McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

The Threaded Communication Library: Preliminary Experiences on a Multiprocessor with Dual-Processor Nodes

Nasser Elmasri
Herbert H.J. Hum†
Guang R. Gao

ACAPS Technical Memo 89
December 26, 1994

†Dept. of Electrical and Computer Engineering, Concordia University, 1455 de Maisonneuve W., Montreal, Canada, H3G 1M8.

Abstract

In this paper, we introduce a user-level threads package called the Threaded Communication Library which provides primitives (C functions) to support programs under a multithreaded program execution model. One primary feature of TCL, in comparison with some other conventional thread packages is that it provides a program execution model where inter-thread communication/synchronization is tightly integrated with computation. Thus, threading primitives as well as communication and synchronization primitives are "internal" to the computation model, thus should incur little software overhead. The TCL library is implemented on the GMD MANNA 2.0 multiprocessor in which one of the two processors in a node is dedicated to computations, while the other is used solely for communications, synchronizations and thread scheduling. This paper further presents a study on the benefits of a communication/synchronization co-processor for multithreading.

The preliminary experiments compared the performance of benchmarks using the dual-processor TCL implementation to a single-processor version in which all computations, synchronizations, and communications are performed on one processor of a node. Our results indicate that: 1) for applications with highly asynchronous fine-grain communication demands, the dual-processor implementation can effectively overlap communication and computation for faster execution; and 2) with more regular communication requirements, the role of caching for the computation processor cannot be ignored. In some benchmark programs in which the data to be communicated between processors can be kept on the on-chip cache of the computation processor, the single processor version of TCL may yield better performance due to the avoided cache misses which the dual-processor version encounters. These results may provide useful insights for the node architecture design of future generation multiprocessors.

Contents

List of Figures

1 Introduction

Writing programs which effectively utilize a parallel computer generally requires expressing the parallelism in an application, partitioning data to minimize communication, and distributing processes for the purpose of load balancing and minimizing the idle time of processors. Decisions in these tasks are ultimately affected by the time spent on communication and synchronization.

Much research has been devoted to reducing and hiding the the communication and synchronization overhead (parallel execution overhead). Reduction efforts tended strictly to involve hardware improvement and/or support, while hiding the overhead required the investigation of non-traditional program execution models such as dataflow and multithreading [DG94], and the addition of requisite hardware support and/or a fast software communication layer. Dataflow and multithreaded execution models 'piggyback' remote synchronizations with communications; thus naturally decreasing part of the parallel execution overhead. Also the smooth integration of computation and communication allow the interprocessor communications to be efficiently (e.g. with little software overhead) performed in a split-phase manner, i.e., a processor can issue a request and continue with another computation (thread) without waiting for a response. The response contains enough synchronization information to start up the computation requiring the data. Split-phase transactions allow communications to overlap with computations and thus hide part of overhead. Active messages [vECGS92] implemented on conventional parallel machines also adopted the piggybacking on communications technique and direct user-level communication primitives to show that decreasing the communication processing time can substantially decrease the execution time. The TAM (Threaded Abstract Machine) model [CSS+91] implemented on top of an active messages layer shows that split-phase transactions complement the benefits of active messages.

Many experimental and proposed architectures [KOH+94, RLW94, NPA92, HTG94] include dedicated hardware for communications and synchronizations such that the main processor can concentrate on computations without being detracted by such tasks.[1]

In this paper, we investigate the benefits of dedicated hardware support for communications and synchronizations under a multithreaded program execution model. Our experimental platform is a distributed memory machine called MANNA 2.0 [GMD93]. An interesting feature of the MANNA 2.0 (Massively Parallel Architecture for Numerical and Non-numerical Applications) machine is that every node has two high-performance i860 XP processors (similar to a general-purpose node in the Intel Paragon XP).[2] We built a *Threaded Communication Library* (TCL) for MANNA which provides primitives (library functions) for a multithreaded program execution model where the parallel programmer can specify threads of computation and their requisite synchronization and communication. Basically, TCL offers C functions for

[1] In [KOH+94, RLW94], the dedicated hardware is mainly used to keep caches consistent which ultimately requires fine-grain communications.

[2] This architecture reflects a current trend to have an extra processor per node that is responsible for non-computational tasks, e.g., Intel Paragon, Meiko CS-2, etc.

- fine- to coarse-grain communications;
- dataflow (producer-consumer) type and control-flow type of synchronizations (such as barrier synchronizations) between threads; and
- creating threads.

TCL supports a multithreaded program execution model where basic mechanisms for thread creation, deletion, and thread switching are tightly integrated with asynchronous events such as interprocessor communication/synchronization. That is, although the generation of threads is realized via library function calls, the functionality of threads in a TCL program is not "external" to the program execution model. Such a difference becomes particular important when application programs contains frequent, irregular, and small-grain communication requirements.

TCL on MANNA dedicates one i860XP to communications and synchronizations, and the other to computations. The design philosophy of TCL is to minimize communication and synchronization processing time as seen by the compute processor. This is achieved by off-loading most of the processing of these tasks to the second processor in the node. Any latency due to interprocessor communication and synchronization is to be effectively hidden by the coprocessor.

To measure the effect on performance generated by the second processor, we also implemented a uniprocessor version of TCL which performs computation, communication and synchronization tasks on a single processor in each of the processor nodes and compared its performance to the dual-processor version of TCL, i.e., the one described in the previous paragraph. We implemented some benchmark programs using the two versions, and collected data on a 16-node MANNA. The preliminary experimental results are interesting. In particular, they show that:

- For applications with frequent, irregular, and small-grain communication requirements, the dual-processor version gives better performance with respect to the single-processor one. The main advantage of the second processor is that it processes all the frequent interprocessor requests with minimal interruption to the main computation processor.
- On the other hand, for applications with more regular communication requirements, the role of caching for the computation processor cannot be ignored. For some benchmark programs in which the data to be communicated between processors can be kept on the on-chip cache of the computation processor, the single processor version of TCL may yield better performance due to the avoided cache misses which the dual-processor version suffers from. That is, the penalty of cache misses can outweigh the benefits of overlapping communications with computations, especially in applications in which the amount of computations performed per data communicated is low. For future node architectures which employ communication coprocessors, caching of shared data must be considered.

1.1 Synopsis

Section 2 contains a brief description of MANNA. Section 3 describes TCL and its implementation. Moreover, an active messages layer which the TCL uses will also be described in that section. Section 4 describes the benchmarks we implemented as well as the experimental results and their analyses. Lastly, Section 5 summarizes related work, and we detail future work in the Conclusion.

