MPI for the Transtech Parastation

L. Lämmer[†], J. Sziveri[§], B.H.V. Topping[‡]

[†]Research Associate

Institute for Numerical Methods & Informatics in Civil Engineering

Technische Hochschule Darmstadt

Darmstadt, Germany

[§]Research Associate

[‡]Professor of Computational Mechanics

S.E.C.T. Research Group

Department of Mechanical & Chemical Engineering

Heriot-Watt University,

Edinburgh, UK

March 19, 1997

Abstract

A partial implementation of the MPI specification has been developed for a special distributed memory MIMD parallel computer, the Transtech Parastation, in the form of a low-level C library. The underlying routing scheme is briefly discussed. System requirements are only a 3L Parallel C compiler. The comparison with respect to a more secure and more portable implementation of application programs is shown. Through an example program which is a parallel non-linear dynamic finite element analysis code, transition notes are given to develop programs from existing 3L Parallel C and native transputer applications into equivalent programs complying to the MPI specification.

1 Introduction

Existing parallel computers can be simply classified into shared memory systems and distributed memory systems. True shared memory machines have a global memory attached to a number of processors enabling several processors to work concurrently on different parts of the same computation. The cost of building shared memory parallel computers increases rapidly with the number of processors. A different approach towards building large parallel computers is to connect several processors via a network, with each processor having its own local memory. Examples are transputer based systems, which are built using the four links of the processor for connectivity, or networks of workstations which use the bus-based LAN-interconnection (mainly ethernet technology). The distributed memory multiprocessor systems are scalable over a wider range than the shared memory computers. There are many intermediate computer architectures, each with its own distinct programming model. In general, computer models can simulate one another, so it is possible for one computer model to run application software developed for other programming models.

Portable parallel programming is essential for the successful development of application software. At best, these proprietary interfaces only provide for the generation of efficient, high performance applications. The ease of programming and the portability of code across platforms were seldom the design goals of these programming interfaces.

The Message Passing Interface (MPI) Standard[1] is a standardised interface to the distributed memory programming model. It was established to provide a highly efficient message passing system on a large number of platforms, each with the same programming interface. A number of hardware vendors have become committed to developing native implementations of MPI on their computer systems. Additionally, public domain implementations of the interface were developed using the exisiting low level communication primitives of the operating systems (mainly UNIX) or the proprietary message passing interface (e.g. NX by Intel) [2]. Programming in parallel with MPI has since become very popular and there is even a need for an MPI interface for a transputer based system.

This paper is organised as follows. After a short description of the transputer based Parastation hardware used in this study, Section 2 gives an overview of the history of MPI and briefly discusses its basic principles. The discussion is restricted to the features implemented in a low level MPI interface for the Transtech transputer system. This implementation is described in Section 4. Next, results and comparisons with the original message passing procedure using the transputer router [3, 4] are given.

1.1 The Parastation hardware

The Transtech Parastation is transputer based product. The transputer is a general purpose computational element packaged as a single chip. It contains a special dedicated communications circuit, four fast DMA (Direct Memory Access) serial-communication links. These links enable the construction of a complete computing engine from several transputers and thus, in this sense, the tranputer is a building block of larger parallel computers.

In keeping with the building block philosophy of transputers the TRAM (TRAnsputer Module) standard evolved so that a transputer based computing equipment could be purchased and built in a way that allowed users to customise their computers for their particular application. TRAMs are self contained computing sub-systems. Physically they are small assemblies based on transputers with a standard electrical and mechanical interface. They plug onto standard, purpose designed TRAM motherboards which in turn plug into or can be connected to a range of host computers. Typically a TRAM would contain a transputer with some memory. (The flexibility of the standard allows parallel machines to be constructed with a mix of computational performance and memory or other functionality to be inserted into the parallel machine at the optimum place, such as disk, graphics or other I/O capability.)

The TRAM brings out all four of the transputer's links for connection to other TRAMS. These links are connected on the motherboard which is designed in such a way that:

- transputers can be configured into any topology;
- it is possible to run and test applications on transputers without first having to configure the links;
- a number of motherboards may be chained together.

The motherboard architecture that was evolved to meet these requirements consists of the following important features:

- all the modules in the network are connected in a pipeline using two of the four transputer links from each module. This is called the default or boot pipeline;
- the remaining links from each module are available for user configuration either by direct wiring or via a programmable link switch;

- the motherboard also provides an interface to allow the non-transputer based host computer to control and communicate with the TRAMs. (This is realised by allowing a small server program running on the host computer to access one of the free links of the first module in the pipeline on the first motherboard.)

The Transtech Parastation is a desktop enclosure based around a TMB24, a 32 slot TRAM motherboard. The Parastation used in this study is connected to the host computer, which is a SUN Sparcstation, by a Transtech Matchbox via the SCSI bus of the SUN workstation.

Transtech Parallel Systems Ltd designs and manufactures embedded multi-processing products for OEM, end-user and scientific research applications. Transtech produces signal processing and computing systems based on Analog Devices SHARC (Super Harvard ARChitecture) DSPs, PowerPC processors, Texas Instruments TMS320C4x DSPs, SGS Thomson transputers and the Intel i860. Thus in addition to the range of transputer based modules, Transtech also produces TRAM products with either a PowerPC or an Intel i860 co-processor which can be easily used as "super" transputers within systems.

3L Ltd is a UK based company developing parallel programming environments for SGS Thomson transputer and Texas Instruments C40 based hardware. Their product Parallel C version 2.2.4 was used in this study to program the Transtech hardware.

2 Message Passing Interface (MPI)

2.1 Forming of the MPI Standard

Early work on the MPI Standard started in 1992 at Oak Ridge National Laboratory and at Rice University to ensure that efficient library and application software could be developed and ported to a wide spectrum of high performance multicomputers. At that time, several message passing libraries had been developed and widely deployed. It was felt that much could be gained in terms of software portability and reuse if some common syntax could be agreed for the most useful features of the existing message passing interfaces and libraries. The preliminary draft MPI1 Standard was presented in November 1992 and prompted much discussion. At that point, a MPI Forum was set up along the lines of the earlier HPF (High Performance FORTRAN) Forum to revise and expand the new MPI1 document. The meetings of the MPI Forum were held every six weeks and several mailing lists were created for discussions concerning the MPI Forum and its sub-committees. Both the meetings and the email discussions were open to all members of the high performance computing community. Members of over 40 academic, governmental and industrial institutions contributed to the MPI Forum. The final MPI Standard [5] was released in May 1994 and was followed that June by a first list of errata and unresolved issues. An updated version of that list is kept on the World Wide Web in the MPI Standard document [1].

There is an ongoing discussion to improve the MPI Standard and to incorporate advanced features such as dynamic process invocation and parallel I/O. The existing standard supports a static process model and is defined as an API (Application Programming Interface). The API-binding for C and FORTRAN is part of the standard.

3 Elementary Concepts

The elementary concepts of the MPI Standard are the *process* and the *message*:

- Each process is associated with a unique local memory. The process can only directly access and modify its local memory.

- A message consists of the data to be exchanged, called its body, together with additional information called an envelope or header. The header specifies the recipient, whether and when to open the message as well as how to access the data in the body of the message.

The exchange of messages among processes is referred to as communication. Communication is the only means by which MPI processes can access data in non-local memory.

Processes are arranged in groups. A process group in MPI is a set of processes whose members are numbered consecutively starting from 0. The number associated with a process in a group is called its rank in the group and defines the naming of processes in point-to-point communication within the group. In addition, groups define the scope of collective communications. To decide whether a process should receive a message, additional information is required beyond that of the process rank. This information is given by the so-called communication context which provides separate "universes" within which all communications can be safely identified. It is possible, for instance, for a communication initiated by the user and a communication initiated by an underlying library to interfere. The information provided by the communication context avoids this situation and enables the correct messages to be selected. This functionality is not required for single threaded applications. Single threaded applications have only one thread of control. Incoming messages, addressed to the process, can be addressed to the active thread only.

The default ranking of processes is not always the most natural way for writing application software. Efficient data exchange patterns may be based on pipelines, hypercubes or meshes. In all these cases it is natural to think of the processes as being indexed according to the chosen topology in order to best support nearest-neighbour relations among the processes. MPI provides management functions for popular graph topologies. In MPI, logically connected processes need not run on physically connected processors. Such topologies are called virtual topologies. A communicator always has information which associates it with a process group, a communication context and a virtual topology. Communication has to use the same communicator as an argument to the corresponding MPI procedure call.

The concept of data types is incorporated into MPI mainly by the basic data types of the language bindings in C and FORTRAN. Inbuilt data types are a union of the atomic data types in these languages with a prefix `MPI_` added to their names. Defining the body of a message to be a unit stride vector is done in MPI by specifying the number of elements in the vector and the common MPI data type of these elements. Furthermore, MPI allows derived data types to be defined. An MPI derived data type specifies the format of a sequence of values and consists of the number of elements in the sequence, the MPI data type of each of the elements and the distance in bytes between elements of the sequence. Vector data types with constant stride, indexed vector data types and structures, consisting of elements of different atomic types are supplied.

MPI communication procedures are classified according to the so-called communication modes, namely *standard*, *buffered*, *ready* or *synchronous*. The default mode is the standard mode. In buffered mode the message passing system uses pre-allocated memory provided by previous calls to buffer initialisation routines. The ready mode only performs the transmission if the receiving process indicates that it is able to receive the message. The synchronous mode ensures program continuation after completion of the send and receive.

