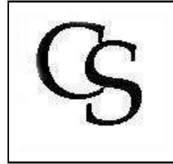


Computer Science
Technical Report

MuPC: A Run Time System For
Unified Parallel C

by

Jeevan Savant

Computer Science Technical Report
CS-TR-02-03

September 2002

MichiganTech

Houghton, MI 49931-1295

Abstract

Unified Parallel C is an extension to the C programming language intended for parallel programming. UPC adds a small number of parallel programming constructs to C. These constructs enable users to write parallel programs based on a distributed shared memory model. MuPC (MTU's UPC), is a run time system for UPC, enabling it to run on a wide variety of architectures including networks of workstations, Beowulf clusters and shared memory machines.

Contents

1 Introduction **3**

2 Background **7**

2.1 Parallel programming languages 7

2.2 UPC's predecessors 8

2.2.1 Split-C 10

2.2.2 AC 11

2.3 UPC 11

2.3.1 The UPC language 12

2.3.2 The frontend 15

2.3.3 The run time system 16

2.3.4 One-sided and Two-sided communication 17

3 MuPC Design **18**

3.1 Using MuPC 18

3.2 The MuPC system model 19

3.2.1 Communication thread 19

4 MuPC Components **24**

4.1 Initialization 24

4.2 Gets and Puts 24

4.2.1 `Get` operation 24

4.2.2 `Put` operation 25

4.3 Synchronization 25

4.3.1 Barrier operation 27

4.3.2 Notify/Wait operation 28

4.4 Memory Management and Locks 29

4.4.1 Dynamic Memory 30

4.4.2 Locks 33

4.5 Termination 39

4.6 MuPC Restrictions 39

4.6.1 Strict and Relaxed 39

4.6.2 **upc_global_exit** 40

4.6.3 Final Barrier Processing 40

5 Summary **41**

5.1 Current state of MuPC 41

5.2 MuPC Testing 41

5.3 MuPC Porting 41

5.4 Release Information 42

1 Introduction

Parallel computing works on the principle of dividing a big task into a number of smaller subtasks, and then executing these subtasks in parallel on multiple processing elements. This is done in order to reduce execution time. Parallel computing platforms can be classified in a variety of ways. One way to classify them is on the basis of how different processing elements interact with each other. Two common ways processors interact are by message passing and by using a shared address space. In a message passing architecture, the processing elements interact with each other by sending messages over an interconnection network (Figure 1). In a shared memory architecture, processing elements interact with each other by sharing a single, common memory (Figure 2).

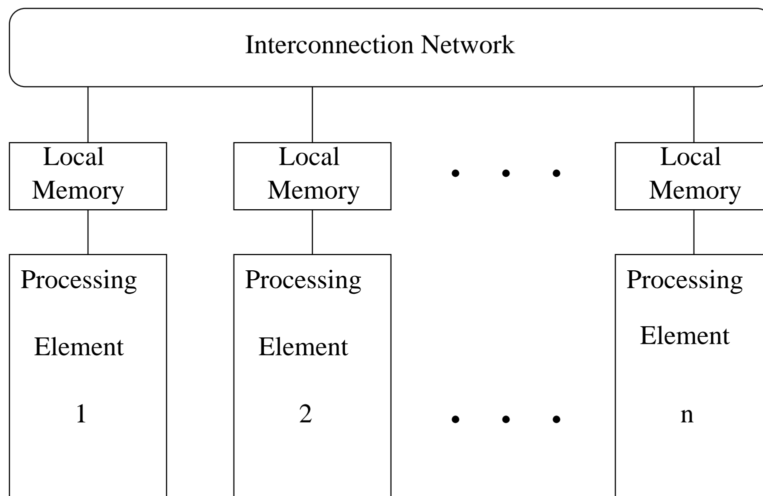


Figure 1: Message passing architecture.

A variety of software mechanisms support these architectures. The Message Passing Interface (MPI)[1] is a library of functions used to do parallel computing on message passing platforms. MPI is widely used with C and FORTRAN programming languages on a variety of message passing platforms such as networks of workstations and Beowulf clusters. MPI can also be used on shared memory platforms. OpenMP is a collection of compiler directives used to do parallel computing on shared memory architectures. The OpenMP compiler directives are added to a C program to exploit parallelism on shared memory platforms.

