shown to provide much the same efficiency with the existing smart single-processor implementation [18]. All tests shown in the following sections were run on MANNA with the single-processor EARTH version.

EARTH programs can be written in either EARTH-C or EARTH Threaded-C, both being extensions of C (see Figure 1 a). EARTH Threaded-C is the basic language dealing explicitly with threads and remote data accesses. EARTH-C is supported by the McCAT compiler [12] and hides remote data accesses and thread handling, i.e. it translates programs written at an abstract level (tree-like parallelism with communication being hierarchical between parent and children but not taking place between siblings) into multithreaded code. It is thus more convenient to use, but it currently supports only one specific programming model, whereas Threaded-C offers considerable flexibility in this respect.

Figure 1 b shows a piece of EARTH Threaded-C code.

a)



```
b) THREADED vadd (SLOT *done, int size, double *a,
                  double *b, double *result)
   { SLOTS SYNC_SLOTS[2];
     int i; double la, lb;

     INIT_SYNC(0, 2, 2, 1);
     INIT_SYNC(1, size, size, 2);

     for (i=0; i< size; i++) {
          GET_SYNC_D(a++, &la, 0);
          GET_SYNC_D(b++, &lb, 0);
          END_THREAD ();
   THREAD_1:
       DATA_SYNC_D(la+lb, result++, 1);
     }
     END_THREAD ();
   THREAD_2:
     RSYNC(done);
   END_FUNCTION();
   }
```

Figure 1: The EARTH-MANNA language/machine hierarchy (a) and an example of EARTH Threaded-C code (b).

Functions can be called remotely by using either INVOKE (explicit node assignment) or TOKEN (subject to automatic dynamic load balancing), and several ones can be simultaneously active on a node. THREADED indicates that the function contains multiple threads. Threads are labeled by THREAD_*n*; the first thread is started at function entry, and END_THREAD informs the runtime system to schedule the next ready thread. Remote data accesses are split-phase transactions. GET_SYNC_*x* and DATA_SYNC_*x* are remote read and write accesses, respectively, and are defined for different types of data. The third argument of these operations specifies a synchronization counter that is initialized with INIT_SYNC and decremented on completion of the operation. The *index+1*-th thread is associated with the counter and ready to run when the counter is zero. The example in Figure 1 b demonstrates the syntax of EARTH Threaded-C and how split-phase transactions are used. Two input vectors are added and the resulting vector is returned. *Vadd* fetches the *i*-th elements of the vector input-operands in parallel, then — again all in parallel — writes the *i*-th result value and reads the *i+1*-th elements. *Thread_2* is scheduled when all elements have been processed. RSYNC signals the end of the function by decrementing the counter *done*. Note that this small example does not, however, already demonstrate the overlapping of computation and communication. In more realistic code, in many cases there are other threads performing local computation being performed while load or store operations are under way.

Note that in the rest of the paper, we use the term 'task' for entities at application level that logically constitute parallel work, and 'thread' for entities at program level which would really be executed in parallel at architectural / system level.

3 Performance Results

3.1 Eigenvalue

Eigenvalue is an example of the important class of search applications. Other search applications (Protein Folding — finding all possible polymers of a specific cube, Paraffins — enumerating all distinct isomers of paraffins up to a certain size, or TSP — computing the optimal route for a traveling salesman through a certain number of cities) have already been shown to parallelize very well on EARTH-MANNA [13]. This class of applications is also commercially relevant in the still emerging application field of job planning in industrial environments.

- Application characteristic: Search applications create a tree of search nodes and are easy to parallelize in terms of the algorithmic structure. However, the overhead involved is critical, because the individual search nodes (basically constituting the potential parallel tasks) represent a small amount of work only and the tree unfolds dynamically and usually with an irregular shape, this then requiring dynamic load balancing.

We used the eigenvalue calculation algorithm of ScaLAPACK based on bisection and the basic parallelization of Multipol. The basic algorithm takes a symmetric tridiagonal matrix and can compute an initial range on the real line containing all eigenvalues. Furthermore, for a given real number, it is possible to determine how many eigenvalues are less than it. The eigenvalues are then found by approximation, successively subdividing the real line (in a binary way) until the intervals containing eigenvalues — and finally constituting the solutions — are determined to the desired accuracy. Thus, this application creates a dynamic search tree.

The tridiagonal matrix is replicated on each node, and only interval boundaries need to be communicated. Search nodes consume sufficient time so that threads can be created (i.e. parallel code entities be specified) for all search nodes in the tree, i.e. no grouping of search nodes in threads is applied (which is necessary in many applications of this sort to create sufficiently coarse-grained and cost-effective threads). EARTH-C is used for parallelization, and the parallel code entities are specified explicitly at EARTH-C level (i.e. we did not attempt automatic detection of parallelism). The scheduling structure and the approximate solution coincide here. Eigenvalues are not equally spread but clustered, which means that the tree is irregular and EARTH's dynamic load balancing has to be applied.

	1000x1000 matrix
problem size (original sequential version)	7310 msec
number of tasks created (= number of search nodes)	935
argument sizes	3 integers and 2 doubles (4*3+8*2 = 28 bytes)
mean computation time per step	7.82 msec
depth of leafs	1 to 22 (most between 18 and 22)

Table 1: Characteristics of ScaLAPACK Eigenvalue algorithm.

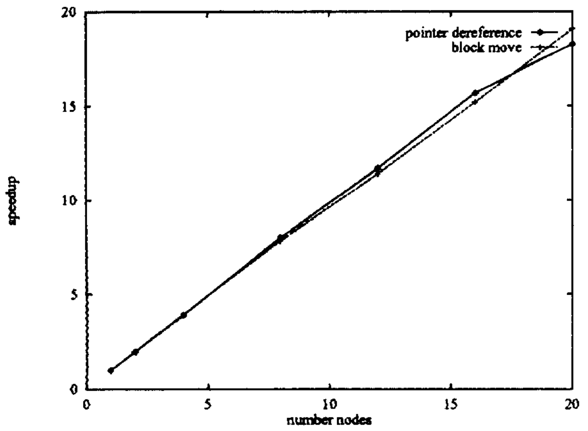


Figure 2: Speedups for Eigenvalue calculation by bisection; time in msec (speedups are relative to original sequential version).