An MPI communication is blocking if a call to it will not return until it is safe for a process to re-use the resources specified in the call. Nonblocking MPI communication procedures are supplemented with procedures to test or to wait for permission to re-use the resources. Point-to-point communication in MPI is implemented in all four communication modes. Table 1 lists the various derivations of the most common function, the send operation, in these communication modes.

Communication mode	Blocking	Nonblocking
standard send	`MPI_Send()`	`MPI_Isend()`
buffered send	`MPI_Bsend()`	`MPI_Ibsend()`
ready send	`MPI_Rsend()`	`MPI_Irsend()`
synchronous send	`MPI_Ssend()`	`MPI_Issend()`
standard receive	`MPI_Recv()`	`MPI_Irecv()`

Table 1. *Standardised communication modes in MPI*

4 Implementation

The development of the MPI library for the Transtech Parastation using 3L Parallel C [6] has been rigidly restricted to the basic functionalities of MPI.

Transputer applications developed by the Structural Engineering Computational Technology (SECT) Reseach Group at Heriot-Watt University, Edinburgh are mainly based on the master-worker concept, where only one communicator, the master process, is in synchronous communication with the remaining, worker processes. Most of the communication is performed by functions which form part of an in-house function library developed by the SECT group. This was done to standardise and help the development of finite element related programs and this library is known as *E-Lib.* However, in terms of communication, these functions provide no further abstraction than the underlying 3L Parallel C communication and must still be mixed with statements giving access to low level communication primitives of the 3L system – namely the *Alt package* [6]. Functions in the *Alt package* enable a process to input from whichever of a specified number partners is first ready to communicate. Simple point-to-point communication is easily portable to standard MPI, although problems arise from the extensive use of the Alt package in the master code to multiplex the incoming worker data.

After a review of the existing routing system, the constraints on the MPI implementation and the interface for the implemented functions are given.

4.1 Transputer Routing System

The routing scheme is designed for master-worker type applications, where the communication requirements are [3]:

- Distribution of relatively large data sets from the master process to the worker processes.
- Freedom to perform data exchanges between any two worker processes for relatively small data sets.
- Communication of the worker processes with the master process for returning relatively large data sets at the completion of the computation.

Only one transputer is directly connected to the host of the Transtech Parastation and is referred to as the ROOT processor. (This is the first module in the boot pipeline of the motherboard.) Normally, only one process is capable of I/O handling (standard I/O with console and file I/O) and this process must reside on this ROOT transputer. The master task is usually chosen for this purpose. All other processes, the workers, run in the so-called stand-alone mode.

Several contexts or programs can be executed on one processor. A context itself may consist of several threads. Communication between the threads of a context are performed on unidirectional channels which can be created during run-time. Communication between contexts on the same processor is very flexible, the static network of virtual unidirectional channels being described in a communications configuration script. The *configurer* uses this configuration script and the task executables to to create the application. Communication to contexts on neighbouring transputers have to be performed on established virtual channels mapped onto physical transputer hardware links. One pair of virtual channels only, usually a bi-directional channel, can be mapped onto one physical link. Hence, an additional routing scheme is essential to exchange messages between non-neighbouring processors. Such a routing system was introduced in Reference [3].

The original SECT router [3, 4] assumes that the processors are arranged in a rectangular grid with the root processor in the top left-hand corner. The root processor contains the master process, one worker process and a master router task. All remaining processors contain a single worker and a worker router task. The master router is essentially the same as a worker router but provides communication paths between master and worker tasks on the same processor.

Every application task (master or worker process) is provided with bi-directional virtual channels to every other application task in the array. Communication with a remote application task is carried out using appropriate virtual channels as shown in Figure 1. Figure 2 and 3 describe the internal architectures of the router tasks in more detail.

As an example, consider a worker task which wishes to send a message to an application task on another processor. The message is first sent to the **worker_listener** thread of the local worker router where it is sent to the neighbouring transputer via one of the four transputer links. One of four **link_listener** threads on the neighbouring router receives the incoming message and either routes it on to another transputer via one of the transputer links or sends it to the **spooler_task** thread associated with one virtual channel. The resident application task must then request the message on this channel. Every **spooler_task** works as an input queue for one source of communication and since it is only capable of storing one message for each source-destination pair, the routing system will be blocked if more than one message is sent from a source that is not matched by corresponding receive statements at the destination application task.

In the master router, the **link_listener** thread is capable of sending messages either to the local worker or to the master process and there is no **spooler_task** to queue messages destined for the worker process.

The deterministic store-and-forward routing technique, based upon the dual of *find-row/ find-column* (i.e. *find-column/find-row*) and *find-row/find-column* [7]) algorithm has been implemented on the occasionally incomplete two-dimensional network of transputers. The algorithm is known to be deadlock-free on account of the fact that the message entering from a horizontal link always uses vertical links for onward transmission if necessary, and vice versa. Thus, as shown in Reference [7] the input-output graph in this case remains acyclic, guaranteeing deadlock-free routing.

The store-and-forward message passing strategy tends to increase the latency of the message. The router tasks are assigned higher priority with respect to the other processes, i.e. master and worker tasks. In Figure 4 the timings for the ping-pong experiment with and without the store-and-forward router are shown. The ping-pong test is sending a message from one source processor to one destination processor followed by an immediate reflection back of the same message to the source. The timing given is half of the total time of the complete cycle.

Obviously, the message latency is increased by a minimum of $160\mu s$, depending on the message size. This reduces the asymptotic transmission rate (transmission rate for very large messages) from 1.7 MByte/s to 1.2 MByte/s. The routing overhead decreases for communication between processors which are not directly connected. For 1 hop the asymptotic transmission rate is 910 kByte and 700 kByte for native and routed communication, respectively.

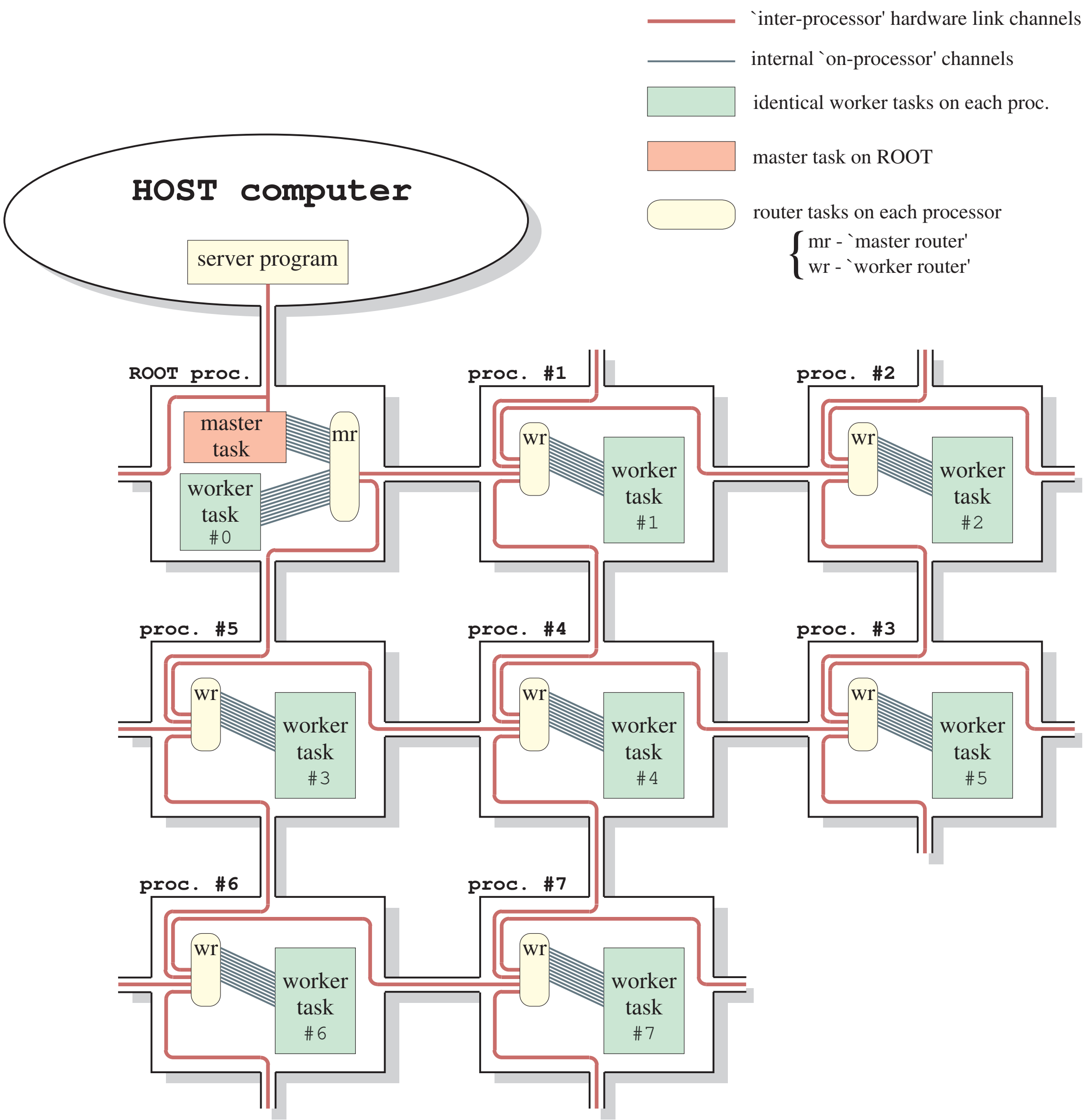

Figure 1. *Transputer routing scheme for parallel finite element analysis*

There are some limitations of the router which have to be changed for the MPI implementation:

1. The software design assumes a master-worker task and does not support the *single program multiple data* model.

2. The software configuration is restricted to a maximum number of 10 worker tasks due to the statically allocated memory of the router.