Both the parallel computing platforms and mechanisms used to program them have

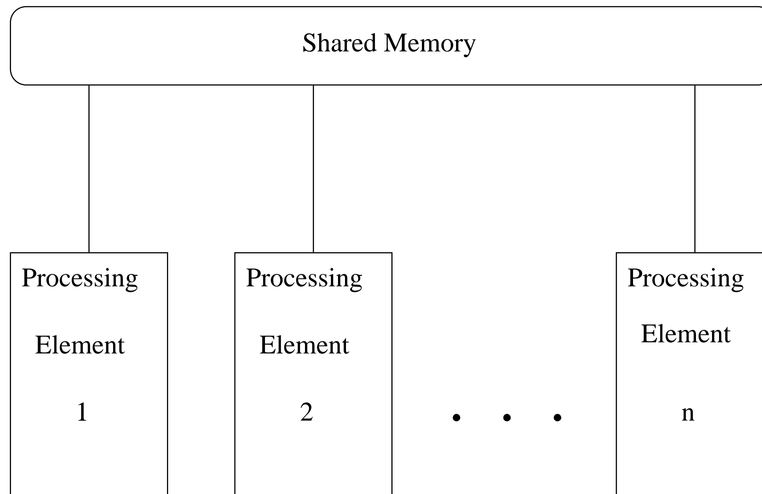


Figure 2: Shared memory architecture.

some advantages and disadvantages. One main advantage of the message passing architecture is its low cost compared to expensive custom built shared memory machines. On the other hand the programming interface for a message passing architecture requires the user to concentrate on interprocessor communication thus complicating the problem solving task. The shared memory programming interface provides a global address space view of the system and thus frees the user from explicit message passing.

The distributed shared memory (DSM) paradigm of parallel programming provides the advantages of shared memory programming on message passing platforms. DSM provides a global memory view of remote memory. This abstraction hides references to remote memory and helps the user to concentrate on the actual problem to be solved (Figure 3).

Unified Parallel C[2] is an extension to the C programming language for parallel programming. Although UPC is based on the shared memory model for parallel computing, UPC offers the flexibility of being able to run on distributed shared memory machines. This is done by defining a clear run time system interface. The run time system is the interface between UPC and the hardware platform. The shared memory constructs in UPC are translated to function calls defined by the run time system interface. The job of the run time system is to implement the shared memory constructs of UPC in a DSM environment thus making UPC portable over large

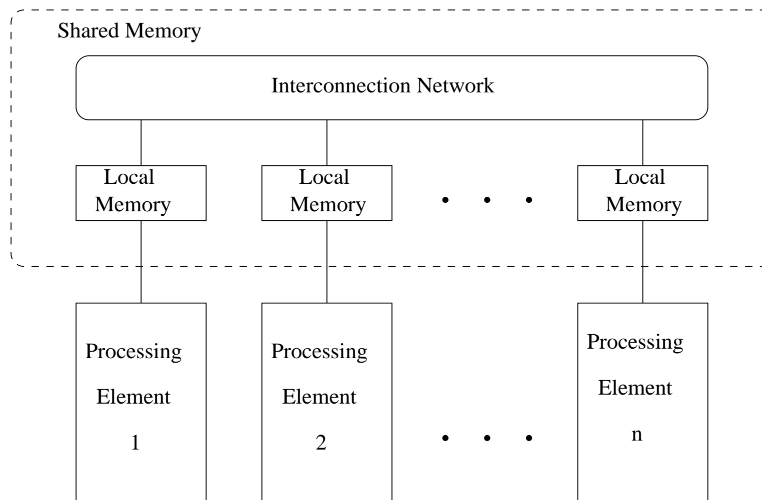


Figure 3: Distributed shared memory architecture.

number of platforms.

The important aspects of parallel programming are how the task is divided, how data is distributed and how the execution is synchronized. In UPC the task is divided among a group of threads. A UPC thread is often equivalent to a processor. UPC does not deal with the details of actual process management. UPC assumes that other software provides process management.