The argument to each thread is a small structure (see Table 1). We tried using both individual remote accesses to the structure elements (by pointer dereferencing) and a block move for fetching the whole structure at once. The differences in runtime proved to be insignificant (see Figure 2). For the former version, the McCAT compiler automatically inserts all the threads necessary for exploiting the latency-hiding potential with respect to the remote data accesses. Speedups are close to optimal, i.e. communication, thread creation and load balancing overhead are in the range of microseconds and do not significantly influence execution time. Furthermore, the speedup is significantly better than in the Multipol version on a CM5, speedup there being only about 8 on 20 nodes. With the granularity of the tasks being still about 8 msec, a communication overhead of 0.5 msec per task would still, however, "only" reduce ideal speedup from 20 to 18.5 on 20 nodes. Thus, the random work distribution applied at thread creation time in the Multipol version probably contributes to the low speedup by not achieving a sufficiently good load balance. On the other hand, load balancing should not create much additional overhead, and the results show that EARTH-MANNA really did not, i.e. was successful and efficient in this respect.

3.2 Gröbner Basis

Gröbner Basis is another application taken from the Multipol library and thus already parallelized for distributed memory. Gröbner Basis is a computer-algebra problem, symbolically performing Gaussian elimination by transforming a set of polynomials into another set with the same roots, which are, however, easier to compute. The new set (solution set) is analogous to a triangular set of equations that are solvable by substitution. Gröbner Basis computation thus has applications in solving systems of nonlinear equations [5]. The algorithm works by forming so-called critical pairs, taking two polynomials of the already existing ones. Then, a new polynomial (a so-called S-polynomial) is calculated from them and simplified (reduced) by subtracting multiples of other polynomials. Polynomials that do not reduce to zero (i.e. are not a linear combination of existing ones) are added to the new set. The order of creating and processing pairs has a significant impact on the overall amount of work to be done, and thus on performance, a good selection heuristic being essential. Using the algorithm, the original set is extended by further polynomials until no more irreducible ones can appear and the result constitutes a Gröbner Basis. The algorithm is thus a completion procedure, i.e. the new set represents a partial solution at any point in time. The basic completion procedure is typical for many other AI applications, i.e. it forms a typical programming pattern. For example, the Knuth-Bendix algorithm (also investigated in [22]) used in theorem provers operates similarly on rewrite rules (but at a finer level of granularity that is also hard to parallelize on shared-memory systems).

Performing the reductions is the algorithm's main job, and the critical pairs thus constitute the work to be done. The algorithm is parallelized at this level, i.e. several "good" pairs are processed in parallel. If any of these computations yield irreducible polynomials that have to be added to the solution set, the first ready one is inserted by acquiring exclusive access to the basis. Because global irreducibility is required, the next candidates need to be checked for reducibility again before potentially being inserted, too. Nodes failing to acquire access to the solution set perform further reductions and try again after having finished the next reduction. Thus, waiting times are overlapped with computation at the algorithmic level — which would not have been available to automatic restructuring. Runtimes per reduction vary significantly (potentially by several orders of magnitude, see [1]).

Both the solution set and the pairs are shared data structures (see Figure 3). The pairs queue is implemented as a distributed shared data structure, i.e. each node maintains its own queue, but special interface functions allow inter-node access for adding entries to other nodes' queues. At each node, one main thread is running, picking up its work from the local pairs queue that is ordered by priority of "goodness". Pairs are constructed from the input polynomials or dynamically from the input and the new polynomials that have been added. The latter are created asynchronously and in varying numbers per node, and are thus subject to dynamic load balancing which is performed by means of a simple receiver-initiated ring distribution (moving pairs as long as a node is found on which the referenced polynomials are already cached — avoiding waiting times on the transfer of potentially large polynomials). Our experiments showed that adequate load balancing can be achieved with this simple strategy. The decentralized maintenance

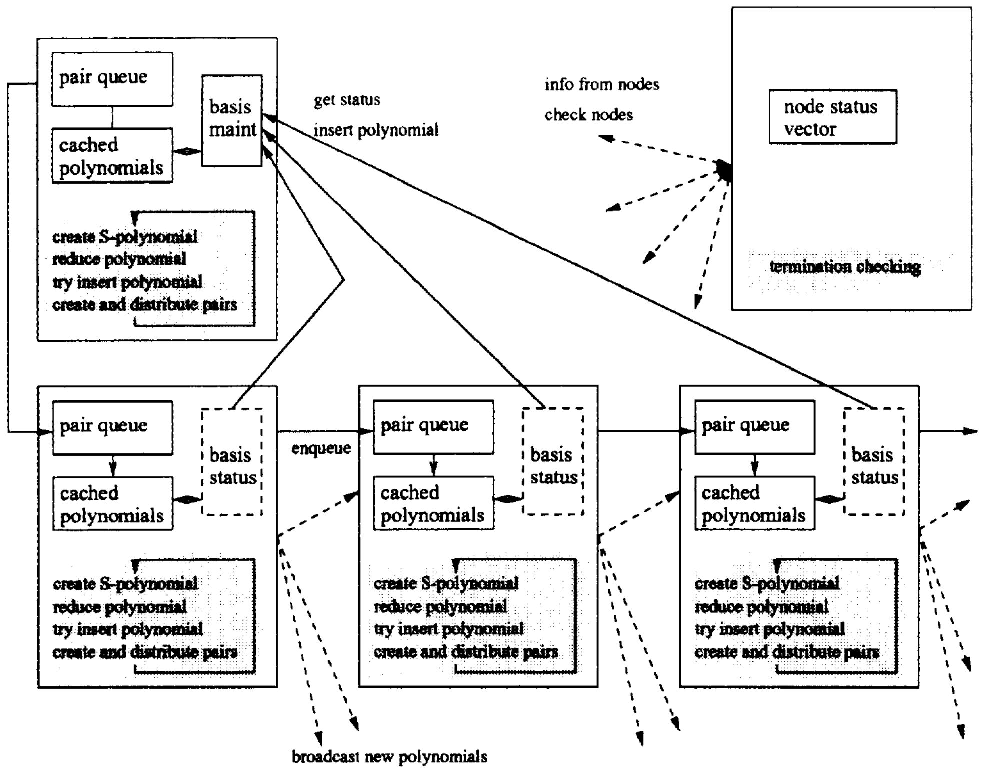


Figure 3: Main data structures and operations of Gröbner Basis algorithm.

of the queue prevents it from becoming a bottleneck, thus offering better scalability, but priorities are only maintained locally and thus nodes do not necessarily work on the globally best pairs. The solution set is implemented by some central maintenance information about its state and by full replication (read caching) of the polynomials. The solution set has only a few entries that potentially need to be accessed by all nodes. Accesses are read accesses, because the algorithm never deletes any polynomial from the solution set (this is not the only approach, but it simplifies the procedure, and Chakrabarti and Yelick [5] say they noted no significant disadvantage). The polynomials are represented in a compacted form as vectors. For access to the solution set, a lock has to be acquired.

