# ParaPET
# A Parallel Programming Environment Toolkit

## P5404 GP MIMD

Our Ref: GPMIMD:003/PPET/OVIEW

Revision 1.8

Mark Debbage and Mark B. Hill

November 3, 1992

## 1 Introduction

Parallel programming is in a state of flux. The choice of parallel programming paradigms and the presentation of concurrency to the applications programmer are still matters of intense research. Fortunately the cost of prototyping and experimenting with programming environments can be dramatically reduced using a low-level toolkit. Furthermore, standard low-level interfaces enable the rapid porting of higher-level code to new platforms.

ParaPET, the Parallel Programming Environment Toolkit, has been developed to fulfill this role. It is based upon the lightweight communications infrastructure of the Virtual Channel Router[1] (VCR) package. The VCR system is readily available from the University of Southampton Parallel Application Centre and is distributed for a nominal cost to cover administration and media overheads[1]. VCR presents a distributed occam programming environment which dramatically simplifies the configuration of concurrent codes onto transputer-based parallel computers. The programming interface to ParaPET is quite different; it provides a message-passing Single Program Multiple Data model (SPMD) supporting full C run-times on every processor in the network. The environmental engineer is responsible for transforming this all-purpose model into whatever programming environment is required. At present only transputer networks fronted by the Inmos C Toolset[3] are supported. However, the interfaces exported to the environmental engineer have been chosen to enable ports to other platforms.

User interfaces based on concurrent library routines to C map directly into ParaPET, and this mode of operation is envisaged as the most common usage of ParaPET. Parallel programming environments of this type include (in alphabetical order) ASI UBIK, CHIMP, CS-Tools, Express, Intel messaging libraries, PICL and the Reactive Kernel. It is possible, however, to hide the function calls using pre-processing to deliver language extensions or even parallel languages implementations.

The package splits cleanly into several modules, any combination of which can be used during the development of the programming environment.

This work has been funded by Esprit projects P2701, Parallel Universal Message-passing Architectures, and P5404, General Purpose MIMD.

[1]Contact: University of Southampton Parallel Application Centre, 2 Venture Road, Chilworth Research Centre, Southampton, SO1 7NP. Tel +44 703 760834. Fax +44 703 760833.

# 2 The Basic Model

The ParaPET system allows the environmental engineer to execute a standard C main on every processor. The input file to the system is therefore a fully-linked seperately compiled C module with a single entry point. These can be produced as the direct output of the linker. The system uses a Type 1 `occam`-equivalent interface to these units, thus hiding the complexity of the C run-time initialisation. The necessary linker runes to construct these units are contained within the linker indirect file `parapet.lnk`.

ParaPET is distributed as an integral part of VCR commencing with version 2.0h and requires no further installation than the VCR system itself. A ParaPET program can be invoked by,
`parapet <topology> <program>`
where `<topology>` is the target VCR topology name and `<program>` is the appropriate fully-linked C module.

Each of these C processes has access to the standard C library functions, including those for device input/output. Requests to the same resource are interleaved in an arbitrary fashion unless controlled by explicit synchronisations. The basic model hides all concepts of node adjacency although this information can be recovered if required for efficiency purposes. All user supplied command line parameters are passed to every processor.

Typically the `main` entry point will be exposed by the library which calls the user code by environment specified entry points. For example an SPMD library might have a single entry point per processor to the user code (which might be spawned many times to introduce parallel slackness), while a host-node library would have two entry points to the user code: one for the user supplied host function, the other for the code executing on the nodes. Termination of the ParaPET environment can instigated by any processor returning from its C main. Any desired synchronisation between processors before termination must be realised above ParaPET.

Since ParaPET adopts an SPMD model, user code loading is entirely static over the duration of an application with each processor having an identical copy of all of the code. A library to permit dynamic code loading is feasible however.

Static variables within the C code have independent state on each processor in the ParaPET network. The organisation of the static area is guaranteed to be identical on each processor. Thus although static pointers themselves are not portable between processors, differences between static pointers are. For example, function pointers can be transported across the network by making them relative to a known static name (such as `main`).