3. The only communication mode provided by the router system is blocking and fully synchronized. The router system is capable of storing one message of a fixed size.

The routing task layout was simplified to meet the requirement for a homogeneous programming model in MPI. The master logic was eliminated and the message buffering and routing algorithms were clarified. The deterministic routing scheme is now implemented in a look-up table. This look-up table is calculated once during program initialisation. This resulted in

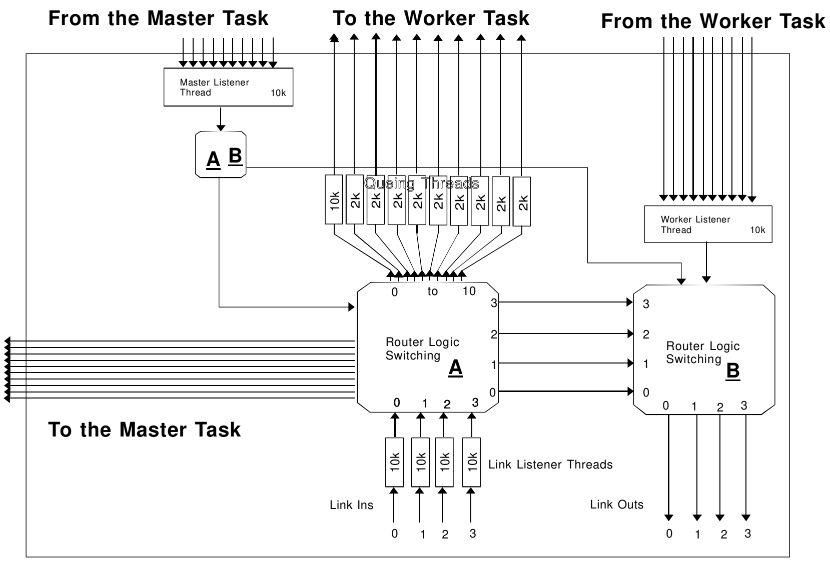

Figure 2. *Internal architecture of the master router task*

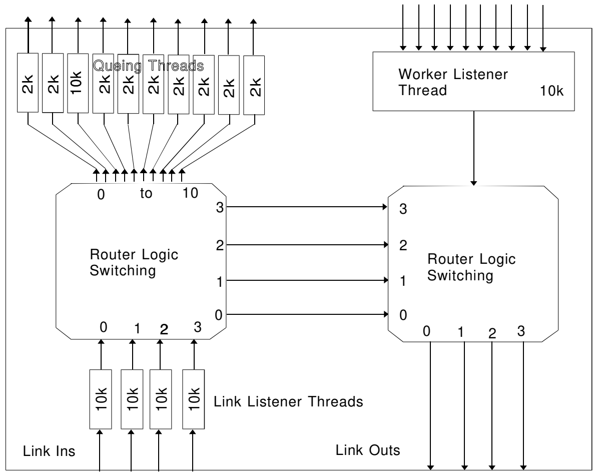

Figure 3. *Internal architecture of the worker router task*

a significant reduction of the routing time to allow additional logic for error checking of the communication functions.

The router threads are now implemented as a library as described in Figure 5. The library is initialised within the call to the initialisation of MPI. The number of processors and the layout of the processor network can be extracted from the command line parameters by the root process. The root process sends this information to all the other processes using the default boot pipeline. The required data structures for the appropriate number of routing threads can be allocated dynamically from the global heap of the application program. There is no longer the need to specify on program configuration time the amount of memory to be allocated for the send and receive buffers. Thus the number of supported processors is restricted only by the main memory making this router scheme totally scalable over any given transputer hardware.

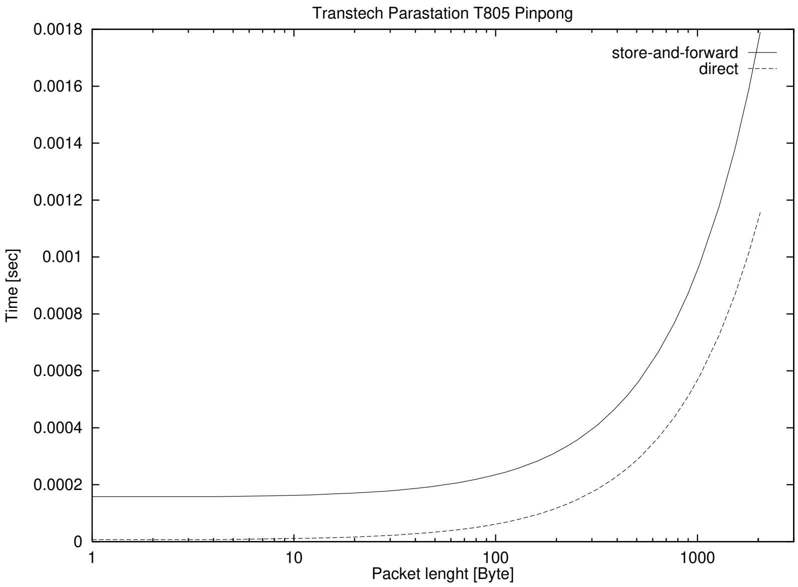

Figure 4. *Timing for ping-pong benchmark for the router and native communication*

A message queuing system was developed to support non-blocking communication modes. The queuing systems consists of a message queue and two additional threads for every communication partner `ReadDataIn` and `WriteDataOut`, respectively.

The non-blocking communication requests are stored in a double linked list. The data handled by the list provide information required to perform the transmission and flags indicating the status of the communication. The requests are consecutively numbered.

The internal structure of the router is shown in Figure 5. Send and receive threads are already launched during program initialisation. These threads are accessing the memory of the application using the information found in the message queue. The message queue item is handled in three steps. First it is aquired by the appropriate send or receive function. The status information (actual transferred message length) and the completion flag are updated after performing the transfer. The queue item is de-queued either by a consecutive function call for completion in the application program or by the message passing layer itself for self initiated communication requests. These communication requests are used for the first blocking communication following a nonblocking communication with the same partner processor allowing intermediate completion of other message transfer functions.

4.2 Implemented MPI routines

Table 2 lists all the functions which were implemented in the proposed MPI library. The following sub-sections give detailed information on each class of functions.

4.2.1 Initialisation, Status and Termination The initialisation of the MPI library for the Transtech Parastation is performed by the function:

```
int MPI_Init(int    *argc,
             char ***argv);
```

The function must be called only once in the program and would usually be the first call in the `main()` function. Command line parameters are passed to MPI to provide information on the processor network. The following command line parameters are required:

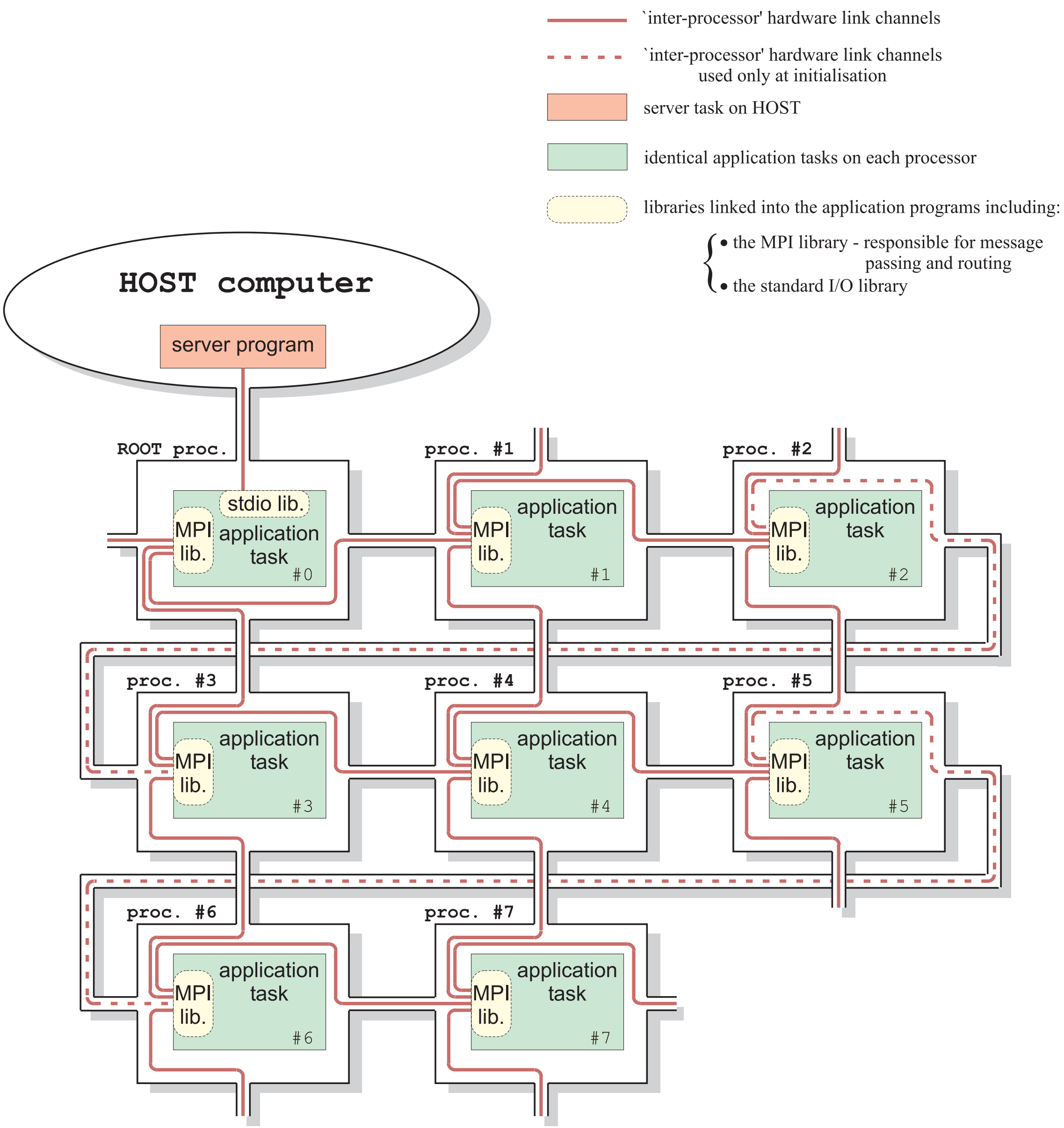

Figure 5. *The general structure of the MPI routing scheme*

-np NPROCS The number of processors in the MPI task. This number may be less than the actual number of transputers. The first NPROCS transputer will constitute the virtual machine and will run the MPI application. The remaining transputers will not be used. The processors are numbered accordingly to their order on the default pipeline[6].

-ncols NCOLS The number of columns in the two-dimensional transputer mesh. The mesh may be incomplete in the last row if NPROCS is not a multiple of NCOLS. The default pipeline has to follow the mesh row by row. But one part of the pipeline which is the connection between the last process in a row to the first process in the next row is used by MPI for initialisation only (see Figure 5).

These command line parameters are stripped from the argument vector to prevent the application program from (miss-) interpreting them.

The function initialises status variables such as the process rank and task size. These parameters are provided assuming that only one task is mapped to each processor and are therefore

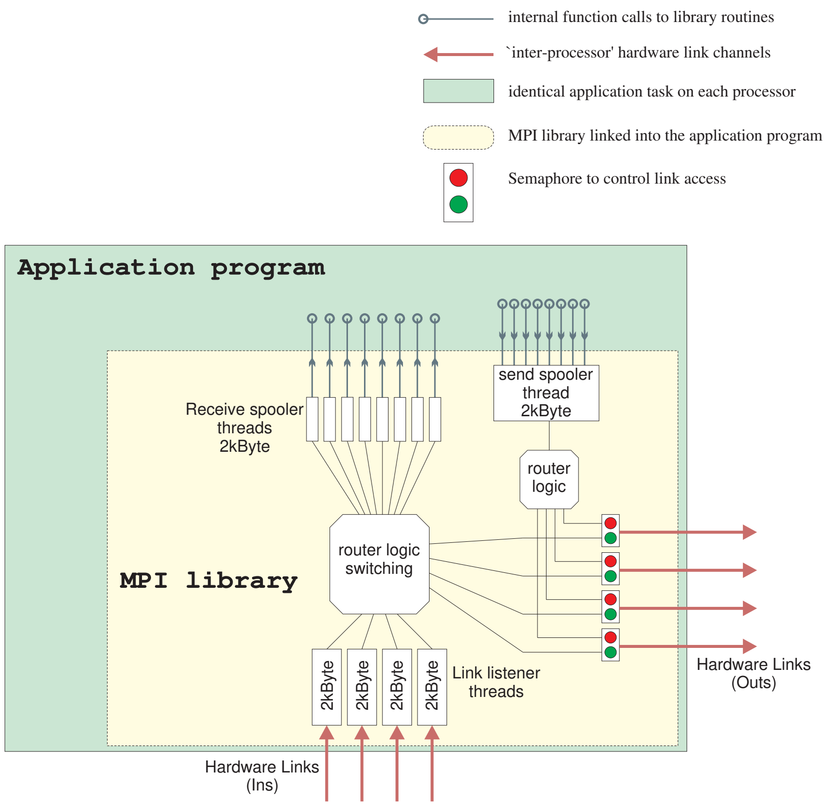

Figure 6. *The internal structure of the MPI router*

both identical to the processor number. The default communicator provided is `MPI_COMM_WORLD`. All processes are members of this communicator.

Rank and size can be obtained by the following two routines where `comm` is always `MPI_COMM_WORLD`. The functions require a prior call to `MPI_Init()`.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);

int MPI_Comm_size(MPI_Comm comm, int *size);
```

The function `MPI_Finalize()` is used to terminate MPI. The features of the transputer run-time system require that this function is used to wait on the running communication threads. The use of this call before a (safe) exit from the application program is strongly recommended. Otherwise, routed messages may not be delivered to the recipients because of the unexpected termination of required routing threads. The function sends a synchronisation token from processor 0 to all other processors. The processors are informed in decreasing order. So, the message with the longest path in the routing network is sent first. The function is implemented as:

```
int MPI_Finalize(void);
```

The MPI standard requires a timer for "performance debugging" and because existing timers are either inconvenient or do not provide adequate access to high-resolution timers. `MPI_Wtime()` returns a double precision number of seconds, representing wall-clock time since some time in the past. It is implemented as a low-priority timer and has a resolution of 64 μs.

```
double MPI_Wtime(void);
```

Functionname	Functionality or return value
Startup	
`MPI_Init()`	initialization routine
`MPI_Finalize()`	cleans up all MPI state
`MPI_Get_processor_name()`	identificator for the actual processor
Communicator accessors	
`MPI_Comm_size()`	number of processes in the group
`MPI_Comm_rank()`	rank of the calling process in group
Point-to-point communication	
`MPI_Send()`	blocking send
`MPI_Recv()`	blocking receive
`MPI_Sendrecv()`	combined blocking send and receive
`MPI_Isend()`	start a nonblocking (immediate) send
`MPI_Irecv()`	start a nonblocking (immediate) receive
`MPI_Test()`	status of a nonblocking communication
`MPI_Wait()`	wait for a nonblocking communication
`MPI_Waitall()`	wait for a number of nonblocking communications
`MPI_Waitany()`	wait for one out of a number of nonblocking communications
Collective communication	
`MPI_Bcast()`	Broadcast
`MPI_Reduce()`	Reduce
`MPI_Allreduce()`	All-Reduce
Return status	
`MPI_Get_count()`	number of received entries
Address and extent functions	
`MPI_Type_size()`	datatype size
Timers and synchronization	
`MPI_Barrier()`	Barrier synchronisation
`MPI_Wtime()`	elapsed wall-clock time
`MPI_Wtick()`	the timer resolution

Table 2. *Implemented MPI functions*

This function is capable of handling an overflow of the timer register between subsequent calls to `MPI_Wtime()`. This overflow occurs once in 72 hours. The time is not global on the system. Different processors will return different times. The resolution of the timer can be obtained by a call to

```
double MPI_Wtick(void);
```

4.2.2 Point-to-point communication functions The standard mode point-to-point communication procedures are provided in the blocking version. These routines can simply be seen as function wrappers to the underlying routing functions. Nevertheless, they provide better error handling with respect to the function parameters `count`, `datatype` and `source`. The values of the destination and source are checked. Local communication and communication to partners outside the context are prohibited.

```
int MPI_Send(void *buf,
             int count,
             MPI_Datatype datatype,
             int dest,
             int tag,
             MPI_Comm comm);
```

```
int MPI_Recv(void *buf,
             int count,
             MPI_Datatype datatype,
             int source,
             int tag,
             MPI_Comm comm,
             MPI_Status * status);
```

On output `MPI_Recv()` provides status information. `status` may be evaluated with the function `MPI_Get_count()` (see below) to obtain the actual number of transferred entities.

The non-blocking counterparts to the point-to-point communication are the standard non-blocking send and receive operations. The functions return a request, which has to be used in a subsequent call to `MPI_Wait()`. This function is required to check for safe re-use of the resources of the prior `MPI_I*()` call, mainly the data buffer. Subsequent calls to `MPI_I*()` functions with the same partner are queued. Following calls to blocking communication routines will block until the previous `MPI_I*()` functions with the same partner has been completed. The use of the procedure pair `MPI_I*()` and `MPI_Wait()` is therefore strongly recommended.

```
int MPI_Isend(void* buf,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);

int MPI_Irecv(void *buf,
              int count,
              MPI_Datatype datatype,
              int source,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);
```

The blocking wait for completion of one nonblocking communication is provided by `MPI_Wait()`. If the function was waiting for a receive request, the status object contains the informations provided by that receive operation.