UPC addresses the issue of data division by providing built-in language constructs. These language constructs enable data to be distributed among a set of threads at compile time. This frees the user from doing explicit message passing to distribute data at run time. UPC introduces a new type qualifier `shared` in order to distribute data. A data object qualified as `shared` is shared among all threads. The data object can be a scalar or a vector. In the case of a scalar, the actual data object resides in the memory of thread 0. In the case of a vector the elements are distributed in round robin fashion among the threads. UPC also provides an optional `block size` parameter which specifies data layout in the case of vectors. A vector qualified with a `block size` of B is distributed among threads so that B elements are on thread 0, B elements are on thread 1 and so on.

UPC also provides built-in constructs for synchronization. These constructs include atomic synchronization such as a barrier and a split barrier consisting of `notify` and

`wait`. Additionally, UPC has library functions for shared memory allocation and deallocation.

Compaq's implementation of UPC defines a clear interface between UPC compiled code and the run time system. This interface is the starting point for the design of MuPC. The research done here focuses on the implementation of a run time system for UPC on distributed memory platforms. This implementation uses MPI and Pthreads libraries.

2 Background

2.1 Parallel programming languages

A large number of languages is available for parallel programming. These languages are the result of research efforts both by commercial vendors and academic researchers. The reason for the existence of so many languages is the wide variety of parallel computing architectures and parallel programming paradigms. Each parallel programming language is designed with the goal of being most efficient on a particular parallel computing architecture using a particular programming paradigm. In this section we briefly discuss various parallel programming paradigms and languages designed to support them[3].

- Implicit and explicit parallel programming[4]: In the implicit parallel programming paradigm programs are developed using a sequential programming language and the task of making it parallel is left to the compiler. No information is passed to the compiler explicitly about how the program should be made parallel. The job of the parallelizing compiler is to divide the program into blocks and do dependency analysis among those blocks to identify which blocks can be parallelized. Such a task is not always easy and efficient and depends on how the sequential program is coded. Haskell[5] is a parallel programming language that supports implicit parallel programming.

  In the explicit parallel programming paradigm the programmer explicitly specifies how the program is to be parallelized among a group of processors. An explicit programming language has mechanisms to achieve parallelism built into the language. Such mechanisms include constructs for message passing, synchronization and process management. SR[4] is an example of an explicit parallel programming language.

  Implicit parallel programming languages are easy to use but suffer from low efficiency. Explicit parallel programming languages enable programmers to optimize the program taking the architecture into consideration but they also increase the complexity in programming.

- Shared memory and message passing parallel programming: The shared memory parallel programming paradigm is based on shared address space architectures. In such architectures a collection of processors has access to a common pool of memory which can be accessed and updated by all. OpenMP is a set of compiler directives that can be used to specify shared memory parallelism in FORTRAN and C/C++ programs. Since the shared memory model of parallel

programming saves the effort of message passing, languages often simulate the shared memory model on distributed memory architectures. An example of such language is AC[6]. UPC is a descendant of AC.

In the message passing parallel programming paradigm individual processors have local memory. Data exchange takes place through explicit messages. Message passing programming can be done both on shared and distributed memory machines. ANSI C with MPI is an example of a message passing programming language.

- Data parallelism and control parallelism: Data parallelism is achieved by assigning data elements to different processors executing the same code. A single instruction stream then executes on all processors. Data parallel languages provide constructs for data distribution and parallel operations. There is no explicit message passing involved and all communication needed to achieve a parallel operation is built-in. C*[7] is an example of a data parallel language.

  Control parallelism involves execution of multiple instruction streams on different processors. The instruction streams can work on the same data or on different data. An example of control parallelism is pipelining.

2.2 UPC's predecessors

UPC belongs to the class of parallel languages based on the distributed shared memory model. The programmer is given the view of working on a shared memory architecture. Internally the compiler and run time system take care of simulating a shared memory model irrespective of the underlying platform. In this section we discuss the predecessors of UPC, *i.e.*, AC and Split C. Both of these languages are extensions to the ANSI C language[8] to support the distributed shared memory programming model. Languages in this genre can be compared on the basis of several common characteristics[9].