2 Overview of the MANNA Machine

MANNA [GMD93] is a distributed memory multiprocessor which has nodes connected via a fast communication network consisting of one or more crossbar switches; multiple crossbar switches would be configured in a hierarchy of switches. Figure 1 shows a block diagram of the MANNA node. Each node contains two Intel i860XP processors—resulting in 100 MIPS and 200 MFlops

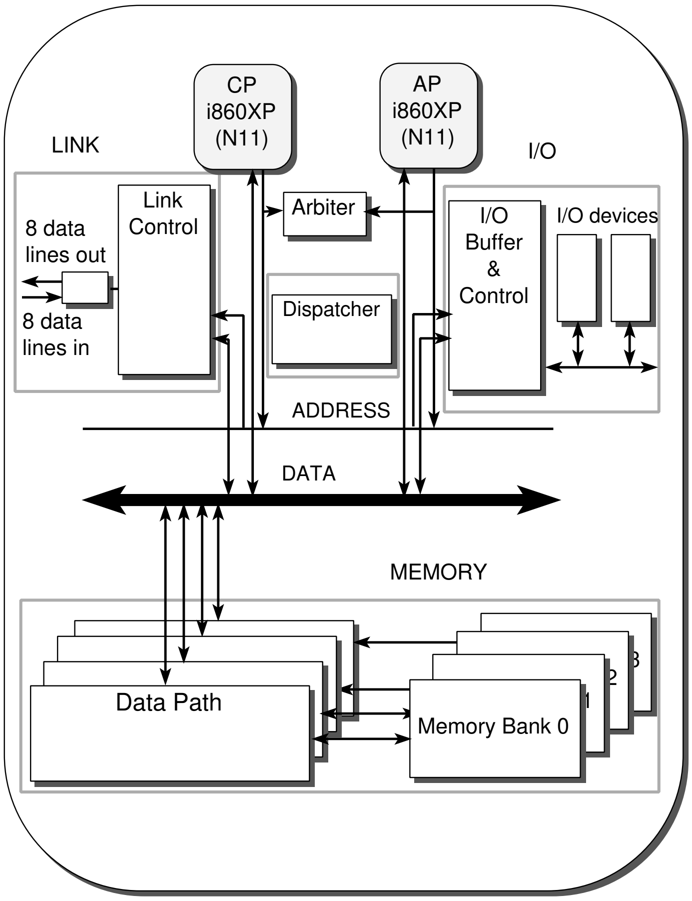

Figure 1: Block diagram of a MANNA node

peak performance—which share a common bus. A node has 32MBytes of memory organized as four interleaved banks with a peak memory access bandwidth of 400MBytes/s. The *Link* control chip which connects the node to the network contains send and receive FIFOs (of 32 words each) to buffer messages.

The MANNA interconnection network is realized by a hierarchy of 16×16 crossbars which utilize the link protocol [GMD93] for flow control and wormhole routing for routing messages. The MANNA link protocol is a clock-synchronous, byte parallel, bidirectional point-to-point connection working with a maximal clock frequency of 50 MHz. Each port may simultaneously support an arriving and a departing logical connection of 50 MBytes/s each (100 MBytes/s full-duplex).

3 Threaded Communication Library

In this section, we first describe the active messages layer in which the Threaded Communication Library (TCL) is built on. Our active messages layer is based on the concept of the same name from Berkeley [vECGS92]. Then we describe the primitives of TCL and some important implementation details. Lastly, we show the use of TCL via an integer sort program.

3.1 Active Messages Layer

The Active Messages (AM) layer offers a basic and minimal layer of communication on top of the network hardware, i.e., no system call overhead is incurred. Minimizing the overhead of sending a message permits the use of finer grain communications than those offered in traditional message-passing routines. This is beneficial for applications which cannot or do not want to cluster data into large chunks for communications. The major idea of active messages is that a message carries along with the data, a pointer to the handler that will integrate this data with the computation on the receiving end [vECGS92]. This permits the introduction of split-phase transactions which are basically non-blocking sends and receives, and allows the AM layer to overlap communications with computations.

Our AM layer was implemented in assembly language to get the best possible performance. It is offered to the user (compiler or library writer) as a library of C functions. The main characteristics of this layer are:

- It avoids deadlocks by always being ready to accept messages from the network. This is done by frequently polling the network for messages received. Interrupt-driven mechanisms to signify message arrivals were not used due to the excessive number of cycles required in interrupt processing.

- The handlers are atomic, i.e., once activated, they execute till completion. Hence their basic role is mainly to integrate the received data into the computation or to reply to a request. Atomic handlers and deadlock avoidance have conflicting requirements and their resolution is discussed in a later section.

3.2 Example: am_send implementation

To give a better idea of how the AM layer functions, we describe the execution of one of the AM primitives: am_send. The role of am_send is to send an active message to another node. From the point of view of the AM user, *am_send* is a C library function which takes as arguments: the destination node, a pointer to the handler to be invoked on the destination node, and two arguments to this handler.[3] Figure 2 shows the execution on the sending and receiving nodes.

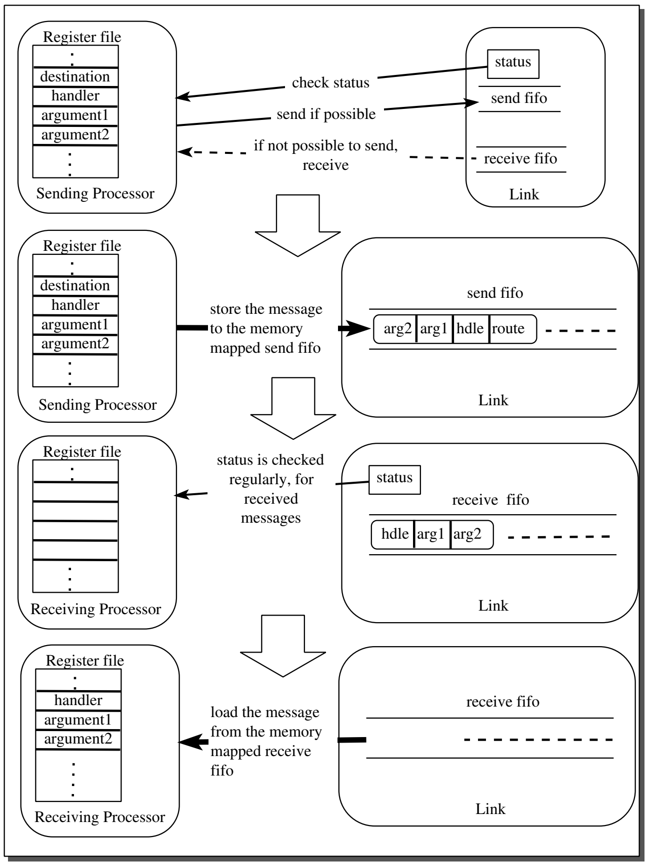

Figure 2: Execution of am_send

- After the call to am_send, the destination node id, the handler pointer, and the arguments are in the registers of the sending processor. The status register of the Link Chip

[3] The restriction of the number of arguments to a handler allows quicker processing of received messages.

is checked to see if it is possible to send. If it is not possible, the processor receives messages from the receive FIFO, if any.