- Application characteristic: Tasks are created dynamically and consume varying amounts of runtime. Thus, dynamic load balancing is to be applied. There are shared data structures, one of them being implemented in a distributed way and the other in a central way (with replication of the entries). The programming style applied is having one main application thread per node, executing the tasks. Threads on different nodes operate and communicate asynchronously (because of the varying execution times for the tasks). Communication is mutual between nodes, and mostly serves to access the shared data structures. The application is dependent on the execution order (with respect to work performed and thus runtime) and is thus intrinsically indeterministic in its parallel behavior.

As we have already said, the order in which the pairs are processed has significant impact on the amount of work that ultimately has to be done. Parallelization changes the order in which the pairs are processed and thus may influence the amount of work to be done. Parallel execution can then, in principle, be superlinear as well as performing very badly. Potential superlinear speedup by mutual utilization of the partial solutions of other concurrent solvers is a phenomenon that is observed and exploited in several AI applications. We obtained superlinearity in several test runs, and [1] recently discovered the superlinearity effect for the Gröbner Basis computation, too, discussing the phenomenon in depth and providing many results of this type. The dependence on the order in which the pairs are processed makes the application intrinsically indeterministic (on shared memory, too), because subtle time differences in the execution may change the order of accessing the solution set. Our experiments showed significant differences in processing time, as can be seen from Figures 4 and 5. Although most results are close to the mean, some vary by a factor of up to 7 for different runs with the same input.

We adopted the algorithm in its essential structure from Multipol, but simplified the code that maintains the shared solution set. The set is extended only and thus a simple vector is sufficient for its implementation whereas a hash table and a more complex maintenance are required for the generality provided by Multipol's data structure. Furthermore, originally the maintenance was performed in a distributed way, and is now centralized as explained above. (Because of the simple accesses and the fast communication, the central maintenance does not create a bottleneck effect.) This avoids one broadcast when a new polynomial is inserted. Our simplication of the data structure and the central maintenance in addition enable to release the lock earlier when

	Lazard	Katsura-4	Katsura-5
problem size	3761 msec	6373 msec	362750 msec
number of tasks created (= number of pairs)	141	75	168
number of polynomials in solution set	3 as input 27 added for completion	5 as input 15 added for completion	6 as input 26 added for completion
mean computation time per step (time per step varying significantly)	26.7 msec	85 msec	111.86 msec
mean size of polynomial	454 bytes	439 bytes	3243 bytes

Table 2: Characteristics of the Gröbner Basis application (sequential run), all inputs dealt with in total lexicographic order.

performing changes to the solution set. As a basic change, of course, the operations referring to TAM or the CM5 were to be replaced to use the EARTH system. Because of the mutual communication, we used EARTH Threaded-C for the implementation. To allow evaluation of the results obtained, we extracted the characteristics of the sequential algorithm, as shown in Table 2. More specifically, communication in this algorithm is applied for

- distributing pairs,
- caching polynomials, i.e. broadcasting them to all nodes,
- centrally making known the change to the set of polynomials (maintenance at one node),
- acquiring the lock (the lock data structure is maintained on one node, though performing normal processing, too), and
- detecting termination (performed by one node that is currently not doing any other useful work).

The EARTH multithreading model supports asynchronous processing and communications very well. The specific EARTH operations used for the communication purposes described above are

- block moves for transferring the polynomials
- individual — synchronizing — data load and stores for centrally making known changes to the solution set, for obtaining status information about the solution set, for releasing the lock, and for setting the termination flag per node
- remote function calls (invokes) for broadcasting the information about changes to the solution set, for distributing pairs, for acquiring the lock, and for central global termination checking.

Invokes are used at some points because they enable several arguments to be passed together and computations on them to be processed within a noninterruptable thread. (The original Multipol version already used remote invocation of Multipol threads — being somehow in between EARTH threaded functions invocations and threads — for mutual communication.)

The main benefits of EARTH for this application are the short communication start-up times. Furthermore, latencies for block moves and remote invokes are overlapped with local computation. However, the major potential for overlapping communication and computation is exploited here at the higher algorithmic level — where the necessary semantic knowledge is available. Figures 4 and 5 show performance results. The mean speedups (Figure 4 a) are mostly ideal for low numbers of nodes, but become increasingly sublinear for larger numbers of nodes. The exact limit of obtainable speedup depends on the problem size, i.e. is about 9 on 11 nodes for Lazard, about 12 on 12 nodes for Katsura-4, and about 12.5 on 14 nodes for Katzura-5. Thus, for larger problem sizes we can expect to obtain greater speedups, i.e. the application is scalable to some extent. The overall communication cost is fairly small, and the reason for the limited speedup, is therefore, probably the limited inherent degree of parallelism. It may also be due in part, however, to the simple method of work distribution, although our tests showed no significant idling times. A limited degree of parallelism would, however, not be surprising, because processing based on logical dependencies is largely sequential (similar limited speedups for such kinds of AI problems were obtained in [20]). Figure 4 b shows maximum and minimum values, too, demonstrating the inherent nondeterminism of the application.