# 3 The Low-level Libraries

This section describes fairly briefly the libraries available with the ParaPET distribution. ANSI C libraries are provided by the Inmos C Toolset. The virtual processor interface (VPI) forms the core of ParaPET's message-passing interface. Additionally, VPI exposes a low-level scheduling interface to insulate higher layers from the implementation of processes. The universal packet routing interface (UPRI) can be referenced to enable the direct synthesis of communication protocols. The Inmos specific libraries, supplied with the Toolset, such as the process, local channel and semaphore libraries, may also be used. The linker indirect file, `parapet.lnk`, lists all of these libraries and should be included into the `.lnk` file of the environment.

## 3.1 The Virtual Processor Interface

The virtual processor interface (VPI) provides a message-passing model exploiting the virtual channel facilities of VCR. The interface contains an implicit all-to-all interconnection of channels to permit any processor to efficiently invoke action on any other processor. This mechanism can be used to implement the desired communication protocols or can be used to propagate information to initialise custom protocols built through the UPRI.

### 3.1.1 Processor Numbering

VPI numbers the processors in the network using unique contiguous integers from 0 upwards. In the current implementation, number 0 identifies the root processor which is attached to the host and its filing system. The number of processors in the network is returned by `NumProcs`, while the identity of the current processor (in the range of `0` to `NumProcs()-1`) is revealed by `ProcId`.

```
int NumProcs (void) ;
int ProcId (void) ;
```

Figure 1: VPI Processor Numbering

### 3.1.2 Client-Server Interface

The client-server interface exposes access mechanisms to an all-to-all interconnection of virtual channels. Each processor pair in the network has a pair of interconnecting virtual channels, one in each direction. One end of each of these channels is denoted the server end and the other the client end. In typical use, all of the server ends at a particular processor are attached to a server process which sequentially services requests arriving at these ends in their order of arrival. Client ends may be shared between multiple processes on a particular processor thus permitting many processes on many processors to invoke action in the server of any other processor. A future extension will enhance the interface to enable multiple parallel servers on each processor.

The channels comprising this arrangement are synchronous point-to-point connections, and are actually invisible in the interface. They have the potential for server action at one end, this is implemented by a request queuing mechanism for efficiency. The first communication over the client-server channel must be directed from client-to-server in order to generate a server request.

Each client-server interaction is built from the calls in Figure 2. The server side of a transaction

```
int GrantService(void);
int ServerIn (void *message, int length);
void ServerOut (void *message, int length);
void EndService (void);

void ClaimChannel (int target);
void ClientOut (int target, void *message, int length);
int ClientIn (int target, void *message, int length);
void ReleaseChannel(int target);
```

Figure 2: VPI Client-Server Interface

commences by granting its service. The server blocks until a request is received, at which point the function returns the requesting processor id. A series of communications are then entered into, the first of which is always an input at the server end. On completion of the transaction the server terminates its service. Conversely, at the other end, a client first claims a channel to the server on the required processor (this may block to perform local sharing of the server). A complementary series communications are then entered into, the first of which is always an output at the server end (this may block to perform global sharing of the server). On completion of the transaction the client releases its channel. Client-server transactions where both client and server are located on the same processor are permitted.

Each message sent over the channel may be of variable length in the sense that it may be unknown to the receiver at the point of its input call. It is necessary for the receiver to specify a buffer of ample length for any expected message; the receiver is informed of the actual length through its return parameter. Reception of data exceeding the receiver buffer length gives a negative result to the input and the contents of the message buffer are undefined.

#### 3.1.3 Creating Dynamic Virtual Channels

The VPI provides mechanisms to create new duplex virtual channels, these are of type `VC`. No selection mechanism, neither alternation nor client-server, is available with these channels. These are connected together through the manipulation of location-independent virtual channel end structures, `VCE`s. State for virtual channel ends is declared by the caller and then initialised by a call to `NewVCE`. The resulting information must be transferred between processors in order to connect two virtual channel ends together to form a virtual channel using the call `ConnectVCE`. The first parameter to this function must be a local channel end, and the function must be called twice on the two involved processors with identical, but swapped, parameters. Local channels can be created with these calls if desired. No additional synchronisation is required other than that inherent in the swapping of channel ends between the involved processors.