- When it is possible to send, the above mentioned registers are stored in the sending buffer of the Link chip.
- At the receiving end, the processor frequently polls the receiving status, either through explicit polling primitives or from within sending primitives.
- When the processor finds a message in the receive FIFO, it loads the message from the receive buffer to the appropriate registers. It then calls the function pointed to by the handler pointer, which uses the arguments that are already in the register file. Upon completion of the handler, the processor resumes the task it was performing before.

3.3 Request-Reply Mechanism

To satisfy the second characteristic of the AM layer (atomicity of handlers), one problem arises: if a handler wants to send a message and it is blocked due to a temporary network congestion, it can lead to deadlock. That is, there can be a cycle of nodes which are not able to reply to each other (every one wants to send and none is receiving). Therefore, in every sending primitive, if the processor cannot perform the send, it must check the network for messages received in the receive buffer. If there is a message, it must process it by invoking the appropriate handler. However, allowing messages to be received during handler execution can lead to an arbitrary growth of the runtime stack.

Our solution to limiting the runtime stack growth is to provide a set of request and reply primitives. The idea is to prevent a node from issuing more than a certain number of unsatisfied requests to another node. By making the requester and the replier communicate through those explicit primitives instead of *am_send*, the active messages layer is able to monitor the number of unsatisfied requests, and hence can block the requester from sending too many requests.

3.4 TCL Primitives

The Threaded Communication Library (TCL) consists of C functions—TCL primitives—which utilizes the AM primitives to perform communications, synchronizations and multithreading support. It permits a user to write programs under a multithreaded program execution model, and it also has primitives to support the Single-Program Multiple-Data (SPMD) paradigm. That is, TCL primitives allow a processor node to switch to another part of the computation while the current computation must wait for data and/or synchronizations from another processor node.

Communication primitives come in two flavors: fine and coarse grain. Fine grain primitives are basically remote accesses (reads and writes) for basic data types in the C language (e.g., int, float, double). Coarse grain primitives are for block transfers. The program accesses a piece of data on another node by specifying the node and the address to be accessed on that node. Unlike traditional message-passing primitives, the receiver node of a send message does not have to post any receives; since in the AM layer, any node can access the address space of another without the latter executing user code to facilitate the transaction. This should be more efficient in the case where it is not possible to define statically the communication patterns.

In addition to these basic communication primitives, TCL has some primitives that permit the user to specify operations to be done on the remote data (as will be seen in the example subsection). In this manner, the programmer can tailor some of the TCL primitives by augmenting the base handlers for efficiency.

TCL provides synchronization primitives for the following cases: 1) when a piece of data produced by a node (or thread) is consumed by another—producer/consumer operations—2) in split-phase remote accesses (the standard in our case) to ensure that the fetch requests has been satisfied and the data has arrived, and 3)barrier synchronizations (SPMD support) for coarse-grain synchronizations between threads.

Augmenting the communication layer with multithreading capabilities allows a programmer to integrate asynchronous events such as communication/synchronization into computation, with little software overhead. Furthermore split-phase transactions permit to hide synchronization and communication latency. This is especially true for programs with irregular communication patterns since multithreading permits "dynamic" scheduling of computations with satisfied dependencies. The multithreading support provided is very simple: the programmer creates a thread by associating a function and its arguments to it, context switching between threads is done when a synchronization is not satisfied, and the thread associated with a terminating function is deleted automatically. In other words, communication/synchronization and computation are "tightly coupled" under TCL.

The reader may refer to appendix A for a listing of the TCL primitives.

3.4.1 Structure of TCL

TCL on MANNA is implemented in such a way as to dedicate one of the i860s in a MANNA node for application execution, while the other is dedicated for communication and thread scheduling. We call the first processor the applications processor (AP) and the second, the communications processor (CP); the same nomenclature used in the description of the MANNA hardware. The CP helps the overlapping of communications with computations in the following manner:

- For block transfers, the CP functions as a DMA to the network.

- It relieves the AP from handling messages in the receive buffer. That is, the AP does not need to participate in a communication transaction and can be totally dedicate to the application code. Moreover, cache interference in the AP between application code and code for handling messages are eliminated. However, the received message does not automatically reside in the cache of the AP.

- During the sending of a message, the CP hides any network congestion from the AP, but at the expense of increased cache snooping activity between the two i860XPs and cache misses suffered by the CP.

- Lastly, the CP schedules threads when dependencies are satisfied (synchronizations) without interrupting the AP.

Figure 3 shows the structure of the TCL implementation. The Application Processor