To evaluate the above performance results obtained on EARTH, we have to compare them to more costly communication. This is especially important because the ratio of computation time to the number of communications is not too bad here and the positive influence of a low communication overhead is therefore not obvious. However, because the selection heuristic plays a significant role and the indeterminism creates a wide range of results, speedups obtained in other execution environments (like Multipol) would only be comparable with exactly the same application configuration and the same test-evaluation criteria — but these are not known for the other systems. We therefore "simulated" higher communication cost by artificially increasing communication times to 300 μsec, 500 μsec and 1000 μsec at both sender and receiver side for synchronous communication, and to only 150 μsec, 250 μsec, and 500 μsec at the sender side if asynchronous communication can be used. Furthermore, cost for copying to and from a message buffer is added. The times can be considered to approximately reflect the cost of efficient OS-specific message passing and of standard-library message passing (like MPI). It also gives an indication about the appropriateness for workstation networks where basic transfer cost is higher. "Messages" are assumed to be immediately accepted (i.e. there is not any further delay because of a busy receiver). Broadcasts are assumed to be sent in sequence. The "simulation" and the modeling of message-passing systems as described can, of course, provide a very coarse approximation only. Results are shown in Figure 5. For low numbers of nodes there is

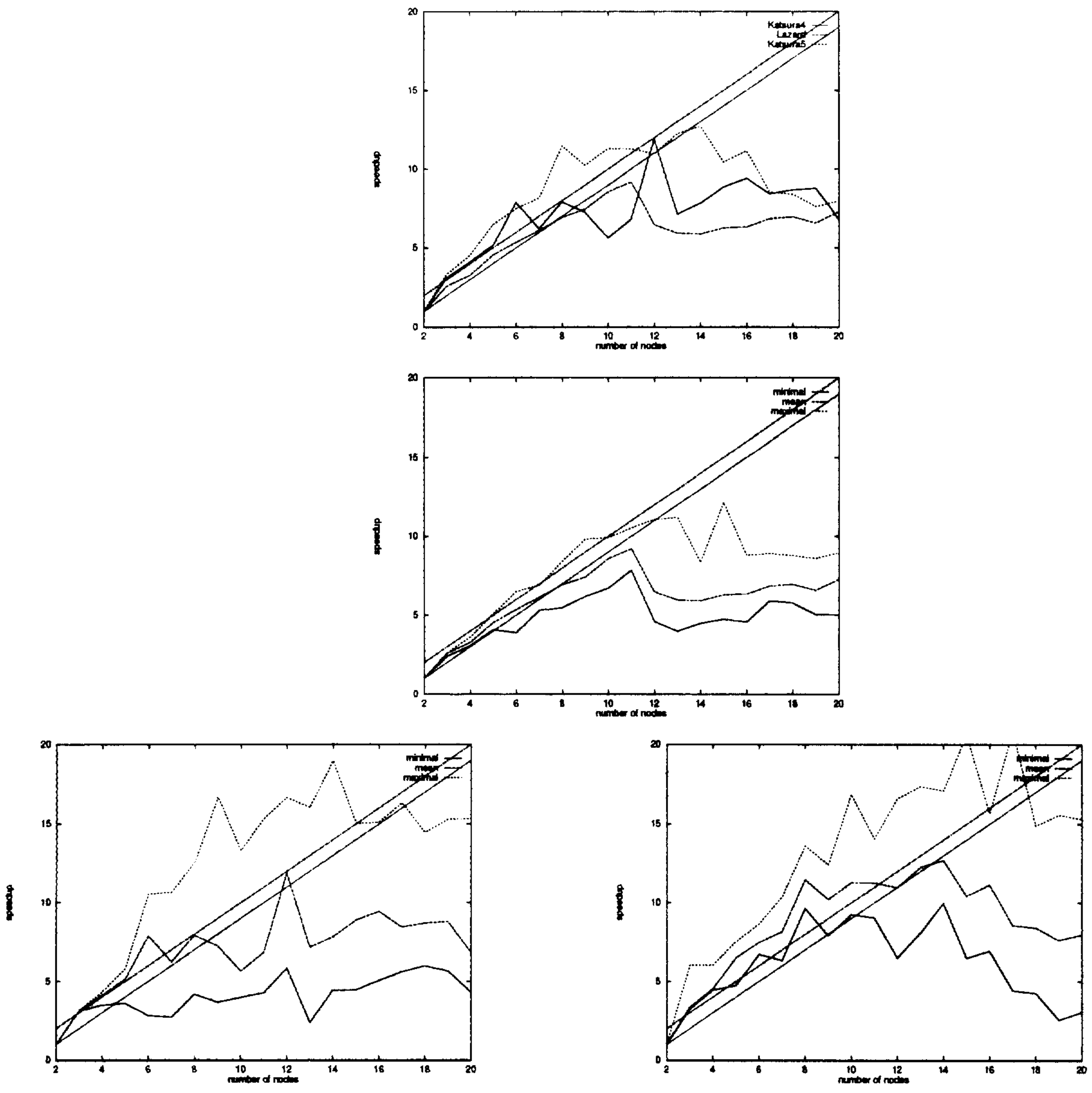


Figure 4: a) Mean speedups for Gröbner Basis with input basis Lazard, Katsura-4 and Katsura-5. b) Mean, minimum, and maximum speedups for Gröbner Basis and inputs Lazard (middle), Katsura-4 (bottom, left), and Katsura-5 (bottom, right). Speedup values are calculated on the basis of 20 test runs. Note that one node is reserved for detecting termination; the upper linear line represents ideal linear speedup taking into account this node that does not contribute to the computation; the lower line represents ideal linear speedup ignoring it. The reason for showing the latter is that there is some hope of improving the parallelization by letting the termination-detection node perform other work as well.

not much difference to the EARTH version, but the latter scales much better. In other words, there is a medium range of numbers of nodes — for which parallelism is still available in the application — in which EARTH allows the exploitation of this parallelism, higher communication overhead as "simulated" demolishing this potential. However, Katsura-5 still performs well for 300 μsec and 500 μsec because its granularity is quite high. In the 300 μsec case, Katsura-5 is even faster, which can only be explained by the pair-selection heuristic creating this astonishing effect. For the examples with a smaller computation/communication ratio, EARTH clearly shows its superiority.

Figure 5 also shows minimum and maximum values for checking our expectation of a smaller spread of values. For Lazard, this expectation is met (in the range of mostly linear speedup, the spread of values is much smaller for the EARTH version, especially when compared with the 300 μsec and 500 μsec version), but for Katsura-4 it is not (spread is greater with EARTH). For Katsura-5 no specific conclusion can be taken in this respect. More tests are therefore needed to obtain reliable conclusions with respect to the effect on spread.