A connected channel end can be refined into a channel through a call to `GetVC`, at which point the channel ends themselves may be discarded. Variable-length point-to-point synchronous communication may proceed over these channels using matched calls to `Send` and `Recv`. Each channel is fully duplex and can support independent concurrent messaging in each direction. The state associated with a channel can be relinquished through calls to `FreeVC`. No communications are permitted over a channel where one end has been deallocated.

```
typedef void *VC ;
typedef struct
{
  ...  implementation specific
} VCE ;

int  NewVCE     (VCE *vce) ;
void ConnectVCE (VCE *source, VCE *destn) ;
VC   GetVC      (VCE *vce) ;
void FreeVC     (VC vc) ;
void Send  (VC vc, void *message, int length) ;
int  Recv  (VCvc, void *message, int length) ;
```

Figure 3: VPI Dynamic Channel Interface

#### 3.1.4 Run-Time Error Function

There is one further function in the VPI:

```
Error( char *file, int line, char *message )
```

This function conveys a system error message to the console and then terminates the calling thread. It does not, however, terminate the `iserver` which continues to correspond with any threads which are still executing.

### 3.2 The Threads Library

This library provides entry points to a light-weight scheduler which should be portable to a variety of platforms. The package uses the Inmos microcoded-scheduler by default. This can be readily changed to a software version by an appropriate undefine in the `threads.h` file and recompilation of the VCR system.

The library manipulates `process` structures, these should not be confused with the `Process` structures used by the Inmos process library. The `process` structure has a `next` field defined as a pointer to a process which allows queues of processes to be constructed. The next pointer cannot be used when the process is waiting for execution since it is used to form the scheduling queue. The rest of the structure is used to hold the context of a process and is implementation dependent.

The primitives given in Figure 4 are used to manipulate `process`es. The scheduler is initialised by a call to `init_scheduler` which must be made before any subsequent calls from this library. The two other calls `reschedule` and `deschedule` permit another process to be restarted or the current one to be stopped respectively. Typically, whenever a process is to be blocked, a `process` structure is taken from the stack and a pointer to this structure stored in some shared data. The process can be restarted later by a suitable rescheduling call. This organisation permits an empty process reference to be designated as a pointer containing `NULL` rather than a special process constant.

```
void init_scheduler(void)
void deschedule(process *me)
void reschedule(process *you)
```

Figure 4: The Thread Primitives

## 3.3 UPRI Messaging Primitives

The Universal Packet Router provides a simple interface to a packet-routing facility for global communications in a locally-connected message-passing machine. Although the VPI provides a significant abstraction away from this low-level, it is often advantageous for the environmental engineer to synthesis new message-passing protocols. The packet-level interface can thus be exposed directly by including the `upri.h` header file.

The description of the interface is kept short since it is has been described in full detail in [2]. Sufficient information is retained here to give the flavour of the interface.

### 3.3.1 The Interface

The Universal Packet Router Interface is a low-level facility which has little in-built protection from abuse. It is intended for use by system engineer's who should protect it from the programming errors of the applications which it supports by careful coding of the protocol and environmental services. The Interface is only *safe if used as directed.*

Three key objects, packets, queues and ports are manipulated using the interface. The *packet* is the unit of routed data. It consists of two parts, the header contains sufficient information to specify the destination and length of the data, the body contains the data and may be empty. Packets are injected into the network by placing them on a *queue*. Since the Interface allows many independent streams of packetised data to be targeted at the same processor, a mechanism is required to segregate packets once at their destination. This is accomplished by directing packets at network *port* addresses. These are represented as destination processor and port number pairs.

Packet headers are manipulated through a set of functions, these functions have been abstracted into a single module and therefore allow the header format to be modified easily and independently of the Interface. This is vital for portability as the ideal size and format of the header is critically dependent on the machine architecture.

A series of functions are provided which allocate port numbers and perform allocation and initialisation of port data structures. All port allocation proceeds dynamically at run-time, clearly for any communication to take place valid port numbers on other processors must be known. This is achieved by ensuring that during the environment initialisation call each service allocates the the same number of ports on every processor. Thus the remote port numbers for these services will be the same as those allocated on remote processors.