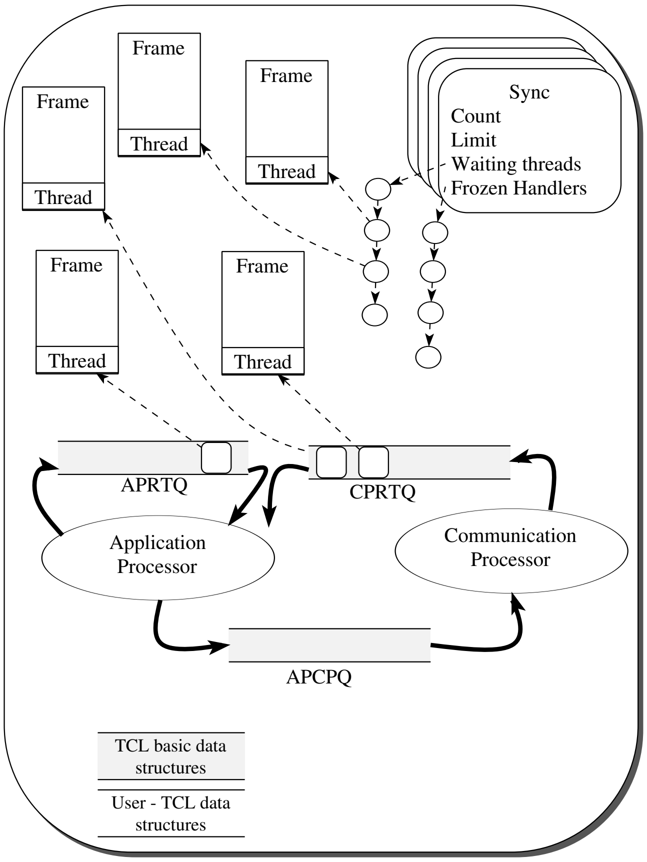

Figure 3: Threaded Communication Layer Structure

Communication Processor Queue (APCPQ) is used for sending messages from the AP via the CP, and when the AP is requesting the CP to enable some threads. The CP Ready Threads Queue (CPRTQ) contains ids of threads enabled by the CP, basically threads that were waiting for synchronization signals which were later satisfied. The AP Ready Threads Queue (APRTP) contains ids of threads enabled by the AP, i.e., when the thread is created the first time. We differentiate between the two kinds of enabled threads (by AP, or CP), to avoid having them in one data structure which has to be locked by the AP or CP for every access. Threads enabled by the CP are given higher priority since they are basically threads that contain on going computations.

For each thread, there is a pointer to its corresponding activation frame (see fig. 3) which also serves as a unique identifier for the thread. The activation frame of a thread is similar to a stack activation frame in conventional processing, i.e., it contains all the necessary contextual data such as storage for register contents, function linkage information, temporary data, etc. An activation frame is created when the function associated with a thread is started, and is deleted when the function terminates. Frames are allocated from a heap rather than from a runtime stack.

For data structures in which producer/consumer operations are performed, synchronization variable(s) are associated with the data structure to regulate access to single elements, parts of the data structure, or the whole data structure itself. All synchronization (sync) variables are stored in a Synchronization Structure (see fig. 3) which is distributed according to the ownership of the sync variables; the ownership of sync variables are determined by the ownership of the data they guard. The producer(s) of data for the data structure increments the appropriate sync variable when data is produced. The consumer(s) of data in the the data structure check if the sync variable equals a predetermined value called the *limit value* (each sync variable has one). If it is, the consumer can proceed, if not, it puts itself in a waiting list of the sync variable and performs a thread switch. The producer which makes the sync variable equal to the limit value is responsible for requesting the CP to enable all the threads waiting on this sync variable. Thus, the synchronization structure has for each sync variable an associated limit value, a pointer to a list of waiting threads, and a pointer to a list of frozen handlers. The frozen handlers are those acting on behalf of remote nodes which are trying to access the guarded data. When the synchronization requirement is met, the handler(s) are restarted (similar to the restarting of threads) and subsequently return the data to the requesting node(s).

3.5 Example of Using TCL

We now illustrate the use of TCL primitives in an example program which sorts integers. to sort integers. The goal of the program is to sort an array of randomly generated keys (integers) in which the range of their values are known. The algorithm used is based on the bucket sort algorithm in the NAS Integer Sort benchmark program. There are three main data structures: the unsorted_keys array containing the keys (integers) to be sorted, the sorted_keys array to be filled with the sorted keys, and the key_density array which contains for every value in the specified range, the number of keys that have that value. These arrays are block-distributed among nodes, i.e., for an array $A[N]$ to be distributed, we declare on every node an array $A[N/n]$ where n is the number of nodes. The algorithm has two main steps: in the first pass through the unsorted_keys array, the key_density array is updated to get the number of keys that exist for every value. And in the last pass through the unsorted_keys array, every key gets a rank from its corresponding value in the key_density array, and gets sorted in the sorted_keys array according to this rank.

The part of the program which we show below is the one implementing the second step. That is we are starting with the key_density array updated, and on every node we have a number of threads, each of which is responsible for sorting a part of the unsorted_keys array based on the information in the key_density array. The programmer would choose the most suitable number of threads to achieve maximal hiding of the communication latency incurred. Note that all the TCL specific primitives and data types start with *tcl_*.

```
void sorter (num)
int num; /* number of the threads */
{
  int start_keys, end_keys; /* limits of the range of action of the thread */
  int value, rank;
  tcl_sync_type rank_sync;
  int i;

  start_keys = num * chunk_size;
  end_keys = start_keys + chunk_size;
  for (i = start_keys; i < end_keys; i++) {
    value = unsorted_keys[i];
    /* do I have this part of the keydensity array or not */
    if (value2node(value) == me) { /* Yes it is local */
      rank = --key_density[value];
      tcl_signal (&rank_sync);
    }
    else
      /* No it is not local: so I have to get it from the remote node.*/

      tcl_modify_fetch_sync (value2node(value), &keydensity[value%nodes],
                             &rank, &rank_sync, decrement);

    /* Wait till the rank is received from the remote node. */
    tcl_wait (&rank_sync, 1);

    /* Depending on the rank, sort the key in its place either locally or
       remotely. */
    if (rank2node(rank) == me)
      sorted_keys[rank] = unsorted_keys[i];
    else
      tcl_rem_write (rank2node(rank), &sorted_keys[rank%nodes],
                     unsorted_keys[i]);
```

```
    }
  }

  void sort_keys()
  {
    tcl_local_barrier_type sort_bar;
    int i;

    tcl_init_barrier (&sort_bar, num_sorter_thd + 1);
    for (i = 0; i < num_sorter_thd; i++)
      tcl_create_thread (sorter, i);
    tcl_barrier (&sort_bar); /* wait till all the forked threads are
                                finished before ending the function */
  }
```

In the 'sort_keys' function, tcl_create_thread creates a new thread, and associates a function and an argument to it. Tcl_init_barrier initializes a barrier synchronization variable and tcl_barrier performs a join operation between local threads. That is, once all local threads have synchronized on the barrier variable 'sort_bar', the computation can progress to the next step, i.e., return to the caller of sort_keys.

In function 'sorter', 'value2node' and 'rank2node' functions produce the processor node id which owns the value and the rank of the key, respectively. Their results are used within TCL functions to specify destination nodes for the TCL operations which must perform an interprocessor communication. 'Rank_sync' is declared as a synchronization variable which is initially set to zero along with its corresponding limit value. Tcl_signal and Tcl_modify_fetch_sync increments rank_sync by one, while tcl_wait is responsible for setting the limit value of rank_sync to one (the second parameter specifies the limit value for the synchronization variable). If rank_sync is not equal to one during the tcl_wait operation, the thread is suspended and put in a list waiting for the synchronization signal, and the processor context-switches to another runnable thread.