- Control model: The control model defines how control flows during execution of a program. In the case of a sequential program the control flow starts at `main()` and follows sequentially to the end of the program. In the case of a parallel program where multiple processing elements are available for execution, there can be a variety of control flows. One common approach is the master-worker model where a single master controls the flow among a set of workers. The control flow model used for languages discussed in this section is SPMD (Single Program Multiple Data). In this model all processing elements execute the same code but with different data sets.

- Global shared address space: Irrespective of the underlying architecture, these languages assume the existence of a common pool of memory shared among all processing elements involved in the computation. Mechanisms and constructs are provided to access and update memory. The address space can store both static and dynamic structures. Also important to note is that each processor owns a part of the global address space and can access it as if it was local. The actual realization of the global address space is left to the underlying run time system. The global address space has the following characteristics.
  - Pointers: Pointers in these languages are of two types. Local pointers that point to objects in the local address space and global pointers for objects in the global address space. These can be further classified as:
    * Local pointers pointing to the local address space.
    * Local pointers pointing to the global address space.
    * Global pointers pointing to the local address space.
    * Global pointers pointing to the global address space.
  - Shared scalar variables: Languages supporting a global address space provide constructs to qualify a scalar to be in the global address space.
  - Shared vectors/arrays: Additional syntactic features are provided to support vectors in the global address space. The general approach is to allocate an equal number of elements on all processing elements but the user is also given the option of trying a variety of ways in array declaration to achieve optimal allocation strategy.
  - Shared dynamic memory allocation: The general approach to do this is each processing element allocates an equal-sized block of memory which together become the global address space. Such memory can then be operated on using global pointers.
- Synchronization: Synchronization mechanisms are needed to avoid race conditions and deadlocks when multiple processing elements access the global memory. These mechanisms are either provided as built-in constructs or as library functions. Commonly provided synchronization mechanisms are barriers and locks.
- Split-phase assignment: An assignment operation involving the global address space has two phases, the computation phase and the communication phase. The communication phase has higher latency than the computation phase. If such assignments are implemented sequentially there is loss of efficiency since the processor remains idle while the remote access is taking place. The two

phases can be overlapped by providing constructs which initiate communication and constructs that wait for its completion. Between the two constructs the processor can do other work thus hiding the latency of remote memory accesses.

2.2.1 Split-C

Split-C is a parallel extension to C developed at UC Berkeley[9]. It provides useful elements of shared memory, message passing and data parallel programming models in the context of the C language. Split-C is the oldest of the three languages discussed in this section. It is targeted for the Thinking Machines Corporation CM-5.

- Control model: The control model for Split-C is SPMD. All the processing elements execute the same code. Each element may work on the same or different data-sets. Split-C provides variables such as `PROCS` and `MYPROC` to find the total number of processing elements present and the individual process identification number.
- Global address space: Each processor has access to the entire global space and each process owns a part of it, which it can access as local.
  - Pointers: Global pointers are declared with the keyword `global`. Pointers can be of any type except the function type. Additionally, Split-C has a type of pointer called as spread pointers declared with the `spread` keyword. The difference between the global and the spread pointer is that a spread pointer points to a collection of data items on a processor and an increment makes it point to the collection of data items in next processor.
  - Arrays: Split-C calls global arrays as spread arrays. They are declared by inserting a single spreader to right of the array dimensions. For example, `X[n]::[m];` where `::` is the spreader. The dimensions on the left side of a spreader are spread across all processors. The dimensions on right side define per processor sub-arrays.
  - Dynamic memory allocation: Dynamic memory allocation and deallocation is done using library functions `void *spread_all_spread_malloc(int count, int object_name);` and `all_spread_free(void *spread sptr);`
- Synchronization: Split-C provides primitives such as `atomic` and `barrier`. The `atomic` primitive is used to qualify a function which is then executed atomically on the processor which owns the global object. The particular global object is the object on which the function acts. Split-C also provides the common barrier

synchronization primitive `barrier`. Using these primitives users can build a variety of other synchronization operations.