At run-time each processor makes two calls to higher kernel levels. The first is made after the router is initialised, but before any packets are accepted from the network. This call informs the protocol layer of the processor id and the number of processors in the network. It also gives details, on every processor, of any command-line parameters; potentially allowing the user to select between a variety of programming environments via a command-line switch. This entry-point should be used to initialise the static variables associated with protocol layers and to allocate port numbers. Upon return of this call message-passing is commenced and the second call made to the protocol layer, the only parameters passed to this call are the command line arguments.

### 3.3.2 Packet Departure

A packet is introduced to the network by enqueuing an *output request block* (`ORB`, see Figure 5) on an outgoing communication resource. An `ORB` contains a packet header and a pointer to a packet body.

```
typedef struct
{
   BYTE    *bffr;    /* Packet buffer pointer    */
   HEADER  header;   /* Packet Header            */
   ...               /* Implementation specific  */
} ORB ;
```

Figure 5: The ORB data structure.

The packet departure library is illustrated in Figure 6. To send a packet an instance of the `ORB` data structure must be declared and its fields initialised with appropriate header information and data address,

```
void UPR_Enqueue_BK   (ORB *orb) ;  /* blocking enqueue      */
void UPR_Enqueue_NBK  (ORB *orb) ;  /* non-blocking enqueue  */
void UPR_MultipleEnqueue_BK   (ORB *orb,length) ;
void UPR_MultipleEnqueue_NBK  (ORB *orb,length) ;
BOOL UPR_Pkt_Serviced (ORB *orb) ;  /* enquire if serviced   */
void UPR_Pkt_BK       (ORB *orb) ;  /* block if not serviced */
```

Figure 6: The packet departure library.

The `ORB` can be enqueued by either blocking or non-blocking calls. The multi-packet operations allows the output of multiple contiguous data segments to me optimised inside the messageing kernel as the `ORB` can be re-used without intervention from the requesting thread. The status of a non-blocking call can be determined using the predicate `UPR_Pkt_Serviced`. It can be converted into a blocking one by the function `UPR_Pkt_BK`.

### 3.3.3 Packet Arrival

The arrival of data at a particular port may require quite complex activities to realise the desired protocol. State associated with a particular protocol can be installed off the *state* pointer in the *port* structure (see Figure 7), while the semantics of the protocol are introduced by binding action functions to the port which are called between the arrival of the header and the data, the `PreAction`, and after the data has been read, the `PostAction` (see Figure 8).

The pre-action is called if either the pre-action flag is set or if there is insufficient buffer space to hold the packet. The post-action is called if the post-action flag is set or if the port buffer was *exactly* filled by the data body; overfilling cannot occur as the pre-action code must provide more buffering if insufficient space was available.

Pseudo-code for the steps taken by the routing kernel on packet arrival are shown in Figure 9.

```
typedef struct
{
   int   id ;                /* Port identification number          */
   void  (*pre_action)() ;  /* Action called before data body read */
   void  (*post_action)() ; /* Action called after  data body read */
   BOOL  do_pre ;           /* Pre-action flag                     */
   BOOL  do_post ;          /* Post-action flag                    */
   void  *state ;            /* Pointer to state for port           */
   BYTE  *buffer ;           /* Pointer to buffer                   */
   int   space ;             /* Length of space remaining in buffer */
} PORT ;
```

Figure 7: The port data structure.

```
void PreAction (PORT *port, int length) ;
void PostAction (PORT *port, int length, BYTE *buffer) ;
```

Figure 8: Pre-action and post-action procedural interfaces.

#### 3.3.4 A Sample Protocol

The code associated with an acknowledge-transmit protocol in which each end knows the length of the communication is now developed to illustrate the usability of the Interface's primitives. The first step is to consider the actions associated with user input and user output.

The user input process merely records the buffer address and message length in the port and then sends an acknowledgement to the port associated with the channel output and deschedules. If the user output reaches the communication before the acknowledgement then it must deschedule until the acknowledge arrives. Once an acknowledge has been received the message can be transmitted as a series of blocking ORB enqueues. This suggests use of the following structures:

With the communication primitives detailed in Figure 11.