To summarize, we can conclude that for a limited number of machine nodes, chosen in correspondence with the

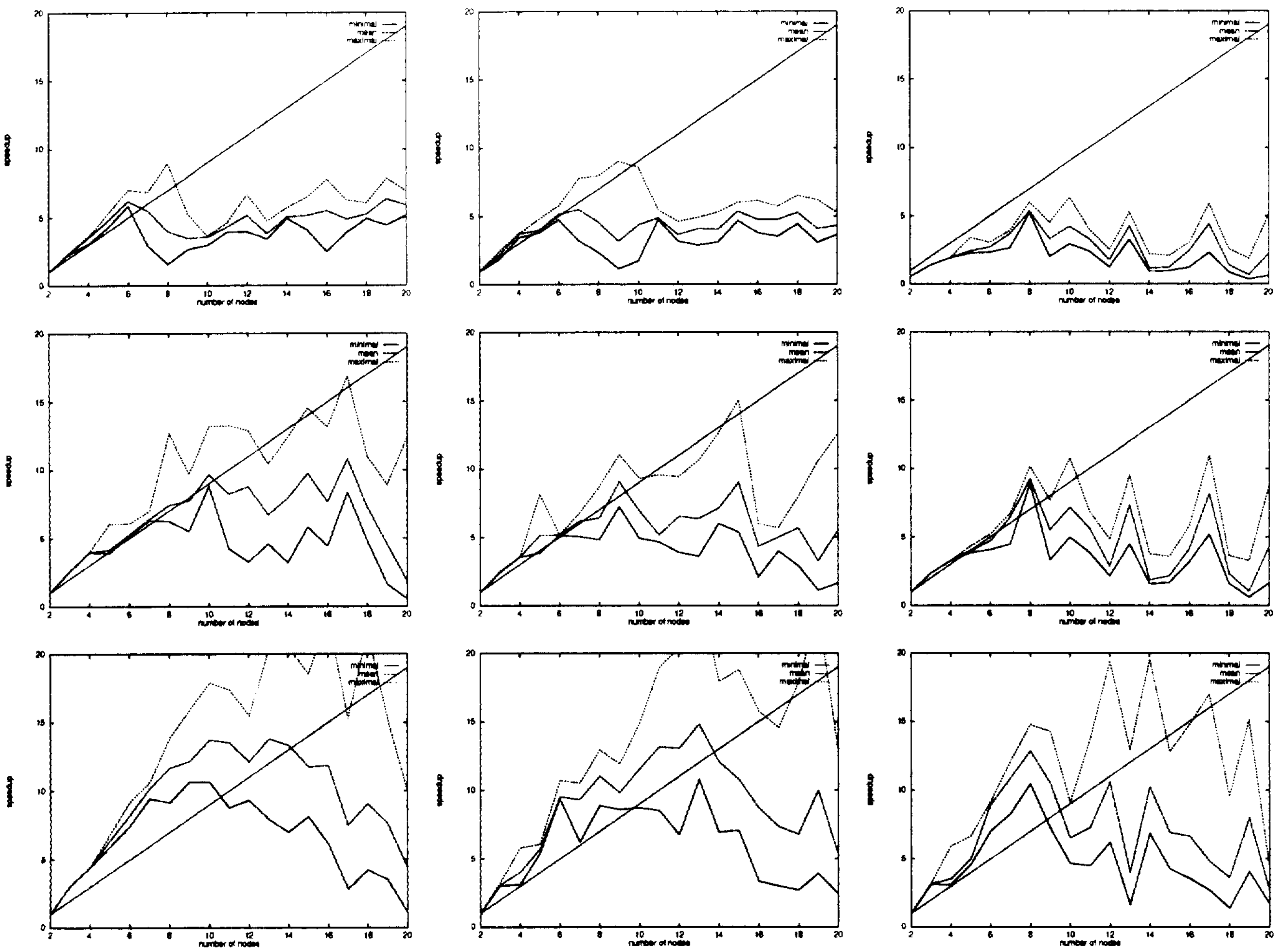


Figure 5: Mean, minimum, and maximum speedups of Gröbner Basis with input basis Lazard (upper row), Katsura-4 (middle row) and Katsura-5 (lower row) for different communication overheads (300 μsec, 500 μsec, 1000 μsec from left to right).

problem size and the inherent degree of parallelism, good speedups can be obtained for Gröbner Basis on distributed memory when using the EARTH multithreaded architecture, whereas the exploitable degree of parallelism is lower for systems with higher communication overhead (message passing via a specific operating system or a standard library). For larger problem sizes as Katsura-5 with a better ratio of computation to communication, the obtainable speedup is even higher, but the difference to higher-overhead communication systems then is less significant.

Because the completion procedure is a typical AI problem, multithreading can be expected to bring even greater benefits for specific applications of this type that have a worse ratio of computation to communication.

3.3 Neural Networks

Artificial neural networks are an application area of increasing practical use. They come in two major forms: feedforward and recurrent networks. The former are more widespread, and they are the ones considered here. The training phase in particular is very time-consuming and thus interesting as regards speedup by parallelization. There are two main levels for parallelization [19]: sample parallelism and unit parallelism. In the training phase, the net is fed thousands of test cases (samples), and sample parallelism means exploiting the inherent data parallelism and running several neural networks in parallel, each processing different subsets of the samples in batch mode (without any communication); only at the end of the training phase is information exchanged. Unit parallelism parallelizes the network itself. Because networks are not very large and their computations are very closely coupled (see below), the degree of parallelism obtainable is known to be limited. Here, however, updates of the net can be performed after each sample, and there are several investigations in the literature showing that such an approach usually converges faster than batch processing (i.e. the overall amount of work to be performed is then smaller). Thus, unit parallelism is interesting, too, and could, for example, be exploited in a hybrid approach, mixing unit and sample parallelism. This would support the frequently used hybrid approach to updating, namely repeatedly presenting small batches and performing an update after every batch (this — depending on the view — dividing an imaginary overall batch into several pieces or collecting several samples to one batch).