- Split-phase assignment: This is the most prominent feature of Split-C from which the name for the language is derived. The overlap in computation and communication phases of an assignment operation in global space is done by splitting the an assignment into two operations. A new symbol for assignment is introduced `:=`. The `:=` marks the beginning of communication, while a `sync` operation is used to determine completion of communication.

2.2.2 AC

AC[6] is an extension to C, that supports the shared address space parallel programming model on distributed shared memory architectures. It is targeted for Cray Research's T3D and T3E.

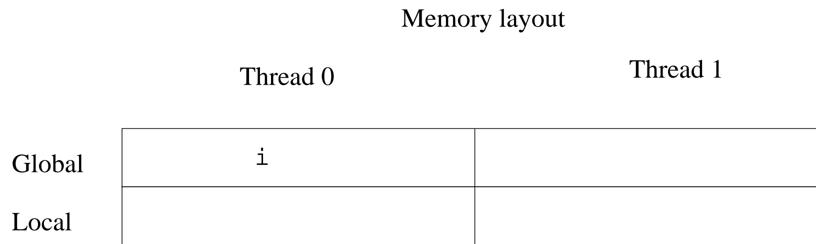
- Control model: The control model for AC is SPMD. AC provides variables such as `PROCS` and `MYPROC` to find the total number of processing elements present and individual process identification numbers.
- Global address space: The concept of global address space remains consistent with the discussion in previous sections.
  - Pointers: Global pointers are declared with the keyword `dist`. Internally a pointer has two components, the processor number and the local address on that processor.
  - Arrays: Distributed arrays are declared in which one dimension of the array is a multiple of `PROCS`.
- Synchronization: AC provides the `_barrier` construct for synchronization.

2.3 UPC

UPC[2] is directly derived from AC. It is designed to do parallel programming in C using the shared memory model. UPC gives a view of the underlying machine model as one having a collection of threads that share a global address space. Threads in UPC terminology are usually synonymous with processes.

2.3.1 The UPC language

- Control model: The control model for UPC is SPMD. The user sees the UPC program to be a collection of threads that share a common global address space. Additionally, each thread has its own private space and part of the global address space which it can access as local.
- Global address space:
  - Shared scalar: UPC introduces a new keyword **`shared`** to qualify data objects in the global address space. A shared scalar is maintained as a single object in global space. In Figure 4, the variable `i` is a shared scalar. It is allocated in thread 0's local memory, but all threads can access it as local.

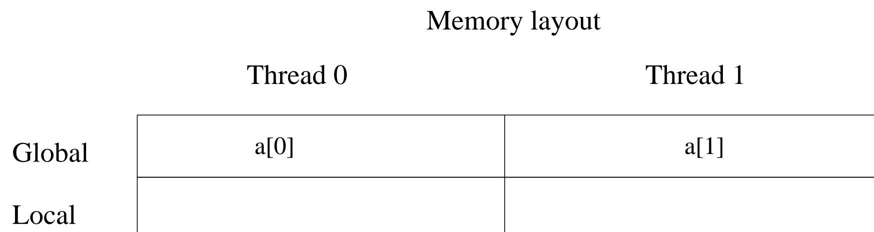



```
shared int i;
void main(void) {
}
```
Thread 0

```
shared int i;
void main(void) {
}
```
Thread 1

Figure 4: A shared scalar in UPC.

  - Shared array: Shared arrays are vectors in global space. UPC requires at-least one dimension of the array to be `THREADS`. Each thread holds an equal share of the array in its local address space. Thus in the case of a 1-dimensional shared array with `THREADS` elements each thread has one element owned locally (Figure 5). This property is defined as *affinity*. Affinity determines in which thread's local memory a particular shared item resides.



One element per thread

```
shared int a[THREADS];
void main(void) {
}
```

Thread 0

```
shared int a[THREADS];
void main(void) {
}
```

Thread 1

Figure 5: Shared array with single dimension and blocksize 1.