The input port need take no action until the entire message has been received. At this point it reschedules the input process. Thus no pre-action is ever required and the post-action is only required when the buffer space is filled exactly. The output port only receives acknowledgements, it must reschedule the output process if it is not the first of the two to reach the communication point. The post-actions for the input and output ports are given in Figure 12.

All this assumes the data structures have been initialised correctly. This involves directing the packet headers contained in the `ORB` structures at each other (with packet lengths of zero), setting

```
...  read and decode header into (port_id, length)
{
  PORT *port = ports[port_id] ;
  BYTE *buffer ;
  if ((port->space < length) || (port->do_pre))
    port->pre_action(port, length) ;
  buffer = port->buffer ;
  port->buffer += length ;
  port->space -= length ;
  if (length != 0)
    ...  read packet into buffer
  if ((port->space == 0) || (port->do_post))
    post_action(port, length, buffer) ;
}
```

Figure 9: Pseudo-code for the arrival of a packet at its destination

```
typedef struct              typedef struct
{                           {
  BOOL iamfirst;              PORT *port;
  PORT *port;                 process p;
  ORB orb;                    ORB ack;
  process p;                } CHANIN
} CHANOUT;
```

Figure 10: Channel structure declarations

```
void in(CHANIN *chanin, BYTE *buffer, int length)
{
  chanin->port.buffer = buffer;
  chanin->port.space = length;
  UPR_Enqueue_NBK(&chanin->ack);
  deschedule(&chanin->p);
}

void out(CHANOUT *chanout, BYTE *buffer, int length)
{
  if (chanout->iamfirst)
  {
    chanout->iamfirst = FALSE;
    deschedule(&chanout->p)
  }
  chanout->iamfirst = TRUE;
  do
  {
    int size = (length>UPR_MAX_PKT_SIZE) ? UPR_MAX_PKT_SIZE : length;
    chanout->orb.bffr = buffer;
    HdrModLength(size,&chanout->orb.header);
    UPR_Enqueue_BK(&chanin->orb);
    length -= size;
    buffer += size;
  } while (length>0);
}
```

Figure 11: User input/output functions

the `iamfirst` flag, and initialising the port structures by clearing the `do_pre` and `do_post` flags, plugging the channel structures into the `state` pointers and binding in the appropriate post actions.

The sample protocol serves to illustrate how succinctly communication protocols can be defined. Clearly more complex and multifarious protocols may be constructed from the Interface's primitives. Their development may require a more carefully reasoned and methodical approach to guarantee their correctness.

### 3.4 Inmos Process Library

The Inmos process library is typically used to create and join parallel processes. Furthermore, its local channel, semaphore and timer functions are often of use to parallel environments.

## 4 Using ParaPET: A worked example

This section describes the development of a simple programming environment. It is loosely based upon a subset of the communication primitives provided by the Reactive Kernel[4].

```
void InputPostAction(PORT *port, int length)
{
  CHANIN *chanin = (CHANIN*)port->state;
  reschedule(&chanin->p);
}

void OutputPostAction(PORT *port, int length)
{
  CHANOUT *chanout = (CHANOUT*)prot->state;
  if (chanout->iamfirst)
    chanout->iamfirst = FALSE ;
  else
    reschedule(&chanout->p);
}
```

Figure 12: Input and Output Port PostActions

## 4.1 The Target Programming Environment

This illustrative programming environment is a SPMD system in which processes communicate through an asynchronous messaging protocol. A single user process runs on each processor in the network. A message send proceeds by `xmalloc`ing a communication buffer, building a message into the buffer and then `xsend`ing the message to a specified processor (labelled from 0 to one less than the number of processors in the network). The `xsend` blocks until the entire message has left the source processor, it then frees the `xmalloc`ed buffer. All messages arriving at a destination are placed in a queue, the top message in the queue is extracted by calling `xrecvb`. If no messages are present in the queue then the call blocks until a complete message has been read in. No form of selection over the messages in the queue may be performed, neither can their source be identified unless this information is introduced explicitly into the message. Once an incoming message has been processed the buffer can be returned to the system pool by calling `xfree`.