```
int MPI_Wait(MPI_Request *request,
             MPI_Status  *status);
```

The function `MPI_Test()` is a nonblocking test for completion of one communication. The output parameter `flag` is true if the operation identified by the input parameter `request` is complete. The status object is set to contain the information on the completed operation.

```
int MPI_Test(MPI_Request *request,
             int         *flag,
             MPI_Status  *status);
```

The use of the nonblocking `MPI_Test()` call allows the user to schedule alternative computations within a single thread of execution. The function `MPI_Test` returns with `flag` equal to `true` exactly in those situations where the function `MPI_Wait()` returns. In such cases, both functions return the same value for status. A blocking MPI_Wait() can be easily replaced by a non-blocking MPI_Test().

Two functions for waiting for a number of communications are implemented. They return if all (MPI_Waitall()) or at least one (MPI_Waitany()) of the specified requests are completed. MPI_Waitany() returns the index in the specified request array of the completed request.

```
int MPI_Waitall(int count,
                MPI_Request array_of_requests[],
                MPI_Status array_of_statuses[]);

int MPI_Waitany(int count,
                MPI_Request array_of_requests[],
                int *index,
                MPI_Status *status)
```

4.2.3 Collective communication functions Collective communications are implemented for a group of processes. The only group handled by this MPI implementation is the group of all processes (MPI_COMM_WORLD).

The function MPI_Bcast() is used to broadcast a message from a process called root to all other processes. On input, root provides the already filled buffer parameter. On output a copy of buffer is available on all processes.

```
int MPI_Bcast(void *buffer,
              int count,
              MPI_Datatype datatype,
              int root,
              MPI_Comm comm);
```

In a reduction operation, input is sent by all participating processes to the root process which then processes the data. Two buffers are required to perform a non-commutative arithmetic operation. Therefore, the MPI_Reduce() has the following parameters:

```
int MPI_Reduce(void *sendbuf,
               void *recvbuf,
               int count,
               MPI_Datatype datatype,
               MPI_Op op,
               int root,
               MPI_Comm comm);
```

Currently, MPI_SUM, MPI_MAX and MPI_MIN are implemented for the data types MPI_INT and MPI_DOUBLE.

The execution of MPI_Reduce() on a root followed by MPI_Bcast() with the same root can be implemented more efficiently. Therefore, the function MPI_Allreduce() is provided. The function returns identical results to all participating processes. Implemented operations and data types are the same as for MPI_Reduce().

```
int MPI_Allreduce(void *sendbuf,
                  void *recvbuf,
                  int count,
                  MPI_Datatype datatype,
                  MPI_Op op,
                  MPI_Comm comm);
```

The function `MPI_Barrier()` is provided to synchronise all processes.

```
int MPI_Barrier(MPI_Comm comm);
```

4.2.4 Miscellaneous functions In some situations it is useful to specify only an upper limit for the amount of data to be received. The `count` parameter to the `MPI_Recv()` function enables this. The function `MPI_Get_count()` determines the actual amount of data received. The parameters to this function are the `status` returned from the `MPI_Recv()` call and the data type used. The data type should be the same as in the `MPI_Recv()` call.

```
int MPI_Get_count(MPI_Status *status,
                  MPI_Datatype datatype,
                  int *count);
```

The function `MPI_Type_size()` is provided to obtain the total size, in bytes, of the entries in the type signature associated with `datatype`. On output `size` is the total size of the data in a message that would be created with this data type.

```
int MPI_Type_size(MPI_Datatype datatype,
                  int * size);
```

4.2.5 Special requirements of the *3L* development environment The 3L linker uses different libraries for those processes which are connected directly to the standard I/O facilities from the remainder of the processes which are not. There is usually one task which runs on the ROOT transputer and is responsible for the I/O of the whole network with the HOST machine. This program is linked with the full C runtime library. The rest of the tasks, running on the other processors cannot call the standard I/O functions and thus they are linked with a smaller sub-set of the C libraries, the *standalone library*. To use the same source code to be compiled with both libraries additional global symbols and functions must be provided for the standalone library. These symbols are defined in the source file `stubs.c`. The compiled image of this source must be linked into the executable of the standalone tasks with the standalon library instead of the full C runtime library. The file uses the standard definitions given in `stdio.h`.

Naturally, in the multi transputer system the 3L environment allows more tasks to be placed on one transputer. As a special case, the proposed partial implementation of MPI can be configured to run on a single transputer. In this case the tasks are connected by internal on-processor software channels instead of the previously described hardware link channels. Such a system is set up by the *configurer* which is part of the 3L development environment. Beside placing the tasks onto the processors this time the configurer has to assign the software channels to them for the connectivity. This is achieved by passing some extra parameters to each task's `main()` function from the 3L runtime environment.

```
int main(int argc, char *argv[], char *envp[],
         CHAN  *in_ports[], int ins,
         CHAN *out_ports[], int outs);
```

These additional parameters provide all the information on the software channels for every task. The internal MPI routing has to have access to the communication layer of the transputer runtime system. The only way to pass this information to the router is to pass them as parameters. Therefore, in the case of a single processor configuration the function `MPI_Init()` takes two extra parameters `in_ports` and `out_ports` which are pointer vectors to the software channels:

```
int MPI_Init(int *argc, char ***argv,
             CHAN *in_ports[], CHAN *out_ports[]);
```

To switch between the single and multiple transputer configurations the target architecture is defined in the header file `config.h` by the macro definition `SINGLE_TRANSPUTER` which is a conditional statement. The definitions in `mpi.h` depend on this file. Changing the target architecture requires a recompilation of the MPI library comprising the files `mpi.c` and `router.c` and the re-linking and re-configuration of the application.

5 Performance tests

The performance of the implemented MPI library was evaluated by several tests comparing benchmark tests with the native router alone as described in [3] and the same benchmark tests implemented with MPI. It was also tested and compared with the native version of a finite element code as described in the next section.

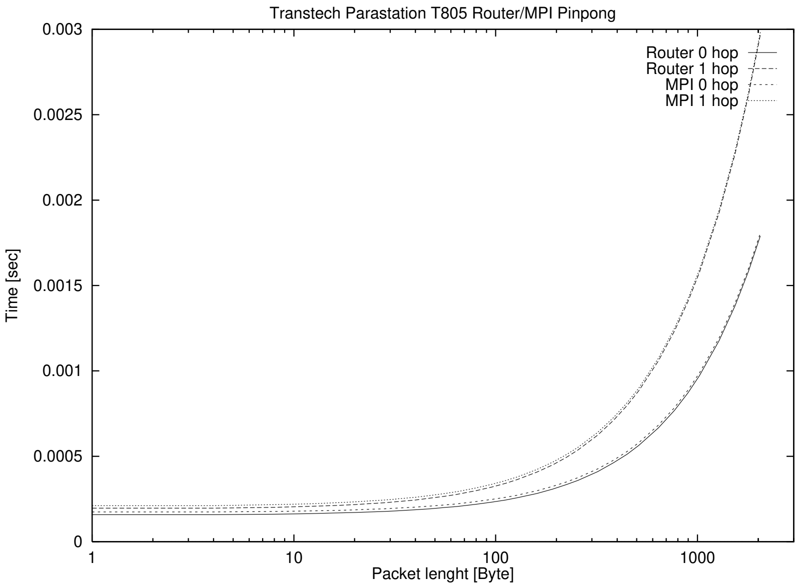

Figure 7. *Timing for ping-pong benchmark for the native router and MPI*

First, the simple ping-pong benchmark was chosen to show the marginal overhead induced by additional parameter checking and the invoking of one more function call. The increased safety of the MPI implementation (parameter checking) does not increase the communication time. This is due to several optimisations of the router implementation itself. The router now has a simpler structure and repeated alternative statements were replaced by look-up tables. Both timing and transfer rates for one and two hop connections are given in Figures 7 and 8. The buffer space for all the communication threads was large enough to hold an entire data packet.

Memory requirements of the router may be crucial due to limited transputer memory. Beside this, a thread in the 3L package is only allowed to allocate memory within the workspace allocated for the thread itself. The size of the workspace must be set before the thread is created and cannot be subsequently altered. Therefore, experiments with different router buffer sizes were performed. Figures 9, 10 and 11, 12 show the impact of the packet length on the communication throughput for 0 and 1 hop connections both for small ($\leq$ 128 Byte) and large ($\geq$ 128 Byte) message sizes, respectively. Small packets have advantages in the transfer time

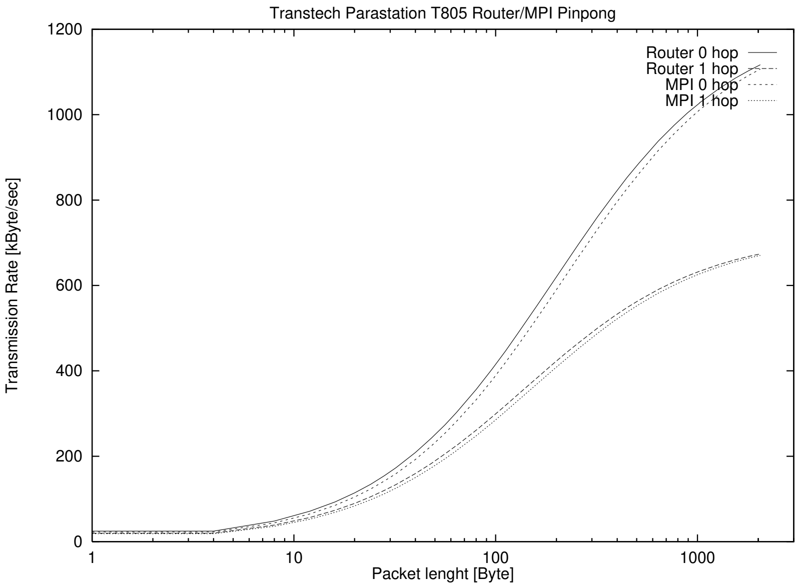

Figure 8. *Transmission rate of the ping-pong benchmarks for the native router and MPI*

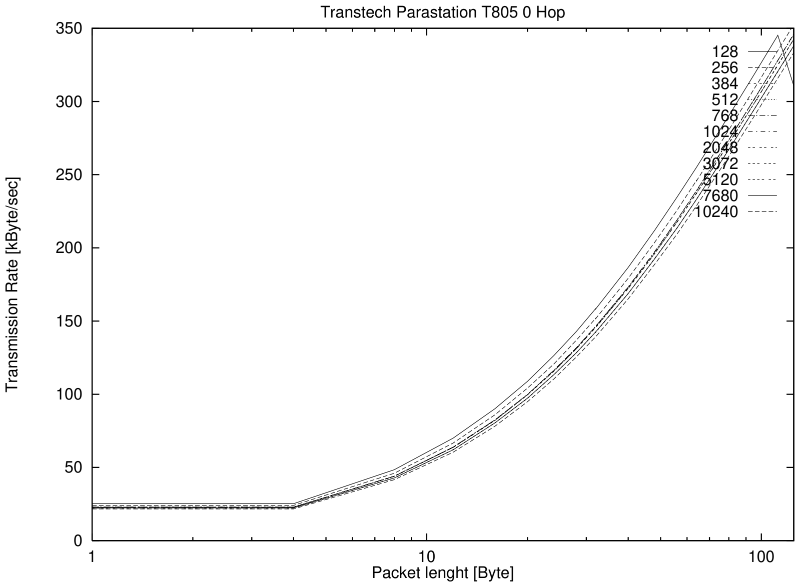

Figure 9. *Timing for MPI and router with different packetsizes, short messages, 0 hop*

between different routing tasks, but a large number of small packets induces a large overhead with respect to the start-up time. As it is shown in the diagrams, the store-and-forward routing scheme used is relatively insensitive to large buffer sizes. The trade-off against small buffer sizes is only useful for extremely large packets. SECT applications tend to have larger number of small messages and therefore a buffer size of 2 kByte was choosen for the router implementation.

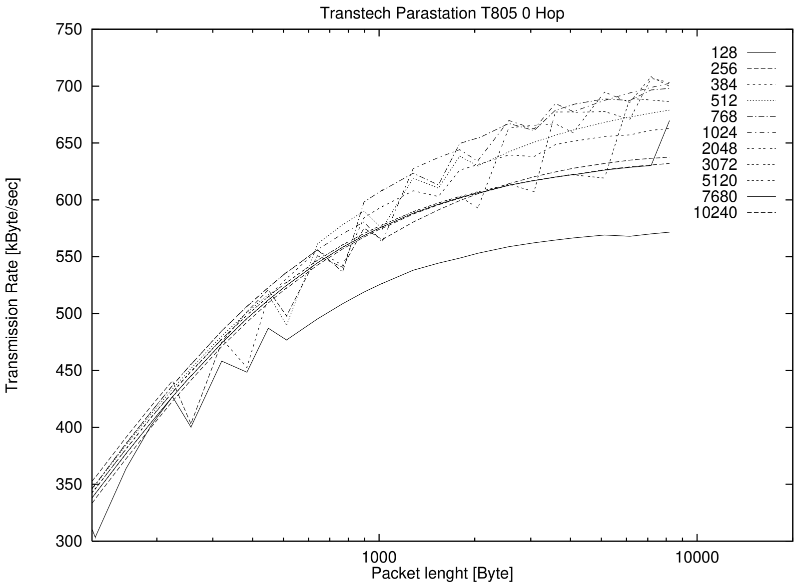

Figure 10. *Timing for MPI and router with different packetsizes, large messages, 0 hop*

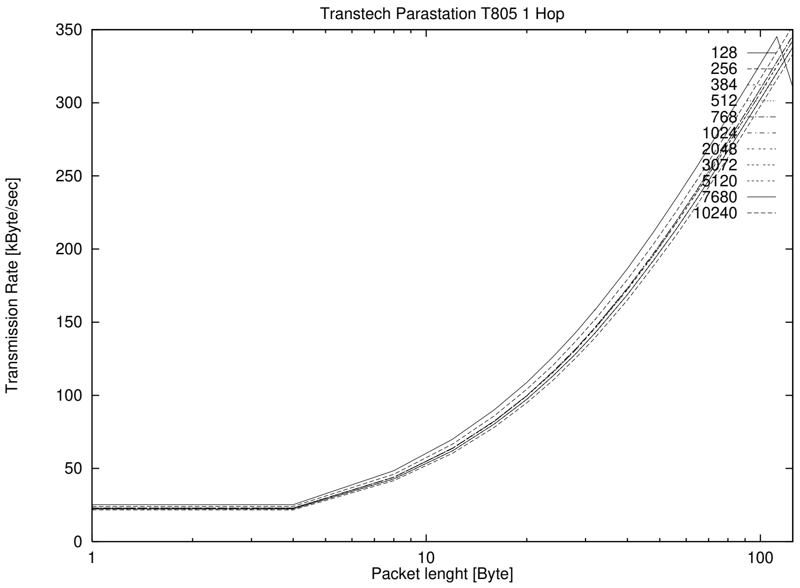

Figure 11. *Timing for MPI and router with different packetsizes, small messages, 1 hop*

6 Transition example

A parallel transient non-linear finite element analysis program which is fully detailed in Reference [8], originally written for transputer systems using the 3L development environment has been choosen to illustrate the following points:

- the transition considerations when adopting MPI for the message passing;

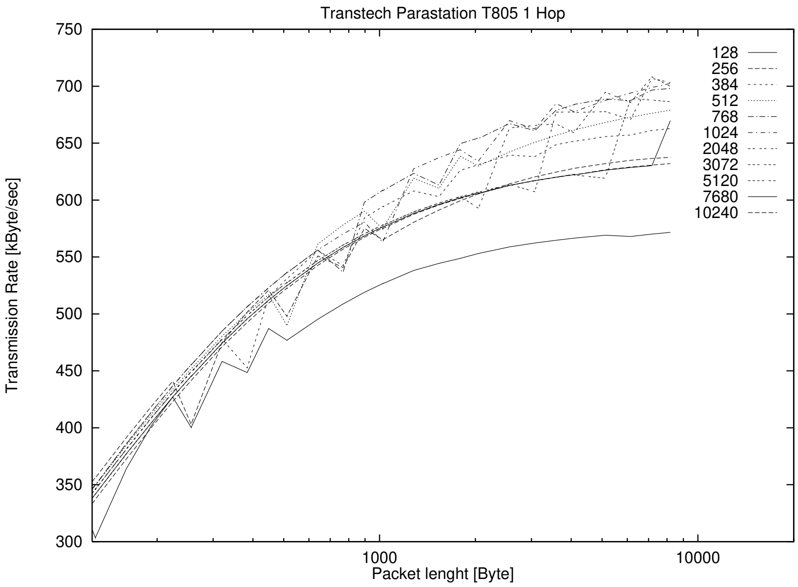

Figure 12. *Timing for MPI and router with different packetsizes, large messages,1 hop*

- a close look at a real life example to see performance advantages or trade-offs arising from the new underlying system; and
- changes in the programming paradigm and its consequences.

In its original form the finite element program utilised the transputer router task as it is explained in Section 4.1. Its source code was written in C using the 3L Parallel C extensions for the message passing functionality. The following sub-sections describe how this was replaced by standard MPI function calls and what other changes have to be made.

6.1 Programming Paradigm

When writing parallel programs it is very rare to use the full generality of the Message-Passing model and write a different executable for each processor participating the parallel system. Usually a problem can naturally be divided into sub-problems in such a way that each of these sub-problems could be solved in broadly the same way. That is exactly the case with this explicit parallel finite element code. The finite element mesh is divided into sub-domains and the same code with the same algorithm is to be run on each. A communication need only arises at the neighbouring boundaries to synchronise the calculation, i.e. to gather element contributions from each sub-domain. In this case typically the programmer wants to write only one code and replicate it across the multiple processors. This is referred to as the *SPMD* model which is an acronym for *"single program multiple data"* and can be considered as a restriction on the message passing paradigm by requiring that all processes run the same executable.

This MPI implementation was designed to support the SPMD model. The first change in the program required is the removal of the master task which was originally built for the master-worker concept of this transputer environment. However the elimination of the master task requires very little effort because it is relatively small in both size and functionality. It has been incorporated into the general overall executable with the use of a few `if()` statements.

It has to be emphasized here, that with this little effort, the use of MPI makes the program portable across almost every special or dedicated parallel hardware platforms, where the hardware vendors often provide parallel environments which only support SPMD parallel programs. Many times, for related reasons of efficiency, some vendors do not allow time-sharing i.e. multiple tasks or processes per processor (some vendors understand the term SPMD to include this further restriction). This latter case has a special consiquence for the programmer. To place a task on a processor which has a very limited load and functionality, such as the master task for the parallel finite element analysis, may be inefficient because the whole processor will be engaged with that process.

6.2 Transition considerations

Thus all the functionalities of the master task, i.e. the I/O support and general administration with pre- and post-processing, have been incorporated into the overall executable, which was generated from the original worker task. This was accomplished by introducing `if()` statements in the program code and thus allowing for MPI process number `#0` to take up the role of the original master task. This process physically resides on the ROOT transputer having access to the necessary I/O services. This was the first consideration in the transition which affected the structure of the program as a whole. There are three other aspects of the transition which are to be mentioned:

- the initialisation;
- the basic function replacement; and
- the general structure of the application and its realisation.

The introduction of the changes neccessary at the program initialisation are straight forward and explained in Section 4.2.1. Only one further feature has to be discussed here. Each task now identifies itself through a simple function call to `MPI_Comm_rank()` rather than reading integer values which were bound to their software channels.

To carry out the function replacement, the different types of communications used within the FE program were considered. First of all there is the transfer of all the finite element mesh data to the processors at the pre-processing stage and the sending back of results to the ROOT transputer for filing. Both of these tasks are realised by series of point-to-point communication instructions. These were simply replaced by their MPI counterparts (`MPI_Send()` - `MPI_Recv()` pairs).

The majority of the message passing is done by calls to the *communication routine* also detailed in Reference [8]. This sub-routine has been written in a general way to carry out the communication between the neighbouring sub-domains and it can be called separately for each system vector to be synchronised within each iteration. It deals with groups of finite element nodes which induce communication with groups of other processors. In its core it collects all the data to be sent to one processor into a buffer on one side and waits for, receives and processes all these buffers on the other side in a sequence of steps illustrated in Reference [8]. Whether in a particular FE node group connection relationship [8] which side of the communication the actual process resides has been decided in advance at the pre-processing stage and is stored in small administration look-up tables. This has been implemented in MPI by building blocks of intstructions for these steps using simple `MPI_Send()` calls in one side and `MPI_Irecv()` - `MPI_Waitany()` function call pairs on the other side.

The third type of communication in this explicit finite element code is required by the energy balance check which is part of the algorithm to make sure that the explicit integration formulae still maintaines its stability at any one time throughout the iteration. That has been carried out on the master task and is now part of the overall code and executed on MPI process `#0`. The communication involved is typically a collective type communication, but it has been implemented simply with point-to-point type function calls as direct translations of the 3L Parallel C code.

It has to be mentioned here that due to the severe memory limitations of the original transputer router the basic 3L Parallel C channel functions had to be called through another function. This function provided the neccessary buffering and in case of longer buffers the slicing or packeting of these buffers ready to go through the routers. Thus the problem was solved at the user/programmer level. As in the case of the MPI library, the MPI functions, with the router logic, have ready access to the memory area of the whole process there is no need for such a function.

Further simplifications may be noted by looking at the general build up of the application. After the compilation of the only one source code the executables have to be linked with the relevant libraries. Because of the nature of the transputer hardware there has to be a different executable for the ROOT transputer than for the rest of the processors as it is described in Section 4.1 and 4.2.5. (But now this is totally hidden on the programming level.) MPI process `#0` which will be placed onto the ROOT transputer is linked with the full C runtime library whilst all the other proccesses are linked with the standalone library and **`stubs.c`**. Obviously all of them are linked with the MPI library. Then the executables created such a way will be incorporated into the application by the *configurer* with the aid of the configuration script. This script describes their placement of these tasks to the processors and their connectivity. The configuration script used for this finite element program is shown in Figure 13. This script directly represents the connection scheme shown in Figure 5. Because of the similarity of the old transputer application (shown in Figure 1) the executable generated for the MPI process `#0` is called *master* in the configuration script and the executable for the rest of the processes is called *worker*. The same configuration script for the old transputer routing scheme is much longer and much more complicated due to the multiple tasks per processor arrangement and the neccessary software link connectivity.

6.3 Timing results

A reinforced concrete panel was idealised in Reference [8] using 208 triangular finite elements. This original idealisation was decomposed into two sub-domains of 104 elements, four sub-domains of 52 elements and eight sub-domains of 26 elements. The idealisation is shown in Figure 14 with the three decompositions illustrated.

The numerical analysis was carried out for a small part of the simulation process using the original sequential version of the program and the execution time was measured, t_{seq}. Subsequently the parallel program executed the same number of iterations on the decomposed meshes. The parallel execution times, t_{par}, for the three different decompositions were measured and the speed-ups, S, and the efficiencies, E, were calculated.

Table 3 shows these results achieved by running the original 3L Paarallel C program for 25 iterations as a benchmark for the finite element mesh using the transputer router scheme. Table 4 shows the same results for the same mesh but here the MPI library and routing scheme was employed.

It can bee seen from Table 3 and 4 that the MPI version of the program runs generally 1% faster. This results in an increase in the efficiencies of approximately 1%. Thus it is clear that perfomance-wise there is definitely no draw-back of using MPI there is even a little gain. This is due to the enhancements in the router scheme and the generally slimmer application structure.

The precise MPI timing functions enable detailed profiling of the program. Figure 15 shows the amount of time spent for each sub-task by every processor within one iteration. It is expressed in terms of percentage where 100% is the total time spent on one complete iteration step. The full description of these tasks may be found in Reference [8].