UPC also has an optional blocksize parameter which is used in array declarations. to control the distribution of data elements. The default blocksize is 1. As shown in Figure 6 a multidimensional array with blocksize of 1 and 2 threads has an equal number of elements stored on both threads in alternate order.

Figure 7 shows a multidimensional array with blocksize of 2 and 2 threads. In this case both threads have a equal number of elements but the ordering is changed.

– Shared pointer: UPC supports the following four types of pointers.

```
int *ptr1; /* local pointer pointing to local data */
shared int *ptr2; /* local pointer pointing to global data */
int* shared ptr3; /* shared pointer pointing to local data */
shared int *shared ptr4;/* shared pointer pointing to global data
*/
```

A shared pointer is internally represented by a thread number and the local address it points to. Pointer arithmetic supports blocked and unblocked array distribution. Additionally, shared pointers can be cast into local pointers but not vice versa.

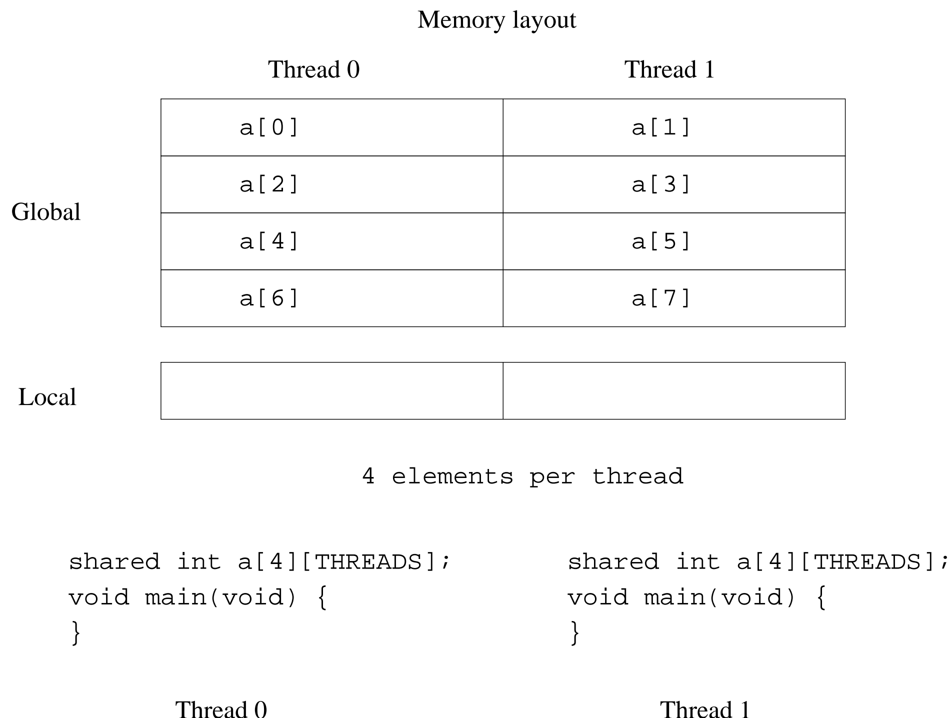


Figure 6: Shared array with multiple dimension and blocksize 1.

– Dynamic memory allocation:
  UPC offers functions to allocate global memory dynamically.

```
shared void *upc_global_alloc(size_t nblocks, size_t nbytes);
shared void *upc_all_alloc(size_t nblocks, size_t nbytes);
shared void *upc_local_alloc(size_t nblocks, size_t nbytes);
void free(shared void *ptr);

/* nblocks:  number of blocks */
/* nbytes:  number of bytes per block */
```

  This can be done collectively by all threads or by single thread. `upc_global_alloc` is a non-collective function allocating shared memory. `upc_all_alloc` is a collective function allocating shared memory. `upc_local_alloc` is a non-collective function which allocates shared memory having affinity to the calling thread. `upc_free` deallocates dynamically allocated shared mem-

Memory layout

	Thread 0	Thread 1
Global	a[0]	a[2]
	a[1]	a[3]
	a[4]	a[6]
	a[5]	a[7]
Local		

4 elements per thread and blocksize 2

```
shared [2] int a[4][THREADS];
void main(void) {
}
```

Thread 0

```
shared [2] int a[4][THREADS];
void main(void) {
}
```

Thread 1

Figure 7: Shared array with multiple dimension and blocksize 2.

ory.