Tcl_modify_fetch_sync permits the user to specify an operation to be performed on the keydensity on the remote node before fetching the data and storing it locally in variable rank. Upon returning with the data, rank_sync will be incremented; if rank_sync equals the limit value, then threads waiting upon rank_sync will be restarted. In the example, the user specifies the 'decrement' function which decrements the keydensity element in the remote node. The decrement function is a user-defined function, i.e., the TCL primitive can perform any operation on the fetched value before it is returned.

4 Experimental Results

In this section, we demonstrate the usefulness of the AP-CP designations of the dual processors in our TCL implementation. For this, we implemented two different versions of TCL: one with the AP-CP duals, and the other which utilizes only one processor in a node to perform both AP and CP tasks. In the following sections, we present those benchmarks, the experimental results, and analyses.

4.1 Benchmark Programs

We have implemented three benchmark programs which use TCL primitives for our experiments. These programs are classified into two categories:

- Numerical benchmarks with regular and coarse communication patterns: Integer Sort and Matrix Multiplication.
- Non-numerical benchmarks with irregular fine grain communication patterns: Paraffins.

Integer Sort Integer Sort is from the NAS parallel benchmarks suite [BBLS91]. The benchmark program is to sort N keys (integers) in parallel. The keys are initially generated by a pseudo-random generator, and distributed on the different nodes of the machine. The keys are within a certain range which is much smaller than the number of keys. These keys are ranked—a rank indicates its position in the sorted sequence, and then the keys are sorted according to the ranks, and checked for being correctly ordered.

The sequential implementation suggested by NAS is based on the bucket sort algorithm.

Our parallel implementation,[4] which is based on this sequential version, distributes the keys and ranks arrays on the different nodes, while keeping on every node a local copy of a key-density array. The key-density array specifies for each rank the number of keys. Every node updates its key-density array locally, and then there is a global communication/computation phase which updates the local key-density arrays so that they reflect the global state. The updating of the array is done in a pipelined manner where a processor A sends to its 'neighbour' a block of its key-density array so that its neighbour processor B can update its portion of its 'running-sum' array. Processor A continues to send the remaining blocks until all are sent. Meanwhile, as processor B finishes updating its portion of the key-density array due to the first block sent from A, it sends that portion of its key-density array to its neighbour processor C. C executes similarly as B. This continues until the last processor in the 'pipe' contains the overall

[4]This version for experimentation is not the same as the one illustrated in the example describing the use of TCL primitives.

key-density array. The overall key-density array is then propagated to the other processors such that their key-density array can be updated. It is in this phase where we expect to have overlapping of computation and communication through the use of CP. After this phase, the keys are ranked locally using the local key-density array.

Matrix Multiplication In our implementation, the resultant matrix C and the two matrices to be multiplied, A and B are block distributed column-wise. Every node is responsible for calculating its corresponding group of C columns. The C and B columns stay local in their corresponding nodes; and the A columns circulate among the nodes. In every step, a node uses the newly received A columns to accumulate the multiplication result in C.

Overlapping of communications with computations should result from the coprocessor sending local columns of A while the main processor is using them for computations. To allow for the overlapping, there are two buffers that are allocated for A in every node; so when one is read by the main processor, the other is receiving data, and they are switched on every step of the algorithm.

Paraffins The Paraffins benchmark [NA90] is an example of non-numerical applications. Its goal is to generate all the paraffins of sizes up to n. Paraffins are molecules with chemical formula C_nH_{2n+2}. A paraffin is composed of smaller paraffins called 'radicals'; a radical is a paraffin with a single hydrogen atom removed (C_nH_{2n+1}).

The main data structures of paraffins are arrays where every element of an array points to a list of paraffins of the same size. Every element of this list is unique in that it is composed of a different set of radicals. That is, the program operates on pointer-based data structures. In Paraffins, functions are used to generate paraffin structures of given sizes. In our implementation, we assign to different nodes, paraffin generating functions of different sizes. Since the number of paraffins of different sizes vary dramatically (exponentially as n increases), a dynamic load balancing mechanism is implemented in such a way that if a node is idle, it sends to the neighbouring nodes asking for work where a unit of work is another paraffins generating function. Due to the load balancing, the major type of communication for this application is fine-grain and asynchronous.

4.2 Experimental Results and Analyses

Figure 4 shows the performance of the dual-processor version versus the single-processor version of TCL for the three benchmarks running on two to sixteen nodes.[5] Matrix Multiply multiplied two 512×512 matrices, Paraffins enumerated paraffins up to size 20, and Integer Sort sorted 65536 integers. Matrix Multiply gives good speedups while Paraffins produces decent speedups and Integer Sort does not. The speedups are with respect to the optimal sequential version without TCL primitives running on one processor of a MANNA node.

[5]The largest available MANNA is sixteen nodes.

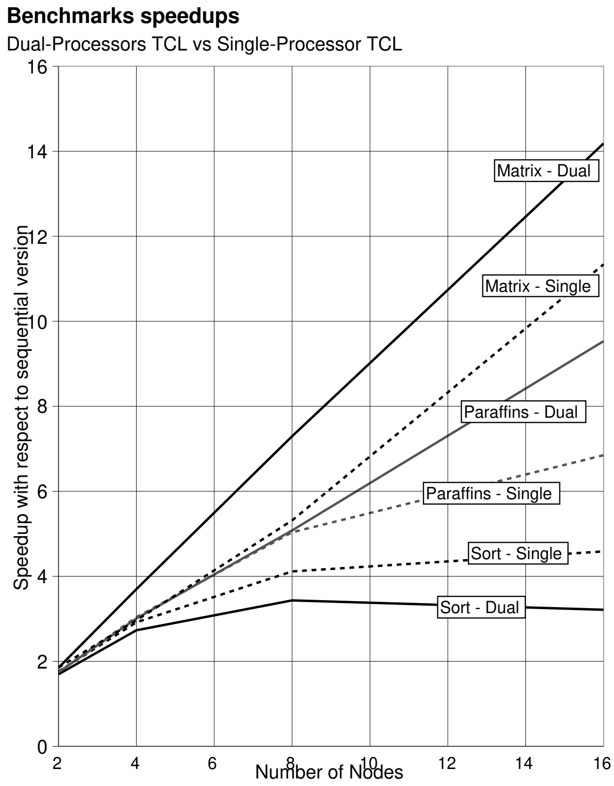

Figure 4: Speedups of the benchmarks for single-processor and dual-processor versions of TCL.