```
! 3L configuration file for 8 processor MPI appl.

processor  host
processor  root
processor  p01
processor  p02
 .
 .
processor  p07

wire ? host[0]   root[1]
wire ? root[2]    p01[1]
wire ?  p01[2]    p02[1]
wire ?  p02[2]    p03[1]
wire ?  p03[2]    p04[1]
wire ?  p04[2]    p05[1]
wire ?  p05[2]    p06[1]
wire ?  p06[2]    p07[1]
wire ? root[3]    p03[0]
wire ?  p01[3]    p04[0]
wire ?  p02[3]    p05[0]
wire ?  p03[3]    p06[0]
wire ?  p04[3]    p07[0]

task afserver ins=1  outs=1
task filter   ins=2  outs=2  data=10k
task w0  file = master  ins = 6  outs = 6 data = ?
task w1  file = worker  ins = 6  outs = 6 data = ?
 .
 .
task w7  file = worker  ins = 6  outs = 6 data = ?

! afserver to filter connections
connect ? afserver[0]   filter[0]
connect ?   filter[0] afserver[0]
! filter to m connections
connect ? filter[1] w0[1]
connect ? w0[1] filter[1]
! w0 connection to w1 via link RIGHT(3)-LEFT(0)
connect ?  w0[5]  w1[2]
connect ?  w1[2]  w0[5]
! w0 to w3 connection via link DOWN(2)-UP(1)
connect ? w0[4]  w3[3]
connect ? w3[3]  w0[4]
! w1 to w2 connections via link RIGHT(3)-LEFT(0)
connect ? w1[5] w2[2]
connect ? w2[2] w1[5]
 .
 .
 .
! w6 to w7
connect ? w6[5] w7[2]
connect ? w7[2] w6[5]
! w4 to w7
connect ? w4[4] w7[3]
connect ? w7[3] w4[4]