- Synchronization:

  UPC offers blocking and non-blocking types of synchronization mechanisms.

```
upc_barrier(); /* Blocking synchronization */
upc_notify(); /* Split barrier start */
upc_wait(); /* Split barrier end */
```

  The advantage of non-blocking synchronization is that the user can do local work before waiting for the synchronization to complete.

2.3.2 The frontend

The first step in generating a UPC executable is to translate the UPC code to ANSI C code. This translation is done by the *frontend*. The frontend is a source-to-source translator that translates UPC constructs to calls to the run time system. For example, each read or write access to non-local memory is translated to a call to the `get` or `put` functions respectively. Figure 8 shows an example of UPC code in

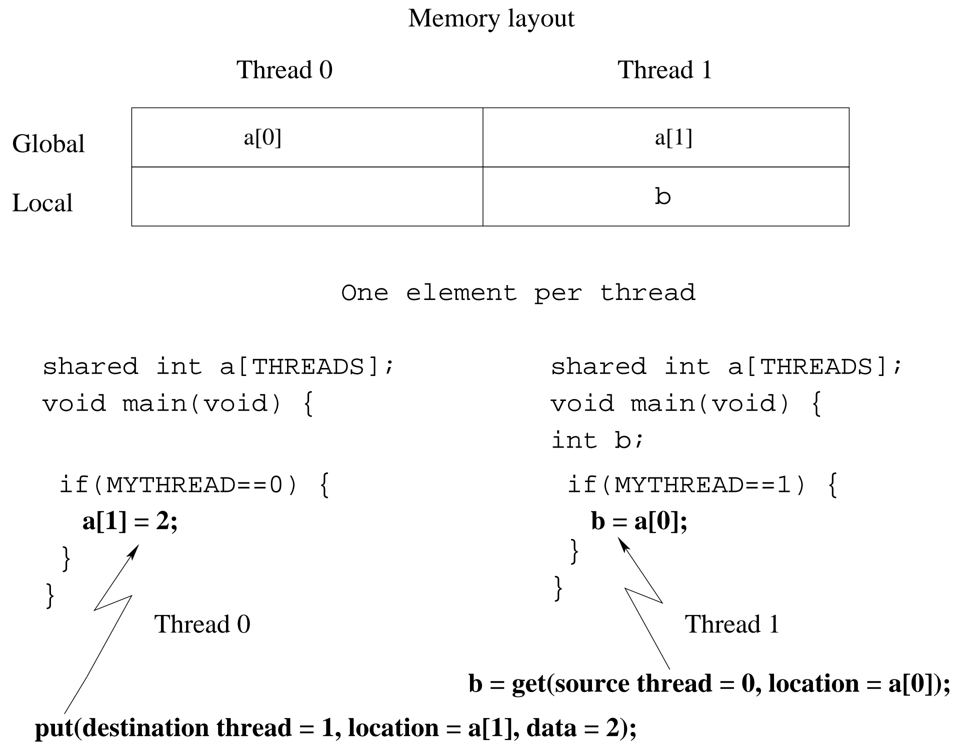


Figure 8: UPC to C translation example.

which references to non-local memory are translated to `get` and `put` function calls. Compaq is providing us their UPC compiler. This compiler makes use of Edison Design Group's frontend to do the UPC to C translation.

2.3.3 The run time system

It is important to note that the UPC to C translation shown in Figure 8 or in other examples elsewhere in this document is based on Compaq's UPC compiler. UPC *per se* does not dictate how the lower layers should implement UPC constructs.

The `get` and `put` functions in the translated code are one-sided communication operations. The one-sided method of communication and its counterpart are explained subsequently.

2.3.4 One-sided and Two-sided communication

A group of processes executing a parallel program need to exchange data and control information. The code in Figure 8 is an example of data being exchanged between two cooperating UPC threads. There are two basic schemes to exchange information between a sender and a receiver.