4.3 Analysis of Matrix Multiply Results

Matrix Multiply results show that as the number of processors increase, the benefits of the CP increases. The reason is that as more processor are involved, the amount of communications increases, and the benefits of the CP becomes more pronounced.

Effect of Problem Size in Matrix Multiply To see how much benefits we can derive from the CP, we varied the size of the matrices in Matrix Multiply, and thoses results are shown in figure 5 from running on four nodes. The synchronization time is attributed to times when a processor must wait for a synchronization signal before it can proceed, and communication time is the time to send the message from the point of view of the AP. For the single processor version, all tasks are performed on the AP. For small matrices, the matrices can reside totally in the data cache of the AP and thus the single processor version shows better performance since the columns of A sent in the messages will be brought directly into the cache of the AP when the AP receives the message. For the dual-processor version, the CP brings the message into local memory, and when the AP wants to access the columns of A in the message, it would suffer cache misses and thus increase the execution time. From the figure, we can see that

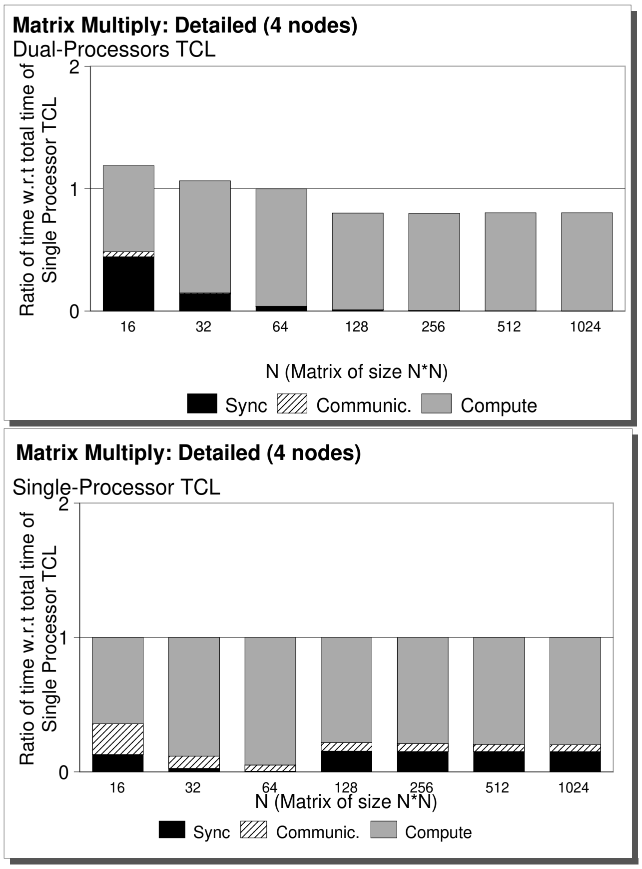

Figure 5: Detailed Measurements of Matrix Multiply

cache miss penalties outweigh the benefits of overlapped communications-computations until matrices of size 64×64. At 32×32, the matrices no longer fit in cache, but still some data brought in by the AP in the single processor version remains in cache when the AP wants to process them. At 64×64, all data brought in by the AP will be stored in main memory before the data is used. Thus, the AP in both the single-processor version and dual-processor version will suffer the cache misses. For larger size arrays—larger amounts of data to be sent and received—the CP benefits the execution since in the single-processor version case, the AP cannot compute while it is waiting for the columns to be received (the synchronization time). In the dual-processor case, the wait time has been eliminated since the CP brings the data in while the AP is computing.

4.3.1 Analysis of Paraffins Results

Paraffins relies heavily on good dynamic load balancing, i.e., spreading out the paraffins generating functions appropriately. If one function is responsible for generating too many, then the processor executing that function will be busy long after other processors become idle; thus limiting the attainable speedup. The trick in writing a good parallel Paraffins is to specify the appropriate granularity of the paraffins generating function such that the units of work can be small enough so that the load balancer can work effectively. However, it cannot be too small where the overhead to dispatch the work is greater than the work done. Currently, the granularity in our version of Paraffins is the smallest achievable without wholesale changes to the algorithm.

On the benefits of the CP in the dual-processor version of TCL, we can see that running on sixteen processors yields much better performance than the single-processor version. The reason is that as the number of processor nodes increases, the load balancing activity increases. This can be seen from figure 6 which shows the increase of the number of messages with respect to runtime. Moreover, it shows that the dual-processor TCL version processed more messages, thus leading to better load balancing. Load balancing tasks do not interfere with operations

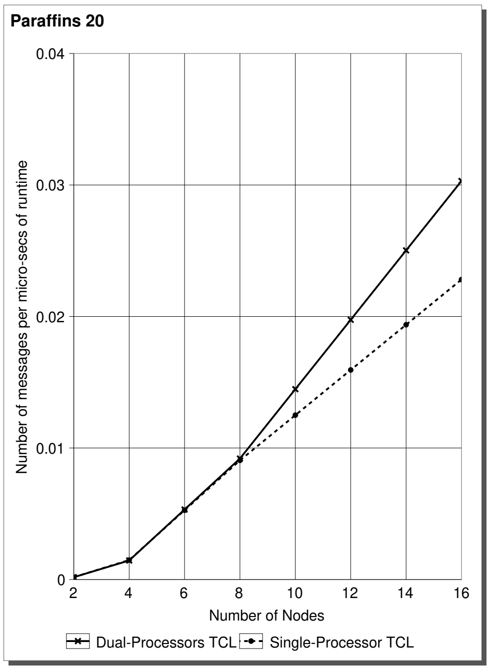

Figure 6: Number of messages communicated in Paraffins.

performed in the AP; therefore, by having the CP always available to receive such asynchronous messages, the AP can concentrate on the main task. For applications requiring dynamic load balancing, the CP is very useful.

4.3.2 Analysis of Integer Sort Results

Integer Sort is dominated by the global updating of the key-density array, which in our case, is done in a pipelined manner. In the run, the size of the block of data sent during the global updating phase is 1Kbytes, where each processor sends sixteen blocks.[6] During the global updating phase, a processor node receives a block of data, adds the elements in the block to its portion of the key-density array and sends the updated data as a block to the next processor. Data received by the CP in the dual-processor version are simply brought in, and when the AP wants to operate on it, caches misses are encountered which lengthens the execution time. Whereas, in the single-processor version, the block of data is brought into the AP's cache when it receives the message, thus suffering none of the cache misses encountered in the dual-processor version.