place afserver host
place filter root
place w0  root
place w1  p01
 .
 .
place w7  p07
```

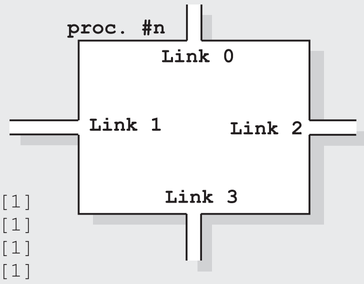

Figure 13. *Configuration script used for the MPI version of the parallel FE program running on 8 transputers*

This representation assumes that there is a syncronisation at the end of each iteration step by the energy communication. This assumption is correct as the length of that communication is very short and the collective nature of it forms a barrier within the flow of the program for every processor.

Timing the native and the MPI versions gives similar results for both. But in the MPI

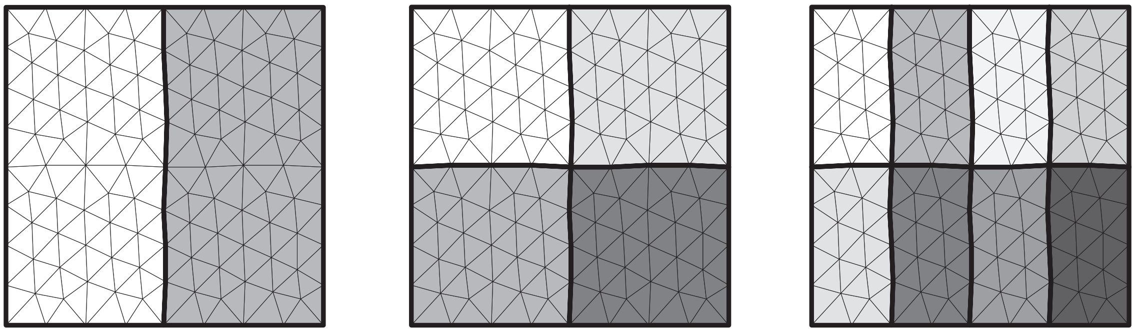

Figure 14. *The finite element idealisation of the reinforced concrete panel divided into two, four and eight sub-domains*

Num. of Proccessors N	Exec. Time t_{par}	Speed-up S	Efficiency E
1 - Sequential run	332.4s ($=t_{seq}$)	n/a	n/a
2	168.9s	1.97	98.40%
4	84.3s	3.94	98.58%
8	42.3s	7.86	98.23%

Table 3. *Speed-up values and efficiencies for the parallel system using the original 3L Parallel C code and transputer routing based on 25 (linear) iterations*

version considerably more of context checking, communication safety etc. is performed, which gives more comfort in testing and running an application with unknown datasets. These self checking routines are part of the MPI-library. They are not part of the router.

7 Conclusions

A partial implementation of the MPI specification has been developed for the Transtech Parastation. This allows programs written in the "native" 3L Parallel C to be transferred into

Num. of Proccessors N	Exec. Time t_{par}	Speed-up S	Efficiency E
1 - Sequential run	332.4s ($=t_{seq}$)	n/a	n/a
2	166.9s	1.99	99.6%
4	83.7s	3.97	99.3%
8	42.0s	7.91	98.9%

Table 4. *Speed-up values and efficiencies for the parallel system using the proposed MPI library and routing technique based on 25 (linear) iterations*

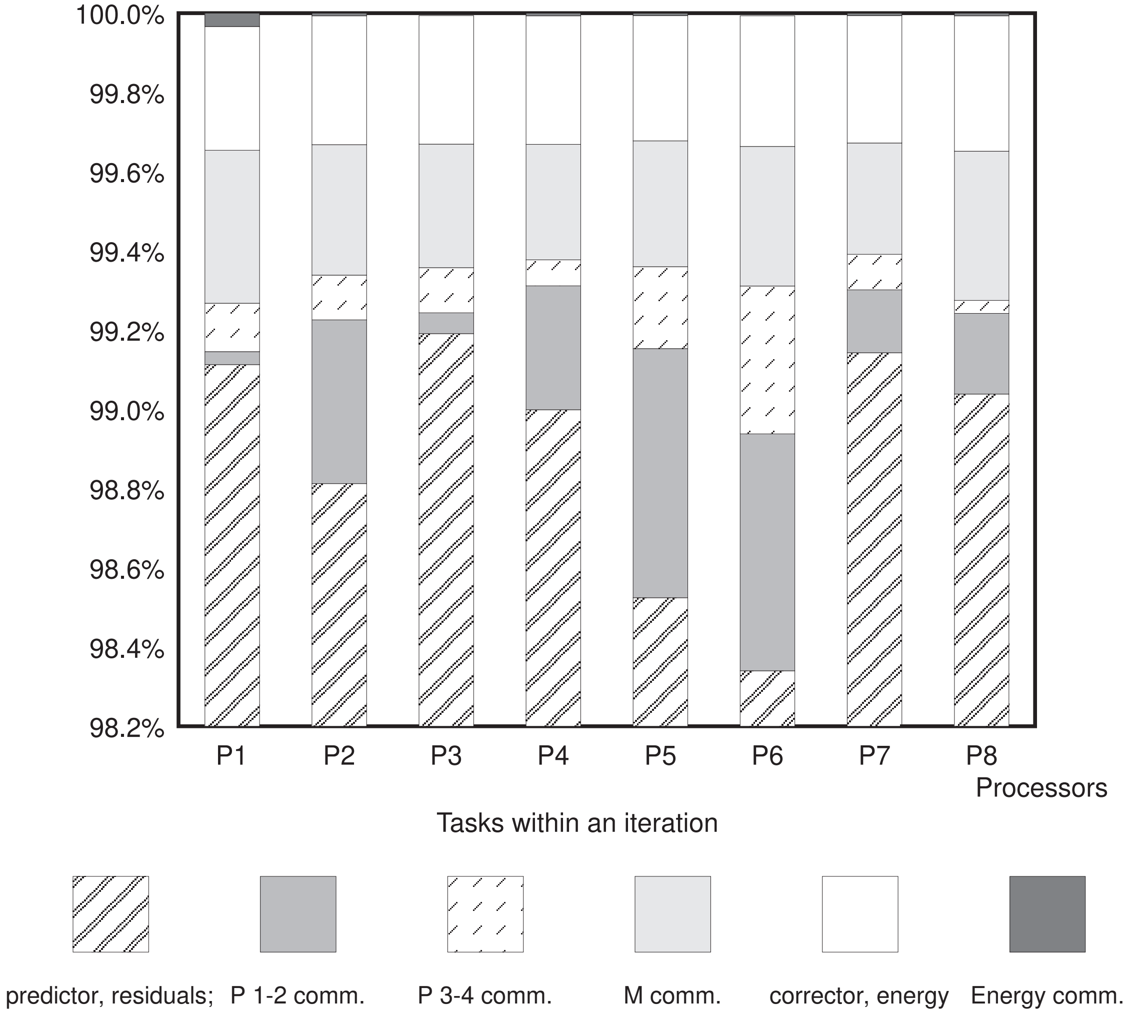

Figure 15. *The detailed profiling within one iteration step of the parallel FE program*

MPI compliant parallel programs. It was shown that the process of transition is very straight forward. The result is a well organised standard program with a higher level of hardware abstraction. There is even an efficiency bonus because of the new, simplified but totally scalable router scheme employed for the MPI library. The most important advantage is that the modified codes become immediately portable to virtually any parallel high-performance computing hardware.

Acknowledgement

The research described in this paper was made possible by the British-German Academic Research Collaboration (ARC) Programme (contract: ARC 834).

References

[1] MPI Forum, *"MPI: A Message Passing Interface Standard"*, Version 1.1, June 1995
World Wide Web address:

http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html

[2] W. Gropp, E. Lusk, *"An abstract device definition to support the implementation of a high-level point-to-point message-passing interface"*, Tech. Report MCS-P342-119, Mathematics and Computer Science Division, Argonne National Laboratory, October 1994.

[3] A.I. Khan, B.H.V. Topping, *"A Transputer Routing Algorithm for Non-linear or Dynamic Finite Element Analysis"*, Engineering Computations, Vol. 11, pp. 549-564, Pineridge Press Ltd, Swansea, UK, 1994.

[4] B.H.V. Topping and A.I. Khan, *"Parallel Finite Element Computations"*, Saxe-Coburg Publications, Edinburgh, UK, 1996.

[5] MPI Forum, *"Document for a standard message-passing interface"*, Tech. Report CS-93-214, University of Tennessee, November 1993, Available on **netlib**.

[6] *"Parallel C User Guide (compiler version 2.2.4), Reference Manual"*, 3L Ltd., Livingston, UK, 1991.

[7] U. D. Carlini, U. Villano, *"The routing problem in transputer-based parallel systems"*, Microprocessors and Microsystems, Vol. **15** No. 1, pp. 21-33, Jan-Feb 1991.

[8] J. Sziveri, B.H.V. Topping, P. Iványi, *"Parallel Transient Non-linear Analysis of Reinforced Concrete Plates"*, Advances in Computational Structures Technology, pp. 383-395, Civil-Comp Press, Edinburgh, UK, 1996.

Contents

1 Introduction **1**
1.1 The Parastation hardware 2

2 Message Passing Interface (MPI) **3**
2.1 Forming of the MPI Standard 3

3 Elementary Concepts **3**

4 Implementation **5**
4.1 Transputer Routing System 5
4.2 Implemented MPI routines 9
4.2.1 Initialisation, Status and Termination 9
4.2.2 Point-to-point communication functions 12
4.2.3 Collective communication functions 14
4.2.4 Miscellaneous functions 15
4.2.5 Special requirements of the *3L* development environment 15

5 Performance tests **16**

6 Transition example **18**
6.1 Programming Paradigm 19
6.2 Transition considerations 20
6.3 Timing results 21

7 Conclusions **23**

List of Figures

1 Transputer routing scheme for parallel finite element analysis 7
2 Internal architecture of the master router task 8
3 Internal architecture of the worker router task 8
4 Timing for ping-pong benchmark for the router and native communication . . . 9
5 The general structure of the MPI routing scheme 10
6 The internal structure of the MPI router 11
7 Timing for ping-pong benchmark for the native router and MPI 16
8 Transmission rate of the ping-pong benchmarks for the native router and MPI . 17
9 Timing for MPI and router with different packetsizes, short messages, 0 hop . . 17
10 Timing for MPI and router with different packetsizes, large messages, 0 hop . . 18
11 Timing for MPI and router with different packetsizes, small messages, 1 hop . . 18
12 Timing for MPI and router with different packetsizes, large messages,1 hop . . . 19
13 Configuration script used for the MPI version of the parallel FE program running on 8 transputers 22
14 The finite element idealisation of the reinforced concrete panel divided into two, four and eight sub-domains 23
15 The detailed profiling within one iteration step of the parallel FE program . . . 24

List of Tables

1 Standardised communication modes in MPI 5
2 Implemented MPI functions 12
3 Speed-up values and efficiencies for the parallel system using the original 3L Parallel C code and transputer routing based on 25 (linear) iterations 23
4 Speed-up values and efficiencies for the parallel system using the proposed MPI library and routing technique based on 25 (linear) iterations 23