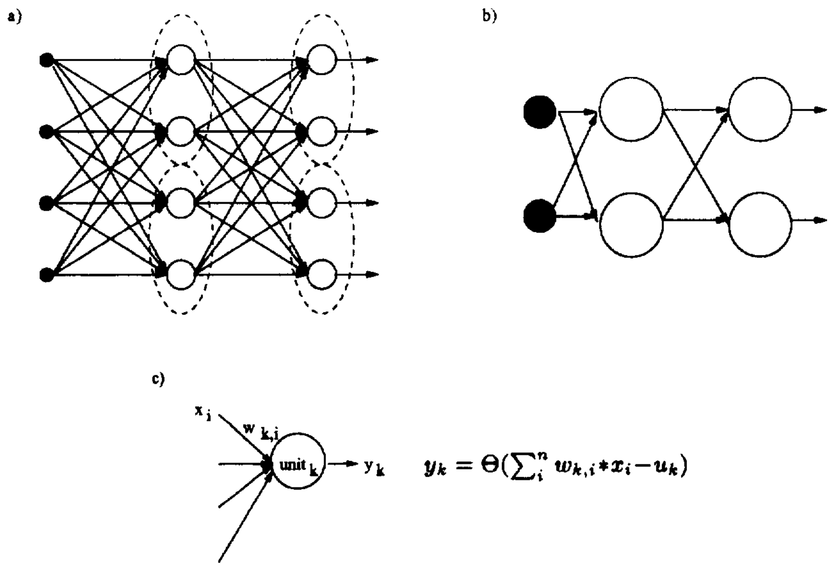


Figure 6: Basic principle of feedforward-neural-network computation with potential grouping of units (a), and structure resulting from partitioning units according to indicated potential grouping (b). c) shows forward-pass calculation per unit.

- Application characteristic: For unit parallelism in neural networks and the current sizes of networks, the amount of local work per unit is fairly low, whereas the amount of communication is very high (in the standard full linkage between the units of two adjacent layers). Thus, unit parallelism is at the very end of the spectrum of parallelizable programs, with a very critical ratio of computation to communication.

Artificial neural networks (see Figure 6 a) consist of a set of units (representing neurons) organized in several layers (typically 3: input layer, output layer and one hidden layer), and — in the basic case — all units of one layer are linked to all units of the next one. Each unit calculates a scalar vector product on the vector of input or previous-layer data and on a vector of local weights. Thus, although neural networks can be applied to the solution of non-numerical problems (like classification or pattern matching), they internally are a numeric problem (performing floating-point operations). For each sample, these calculations are performed at each unit per layer with an information flow from the input to the output layer. Afterwards, the weights are updated by an information flow in the opposite direction (backpropagation), realizing the learning of the net. Parallelization at the unit level means performing the unit calculations per layer in parallel. Typical numbers of units per layer in currently existing practical systems are between 80 and 200 [17]. Larger nets are not used, because of the computational cost in the sequential case, and because it is unclear whether larger numbers of units would really be of benefit; we did, however, add a test with 720 units per layer.

Several units can be grouped per machine node (which means "slicing" the layer), this increasing the amount of work per node while reducing the overall number of communications (see Figure 6 b). However, for the current sizes of networks, either the potential of grouping or the degree of parallelism is low. This is even true for shared-memory environments (involving proocess/thread management and synchronization overhead, too).

Table 3 shows the characteristics of the application. We tested first the forward pass separately, and then both forward and backward pass together inclusive update of the weights. Performance results are shown in Figures 7 and 8. Communication is centralized: all nodes receive the input data for the next layer from one central node and send the result back to it. This reduces the overall number of communications on the network, and at the same time synchronizes the layer computations. In order to further improve performance, a tree organization (as described in [6]) is used for the communication. In comparison to an ealier version using sequential communications, speedups increased — for 80 units from a maximum of 8 to a maximum of 12. We used 3 layers, resulting in 2 computation phases. The number of units is the same per layer, and full linkage is used. In the forward pass, communication takes place for a call of the computation phase per layer, a data transfer from hidden to output layer, and a data transfer from the output layer to the central node (usually calculating a global error value of the current net status then). The backward pass being added, there is another initiation of a computation phase (the forward and the backward computation at the output units can be combined) and one additional data exchange of error values from output to hidden layer. Communication is more costly in the backpropagation be-

units	sequential runtime	rutime per unit
80	5.047 msec	32 μsec
200	26.96 msec	67 μsec
720	319.1 msec	222 μsec

Table 3: Characteristic of neural network computation, forward pass (all computations using floats for the operands). Runtimes for forward and backpropagation together is about twice the time in our test settings.

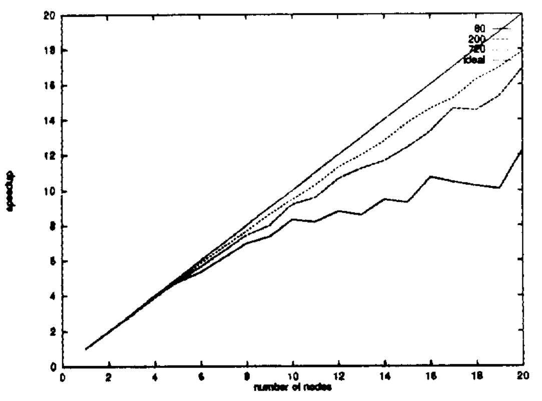


Figure 7: Speedup for forward pass only in neural network computation (using different sizes of networks and different numbers of machine nodes).

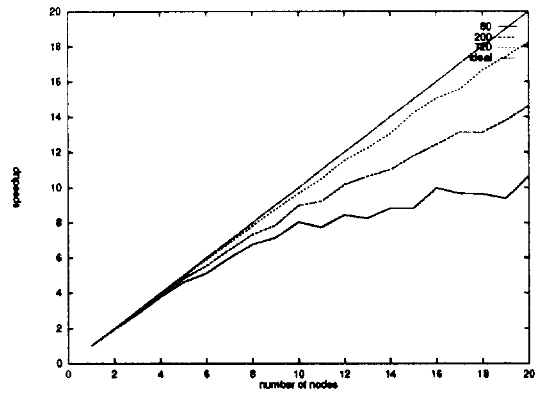


Figure 8: Speedup for combination of forward and backward pass in neural network computation (using different sizes of networks and different numbers of machine nodes).

cause there is not a simple broadcast of the same value, but different values need to be communicated to different units. EARTH Threaded-C is used for the implementation because we wanted to perform subtle tuning and experiments. The algorithm, however, in the organization described above, is transformed to a tree-like one. Except that some short-term data is collected and distributed centrally (the data flowing along the links), long-term data (the weights and accumulated errors) is maintained per node being exclusively used by the nodes and surviving the individual layer activations.