The Integer Sort results and those from Matrix Multiply for small matrices point to the importance of caching incoming data such that the AP does not encounter avoidable cache misses. This can be accomplished by having a common cache in which globally shared data are stored when the CP receives them from the network [HG93].

5 Related Work

In the area of architectural support for communication/synchronization, many commercial products and academic projects have intelligent devices—either processors or specialized hardware—to handle the non-computational activities in a node of a parallel machine. In Paragon [EK93], the interface to the interconnection network consists of a message processor, a network interface controller, and two DMA controllers. The message processor is a second i860XP processor operating in parallel and sharing the memory with the application processor. Its task is to perform the details of inter-node communication and to provide global communication functions. In Typhoon [RLW94], a network interface processor (NP) connects the network to the shared bus of the node. The NP contains a full SPARC integer processor, taken from a previous generation design, with instruction and data caches to enhance performance and a TLB to allow virtual addressing. The NP adopts ideas similar to active messages, and runs user-level code, as a way of increasing performance and flexibility. StarT-NG [AAC94] has multiple sites hooked up in a network. A site has a PowerPC processor, helped by a Network Interface Unit (NIU) to provide the functionality of remote memory accesses and synchronization; while one or more other PowerPC's in the site are responsible for data processing. A node in Flash [KOH+94] contains a custom controller called MAGIC (Memory And General Interconnect Controller). It is a programmable, highly integrated chip that implements all data transfers both within the node and between the node and the network. It provides flexible control to support a variety of distributed shared-memory and message-passing protocols. In MTA [HTG94], a custom chip called the Synchronization Unit (SU) is responsible for communication

[6] This was found to be optimal through experimentation. We found that as the size of blocks decreased, and thus the granularity of computations decreased, the execution time increased. The same occurs as we increase the block size. This points to a tradeoff between the amount of parallelism exposed for parallel execution and the granularity of the parallel tasks—the larger the task size, the more amortization of the parallel overhead. This tradeoff is a subject of another paper.

and synchronization in a node; while an off-the-shelf RISC processor Execution Unit (EU) is responsible for computation. In our work, we used an existing parallel machine which has two processors per node and wrote a multithreading library which dedicates one of the processors for communications, synchronizations, and thread scheduling.

In the area of expressing parallelism and multithreading constructs at the user level, work has been done on defining user-level multithreading packages and extending sequential languages such as C with multithreading functionality. An example of the former is POSIX Threads [IEE92]. It provides the programmer with primitives to create light weight threads— compared to processes in the operating system— and assign priorities and scheduling policies to them. Primitives for mutual exclusion, synchronized waiting, and signal handling are provided as well.

Examples of the latter include: Cilk [HZJ94], Cid [Nik94], and MTA Threaded-C [HMT+94]. Cilk is a C language extension with a supporting runtime system which is based on threads in the explicit continuation-passing style. The runtime system embodies a work stealing scheduler for load balancing purposes. Cilk is designed for message passing parallel architectures (a current version is implemented on Thinking Machine's CM5). Cid is a parallel, "shared memory" superset of C suitable for distributed- memory machines. It has some similarities to TCL such as having forking thread primitives, join cells, and split-phase remote accesses. MTA Threaded-C is an extension for C and is based on a multithreaded execution model. It is augmented by instructions which are supported by the MTA architecture. The language is primarily used as an intermediate language for a multithreading compiler. TCL is different from these efforts in that no compiler components are modified.

TCL, like the POSIX threads package, provides a user with a set of library functions. The major difference is that TCL provides a program execution model similar to a dataflow program execution model where inter-thread communication/synchronization is tightly integrated with computation. Thus, threading primitives as well as communication and synchronization primitives are interwoven in the computation model. Furthermore, TCL is targeted for a dual-processor node in which one processor is solely dedicated to communications and synchronizations.

6 Conclusions and Future Work

We have introduced a basic user-level communication library capable of supporting multithreading operations and SPMD computations. The Threaded Communication Library is implemented on the MANNA multiprocessor in which one of the two processors in a node is dedicated to computations while the other is dedicated to handling communications, synchronizations and thread scheduling.

Our experiments showed that dedicating an extra processor in a node for communication/synchronization proves useful for applications with frequent, irregular and fine-grain communication requirements. However, for more regular applications, caching of incoming data becomes an important issue since cache misses encountered by the computation processor when operating on data in incoming messages can have a major impact.

In the future, we will work in a few directions to address the above problem. One direction is to let the coprocessor do more computation. One way is to implement more high level primitives in TCL, such as “collective communications” operations (gathering, scattering, reduction, etc...). This has the advantage that the communication processor would be able to perform some computation on the data which it already has in its registers or cache, without requiring attention from the application processor. A second direction is exposing a CP and AP to the user such that one can specify which processor in the receiver processes the incoming message.

7 Acknowledgment

We would like to thank the Natural Sciences and Engineering Research Council (NSERC) for their support of this research. The first author would like to acknowledge the support of CRIM and FCAR. The second author would also like to acknowledge the support of a Concordia FRDP.

References

[AAC94] Boon Seong Ang, Arvind, and Derek Chiou. StarT the next generation: Integrating global caches and dataflow architecture. CSG Memo 354, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, Massachusetts, February 1994.

[BBLS91] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS parallel benchmarks. Technical Report RNR-91-002, NAS Systems Division, January 1991.

[CSS+91] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 19(2), April 1991; *Operating Systems Review*, 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.

[DG94] Jack B. Dennis and Guang R. Gao. Multithreaded architectures: Principles, projects, and issues. In Robert A. Iannucci, Guang R. Gao, Robert H. Halstead, Jr., and Burton Smith, editors, *Multithreaded Computer Architecture: A Summary of the State of the Art*, chapter 1, pages 1–72. Kluwer Academic Publishers, Norwell, Massachusetts, 1994. Book contains papers presented at the Workshop on Multithreaded Computers, held in conjunction with Supercomputing '91 in Albuquerque, New Mexico, November 1991.

[EK93] Rudiger Esser and Renate Knecht. Intel paragon XP/S - architecture and software environment. Technical Report KFA-ZAM-IB-9305, Forschungszentrum Julich GmbH, April 1993.