The entry point to the user code is as follows.

```
int usermain(int my_id, int num_procs, int argc, char **argv)
```

Note that the `argc` and `argv` parameters provide access solely to the command-line arguments not associated with lower levels of the system, and are valid on all processors in the network. In addition, every processor also has access to the standard C libraries including those for file and screen I/O.

The communication routines implemented are summarised in Figure 13.

```
char *xmalloc(int length)
void  xfree(char *b)
int   xlength(char *b)
void  xsend(char *buffer, int destn_id)
char *xrecvb()
```

Figure 13: The Communication Primitives

## 4.2 Its Implementation

An implementation of this programming environment is supplied in the `zoo` archive `rk.zoo` which is distributed with releases of VCR Version 2.0h and higher.

The code for implementing the functions described in the previous Section can be found in `rk/source/rk.c`. The `main` in this file is called on every processor in the network, it proceeds by tt mallocing memory for the `PORT` structures associated with message source and for an array which records the base ids of these port blocks on every processor in the network. All ports are initialised, plugging an `MRPS` structure into each one for maintaining the protocol dependent components of the port's state. The `port_ids` array is filled in by performing swaps using the client/server facilities of the VPI.

The user code is then called. Once the user code returns, the `malloc`ed memory can be freed. However, as the return of the `main` on processor 0 terminates the `iserver`, potentially interrupting transactions with other processors, each processor must synchronise with it before its completion.

Three additional files are required. A header file, `rk/libs/rk.h`, which exposes the messaging functions to the user code, a make file, `rk/build/makefile` which generates the communication library `rk/libs/rk.lib`, and a linker indirect file, `rk/libs/rk.lnk`, which identifies the standard C and ParaPET libraries that must be included in the linked user code.

Finally, to expose the communication library to the user, the full path-name of the `rk` directory must be appended to the `$ISEARCH` path (making sure to include a trailing slash). Typing `make` in the `build` directory will then generate the `rk` library in the `libs` directory. Since DOS on PCs has no standard make, a public domain make utility `pdmake` is supplied with VCR.

## 4.3 Compiling and Executing User Code

Two examples of the communication libraries use are given in sub-directories off `rk/examples`. This Section describes the compilation and execution of the parallel `pi` computation.

This example is drawn from the *IRIX: Parallel Programming Overview* by E Manolis. It is a simple calculation of $\pi$ using the rectangle rule. It approximates

$$\pi = 4\int_0^1 \frac{dx}{1+x^2}$$

using the formula

$$\pi \approx \frac{4}{N}\sum_{i=0}^{N-1} \frac{1}{1+\left(\frac{(i+\frac{1}{2})}{N}\right)^2}$$

The total number of terms in the sum, $N$, is represented by the variable `num_rects` in the program: these terms are distributed over the `num_procs` processes. Each process accumulates its share of the terms into a local partial sum. The partial sums are then accumulated.

An appropriate make file is produced by creating an application linker indirect file, `pi.lnk` containing:

```
pi.t8h
#include rk.lnk
```

and then using the `vmakef` tool:

```
vmakef pi.c8h -c
```

specifying the target filename for the fully linked unit and supplying the `-c` option which enables the building of C programs. Executing:

```
make -f pi.mak
```

will then generate a user application binary that can be executed using the following command line:

```
parapet <topology> [-kf] pi.c8h <num_rects>
```

where `<topology>` identifies the network configuration file (ncf) generated using the standard VCR tools, `-kf` switches the routing kernel into fast mode, and `<num_rects>` is the number of rectangles into which the integration has been broken.

# References


[1] Debbage, M., Hill, M. B., and Nicole, D. A. Virtual Channel Router version 2.0 user guide. ESPRIT Project PUMA P2701 working paper no. 25, University Of Southampton, June 1991.

[2] Debbage, M., Hill, M. B., and Nicole, D. A. An interface to a reliable packet delivery service for parallel systems. In preparation, University of Southampton, 1992.

[3] Inmos Ltd. *C Toolset Manual (D7214)*. Bristol, United Kingdom, 1990.

[4] Seizovic, J. The reactive kernel. Cs-tr-88-10, Caltech Computer Science Rechnical Report, 1988.