EARTH performs astonishingly well for this communication-critical application. For networks of the size currently in practical use (i.e. 80 and 200), speedups of 10 on 16 nodes for 80 units and of about 14.5 on 20 nodes for 200 units were achieved. When considering the forward pass only, speedups of 11 on 16 nodes (runtime 458 μsec) are obtainable for 80 units, and for 200 units per layer speedups of 17 are achieved on 20 nodes (runtime 1.59 msec). Achieving these speedups despite the high interconnection in communication exchange and the overall short runtime, demonstrates the effectiveness of EARTH's communication and that only systems with such a low overhead can lead to significant speedups for small-sized neural networks often being practically in use. Note that the network as described is just one basic instance of feedforward networks. The number of units may differ per layer and the linkage may be partial only. Furthermore, the Θ function (see Figure 6 c) was set to a quite simple one in our test and may be more costly, or the error calculation performed differently. Compilers for neural-network description languages [15] might perform the appropriate generation of EARTH code and then well exploit EARTH's potential for parallelizing even small nets.

4 Related Work

Instances of search applications — for example, chess [14] — have already been successfully parallelized on other systems, and there are several publications on problems of this sort (e.g. [16]). However, the lower the overhead in communication, dynamic load balancing and thread management, the more problems of this sort we can expect to parallelize effectively. Furthermore, in most tree-like search applications, the individual steps are very small, and thus a partitioning grouping several individual steps — even if overhead is kept very low — is additionally required, as discussed in several of the papers on parallel search and in [20]. This subject is also addressed in the implementation of functional languages [11]. Despite partitoning, in functional languages there is potentially still much fine-grained tree-like parallelism and functional languages, too, were shown to be supported by multithreading approaches [10]).

For Gröbner-Basis computation, there exist different approaches besides the one Yelick et al. have used here. For example, a recent paper of Amrhein et al. [1] proposes a very different approach, currently running on shared memory only. It is not, however, our intention to compare different Gröbner-Basis algorithms with respect to their effectiveness, but rather to show the influence of communication cost. The approach of Yelick uses an abstract data-structure library, providing convenience of use but inherently imposing extra overhead. Amrhein et al. also discuss the indeterminism effect in depth. For neural networks, most approaches choose sample parallelism, but some have experimented with unit parallelism, as reported in the summary given by [19]. However, no systematic performance tests are presented there.

For Gröbner, our comparison to message-passing cost is a bit pessimistic with respect to broadcasting data; some operating systems offer special optimized broadcasting operations (for example, [6] reports that broadcasting cost is reduced to be even less than logarithmic in the number of machine nodes by organizing communication as a special tree). Broadcasting cost makes up a significant amount of the overall communication overhead in Gröbner. On the other hand, we were quite optimistic assuming the receiver to handle the message immediately. At least, for EARTH it is less important to think about optimizations for specific communication structures, although for the critical Neural Networks we could achieve some improvement this way. Furthermore, some modern operating systems like PEACE [9] [4] or Nexus [8] offer more efficient communication for special cases. PEACE — requiring 210 μsec for a synchronous remote call (involving two-way communication) — provides asynchronous one-way communication, too. This could be exploited for some of the communications in Gröbner. Nexus provides — in addition to standard buffering messages — one-way messages in the form of so-called active messages, which avoid buffering, and are thus basically faster but lead to loss of safety. There are special features in PEACE, too, that allow us to avoid buffering for large messages and transfer data directly to the desired data space (requiring two messages then, however) — as this is the standard case in EARTH. Thus, both PEACE and Nexus are mixed approaches evolving to integrate features originally provided by fine-grained multithreaded systems, although still paying some price for their generality as an operating system. Message-passing libraries like MPI involve much higher overhead (being multiples of that of basic message passing), though they offer the significant benefit of portability. As in the case of the examples presented, the overhead is too high for applications with a high ratio of communication to computation. In general, message passing libraries such as usually provided by operating systems often provides — similar to abstract data-structure libraries — convenience of use and mechanisms for supporting safety (like checking access rights to data). These benefits, of course, have to be paid for by some cost. Multithreaded systems like EARTH are highly tuned for maximum efficiency, but require more support by a compiler, ensuring e.g. to some degree correctness of data accesses.

There are other multithreaded systems at the same fine-grained level as EARTH, e.g. TAM [7] or Cilk [3] which have slightly different features. They run on different machines, and it was therefore not possible to quantitatively compare their performance to that of EARTH for the examples shown, but our concern was to compare EARTH with coarser level systems anyway.

5 Summary and Future Work

EARTH was shown to effectively support applications that have a fairly critical computation / communication ratio and may additionally require asynchronous communication, dynamic thread creation, and dynamic load balancing. Thus, EARTH allows us to parallelize applications or application subcenters on distributed memory that otherwise are not parallelizable or with much less speedup. Experimental results are presented for an Eigenvalue massive search problem, the computer-algebra problem of Gröbner Basis computation, and unit parallelism in feedforward artificial neural networks. Gröbner Basis is an application belonging to the category of complex reasoning or transformations, involving shared data structures and strong internal control dependencies. Speedup here is inherently limited, but the speedups obtained — in the range of 5 to 20 — are nevertheless of great significance because of the potentially large problem sizes. Furthermore, there is a trend toward building more complex hybrid symbolic/numeric applications, and such computations may appear as subcenters of large programs. While isolated problems — because of their limited degree of parallelism — can also be run on shared-memory machines, for hybrid applications it is advantageous not to be dependent on static organizations with perhaps shared-memory subclusters, but to have the option of flexible partitioning and assignment of processing power to match the concrete application structure (although cluster organizations of machine nodes [2] impose differences in transfer time for inter- and intra-cluster communication, especially when the amount of data is large). Mixed programs with frequent small communications, on the one hand, and rare large communications, on the other, require effective support of both types of communications, as provided by EARTH's polling watchdog [18]. Moreover, in the Gröbner Basis we dealt with the runtime-indeterminism problem, which, however, showed further investigation to be needed. Unit parallelism in neural networks involves very intensive data linking, thus coming close to the limits of parallelizability, even on shared-memory architectures. Still significant speedup that makes it worth considering parallelization was obtained on EARTH.