[GMD93] Gesellschaft für Mathematik und Datenverarbeitung mbH. *MANNA Hardware Reference Manual.* Berlin, Germany, 1993.

[HG93] Herbert H. J. Hum and Guang R. Gao. Supporting a dynamic SPMD model in a multi-threaded architecture. In *Digest of Papers, 38th IEEE Computer Society International Conference, COMPCON Spring '93*, pages 165–174, San Francisco, California, February 22–26, 1993. IEEE Computer Society Press.

[HMT+94] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, Laurie J. Hendren, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. The Multi-Threaded Architecture multiprocessor. ACAPS Technical Memo 88, School of Computer Science, McGill University, Montréal, Québec, December 1994.

[HTG94] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 288–294, Cancún, Mexico, April 26–29, 1994. IEEE Computer Society.

[HZJ94] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. Parallel programming based on continuation-passing threads. CSG Memo 355, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, Massachusetts, 1994.

[IEE92] IEEE. Threads extension for portable operating systems (draft 6), February 1992.

[KOH+94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Chicago, Illinois, April 18–21, 1994. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 22(2), April 1994.

[NA90] Rishiyur S. Nikhil and Arvind. Id: a language with implicit parallelism. CSG Memo 305, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, Massachusetts, February 1990.

[Nik94] Rishiyur S. Nikhil. Cid: A parallel, "shared-memory" C for distributed-memory machines. Technical report, Cambridge Research Laboratory, Digital Equipment Corporation, July 1994.

[NPA92] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 20(2), May 1992.

[RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, Chicago, Illinois, April 18–21, 1994. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 22(2), April 1994.

[vECGS92] Thorsten von Eicken, David E. Culler, Sech Copen Goldstein, and Klaus Eric Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 20(2), May 1992.

A Threaded Communication Library Primitives

This appendix lists the basic primitives of TCL. Two notes are to be made: there are memory access primitives for integers, floats and doubles, but we only list the general form. When we say that a primitive *operates on a synchronization variable*, we mean that it increments its count, compares it to its limit value, and if they are equal, it enables the threads waiting on it and restarts the frozen handlers waiting for it.

A.1 Thread Creation

tcl_create_thread (*function*, *argument*)
Creates a thread, and associates to it *function*, which takes *argument* as argument.

A.2 Synchronization Primitives

To declare TCL synchronization variables, **tcl_sync_type**, **tcl_loc_barrier_type**, and **tcl_glob_barrier_type** can be used in variable declarations. **Tcl_sync_type** is used to declare synchronization variables for producer/consumer operations and split-phase transactions.

Tcl_loc_barrier_type is used to declare a local barrier synchronization variable while **tcl_glob_barrier_type** is used to declare a global barrier synchronization variable where each node has a copy of the barrier variable. Updating of the global barrier variable is performed in a tree-structured manner for efficiency.

The synchronization primitives are listed below:

tcl_wait (*sync*, *limit*)
The limit value of the synchronization variable *sync* is set to *limit*. Then *sync* is checked if it equals its limit value. If so, the thread continues execution. Otherwise, it appends the thread issuing the primitive to the list of threads waiting for *sync*, and switch to another ready thread.

tcl_signal (*sync*)
Increments a local synchronization variable *sync*. If it equal its limit value, then it enables all the threads waiting on *sync* as well as the frozen handlers.

tcl_rem_signal (*node*, *sync*)
This primitive is similar to **tcl_signal** except that this one increments a remote synchronization variable *sync* remotely.

tcl_init_barrier (*barrier*, *value*)
Initializes the synchronization variable *barrier* to *value*.

tcl_barrier (*barrier*)
Performs a barrier synchronization on the local synchronization variable *barrier*. That is, if *barrier* equals its limit value then the computation can advance pass the barrier point, else the thread is put in the waiting list of the synchronization variable.

tcl_glob_barrier(*barrier*)
This is the same as **tcl_barrier** except that the barrier synchronization variable is a global one.

A.3 Communication Primitives

tcl_write_sync (*address*, *value*, *sync*)
Writes *value* in *address* on the local node, and operates on the local synchronization variable *sync*. That is, increment it, check if it equals its limit value, if so, then restart the threads waiting on *sync*.

tcl_rem_write (*node*, *address*, *value*)
Writes *value* into *address* in the remote processor *node*.

tcl_rem_write_sync (*node*, *address*, *value*, *sync*)
Writes *value* into *address* in the remote processor *node*, and operates on synchronization variable *sync*.

tcl_fetch (*node*, *source*, *destination*)
Fetches data from address *source* in processor node *node*, and stores it locally in address *destination*.

tcl_fetch_sync (*node*, *source*, *destination*, *sync*)
This is the same as **tcl_fetch** except that it also operates on the local synchronization variable *sync*.

tcl_fetch_rem_sync (*node*, *source*, *destination*, *sync*, *limit*)
Remotely compares *limit* to the remote synchronization variable *sync* in *node*. If they are equal, it fetches data from address *source* at the remote node, and writes it into local address *destination*. Otherwise, the fetching handler will be frozen on the synchronization variable *sync*.

tcl_fetch_rem_sync2 (*node*, *source*, *destination*, *loc_sync*, *rem_sync*, *limit*)
This is a combination of the two previous primitives. It does everything which **tcl_fetch_rem_sync** does where *rem_sync* is the synchronization variable on the remote node and it also operates on the local synchronization variable *loc_sync*. This primitive resembles I-Structure accesses.

tcl_modify_fetch_sync (*node*, *source*, *destination*, *sync*, *function*)
Reads the data in address *source* on the remote processor *node*, and passes it as a parameter to *function* that is also invoked on *node*. The return value of *function* is stored in *source* and sent back and stored locally in *destination*. Lastly, the *sync* variable is operated upon.

tcl_modify_fetch_rem_sync (*node*, *source*, *destination*, *sync*, *limit*, *function*)
This is the 'modify' version of **tcl_fetch_rem_sync** just as **tcl_modify_fetch_sync** is to **tcl_fetch_sync**.

tcl_modify_fetch_rem_sync2 (*node*, *source*, *destination*, *loc_sync*, *rem_sync*, *limit*, *function*)
This is the 'modify' version of **tcl_fetch_rem_sync2**.

tcl_blk_send (*node*, *source*, *destination*, *loc_sync*, *rem_sync*)
Sends a block of data from *source* to *destination* on remote processor *node*. Local synchronization variable *loc_sync* is operated on after the block is sent, and remote synchronization variable *rem_sync* is operated upon remotely after the block is received.