Furthermore, the results show that when overhead is critical it is worth having highly optimized code. In other words, the overhead that has to be paid for the generality of libraries may then be too high just as the price that has to be paid for the convenience and generality provided by message-passing systems is not affordable in overhead-critical applications. It is therefore desirable to have compilers supporting typical computation/communication patterns other than tree-like structures, so that more specialized library routines can be used in a safe way or directly code be generated using more low-level mechanisms. Then both user convenience and efficiency can be provided and fine-grained multithreaded systems become fully accepted for practical use. The EARTH-C compiler is an important step in this direction.

Acknowledgments

Our thanks go to Gerd Kock, Klaus Müller, and Nikola Serbedzija of GMD FIRST for interesting discussions on neural networks. We also wish to thank Windfried Neun of the Konrad-Zuse-Zentrum Berlin who helped us check the Gröbner Basis results for correctness.

References

[1] AMRHEIN, B., GLOOR, O., AND KÜCHLIN, W. A case study of multi-threaded Gröbner Basis completion. In *Proc. ISSAC'96 (Internat. Symposium on Symbolic and Algebraic Computation)* (Zürich/Switzerland, July 1996).

[2] BASAK, D., PANDA, D. K., AND BANIKAZEMI, M. Benefits of processor clustering in designing large parallel systems: When and how? In *Proc. Internat. Parallel Processing Symposium* (Hawaii, April 1996).

[3] BLUMHOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEIERSEN, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An efficient multithreaded runtime system. In *Proc. 5th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '95)* (August 1995).

[4] BRÜNING, U., GILOI, W. K., AND SCHRÖDER-PREIKSCHAT, W. Latency hiding in message-passing architectures. In *Proc. 8th Internat. Parallel Processing Symposium* (Cancun/Mexico, 1994), IEEE Computer Society, pp. 704–709.

[5] CHAKRABARTI, S., AND YELICK, K. Implementing an irregular application on a distributed memory multiprocessor. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP '93)* (San Diego/USA, 1993).

[6] CORDSEN, J., POHL, H.-W., AND SCHRÖDER-PREIKSCHAT, W. Performance considerations in software multicast. In *Proc. 11th ACM International Conference on Supercomputing (ICS '97)* (1997).

[7] CULLER, D. E. Multithreading: Fundamental limits, potential gains, and alternatives. In *Multithreaded Computer Architecture - A Summary of the State of the Art* (1994), R. A. Iannucci, G. R. Gao, R. H. Halstead, and B. Smith, Eds., Kluwer Academic Publishers, Boston/London/Dortrecht, pp. 97–138.

[8] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Systems 37* (1996), 70–82.

[9] GILOI, W. K., BRÜNING, U., AND SCHRÖDER-PREIKSCHAT, W. MANNA: Prototype of a distributed memory architecture with maximized sustained performance. In *Proc. Euromicro PDP96 Workshop* (1996), IEEE-CS Press.

[10] HAINES, M., AND BOEHM, W. On the design of distributed memory Sisal. *Journal of Programming Languages*, fall issue (1993), 209–240.

[11] HAMMOND, K. Parallel functional programming: An introduction. In *Proc. PASCO* (Linz/Austria, 1994).

[12] HENDREN, L. J., GAO, G. R., TANG, X., ZHU, Y., XUE, X., CAI, H., AND OUELLET, P. Compiling C for the EARTH multithreaded architecture. In *Proc. Internat. Conf. on Parallel Architectures and Compilation Techniques (PACT '96)* (Boston/USA, October 1996), pp. 12–23.

[13] HUM, H. H., MAQUELIN, O., THEOBALD, K. B., TIAN, X.-M., AND ET AL. A design study of the EARTH multiprocessor. In *Proc. Internat. Conf. on Parallel Architectures and Compilation Techniques (PACT '95)* (Limassol/Cyprus, 1995).

[14] JOERG, C. F., AND KUSZMAUL, B. C. Massively parallel chess. In *Third DIMACS Parallel Implementation Challenge at Rutgers University* (October 1994).

[15] KOCK, G., ENDLER, M., GUBITOSI, M. D., AND SONG, S. W. Towards transparent parallelization of connectionist systems. In *Proc. 9th Internat. Conf. on Parallel and Distributed Computing Systems (PDCS '96)* (Dijon/France 1996).

[16] KUMAR, V., GRAMA, A., GUPTA, A., AND KARYPIS, G. *Introduction to Parallel Computing - Design and Analysis of Algorithms.* Benjamin/Cummings Publ. Company, 1994.

[17] LECUN, Y., BOSER, B., DENKER, J. S., HENDERSON, D., HOWARD, R. E., HUBBARD, W., AND JACKEL, L. D. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1 (1989), 541–551.

[18] MAQUELIN, O., GAO, G. R., HUM, H. H. J., THEOBALD, K. B., AND TIAN, X.-M. Polling watchdog: Combining polling and interrupts for efficient message handling. In *Proc. 23rd An. Internat. Symposium on Computer Architecture (ISCA '96)* (Philadelphia/USA, May 1996).

[19] SERBEDZIJA, N. B. Simulating artificial neural networks on parallel machines. *IEEE Computer* (March 1996).

[20] SODAN, A. *Mapping Symbolic Programs with Dynamic Tree Structure to Parallel Machines.* Oldenbourg-Verlag, München/Wien, 1996.

[21] TIAN, X.-M., NEMAWARKAR, S., GAO, G. R., HUM, H., MAQUELIN, O., SODAN, A., AND THEOBALD, K. Quantitative studies of data locality sensitivity on the EARTH multithreaded architecture: Preliminary results. In *Proc. HiPC'96 (3rd Internat. Conf. on High Performance Computing)* (Trivandrum/India, December 1996).

[22] YELICK, K., CHAKRABARTI, S., DEPRIT, E., JONES, J., KRISHNAMURTHY, A., AND WEN, C.-P. Parallel data structures for symbolic computation. In *Workshop on Parallel Symbolic Languages and Systems* (October 1995).