In two-sided communication, both the sender and receiver are actively involved in the exchange of messages. An example of this are the MPI functions `MPI_Send` and `MPI_Recv`. A `MPI_Recv` does the job of polling on the receiver side while the `MPI_Send` is used to send data by the sender. The disadvantage of this scheme is the time lost in waiting at the receiver.

In one-sided communication, only the sender or the receiver is actively involved in message exchange. This scheme is efficient when the receiver does not know beforehand which sender to listen to.

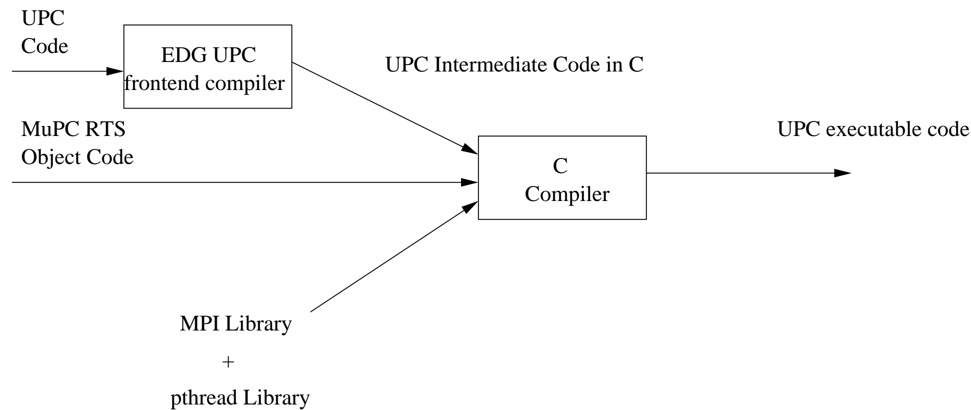


Figure 9: The big picture: mupcc.

3 MuPC Design

The goal of MuPC is to enable the execution of UPC programs on any platform that supports MPI and Pthreads. Some examples of the target platforms include Beowulf clusters, networks of workstations and shared memory machines such as the SUN Enterprise.

3.1 Using MuPC

MuPC provides `mupcc` and `mupcrun` commands and the MuPC library (Figure 9).

The user requires the following software and hardware components to use MuPC.

- A UPC compiler capable of doing UPC to C translation conforming to Compaq's interface.
- The MPI and Pthread libraries.
- The MuPC library, and the `mupcc` and `mupcrun` scripts.
- Any hardware platform with the above software components.

The flow of events in generating a UPC executable is as follows (Figure 10).

- The front end translates UPC code into equivalent intermediate C code with calls to the RTS.

```
mupcc filename.c libmupc.a

where filename.c is the UPC code file
and libmupc.a is the run time system library.


The resulting executable a.out can be executed as

mupcrun -n n a.out

where n is the number of processors
```

Figure 10: Using MuPC.

- The intermediate C code is compiled and linked with the MuPC run time system library.
- The resulting output is the UPC executable which can be executed using the `mupcrun` command.

3.2 The MuPC system model

Each UPC thread in the user program is implemented as a Unix process in MuPC. Each MuPC process is a MPI process which spawns two Pthreads. The two Pthreads are called the *communication* thread and the *UPC* thread. The communication thread is the Pthread that handles the message passing. The *UPC* thread runs the user's UPC code. These two threads interact with each other using the *global* memory. Its important to note the difference between UPC's concept of "shared" memory and the above mentioned "global" memory. The global memory is the per process memory which the two threads in a process can access. All the MPI activity is confined to the communication thread to achieve thread safety. Figure 11 shows the system model described above.

3.2.1 Communication thread

The communication thread is implemented as an infinite loop servicing the outgoing send requests and the incoming receive requests. The communication between the UPC thread and the communication thread is achieved through global memory (Figure 12). Two structures, one for the send requests and the other for the receive requests are shared by the two threads.

The communication thread starts by posting non-blocking requests to receive data from all the processors involved in the execution of the program. Subsequently it

```
mupcrun -n 2 upc_hello_world
```

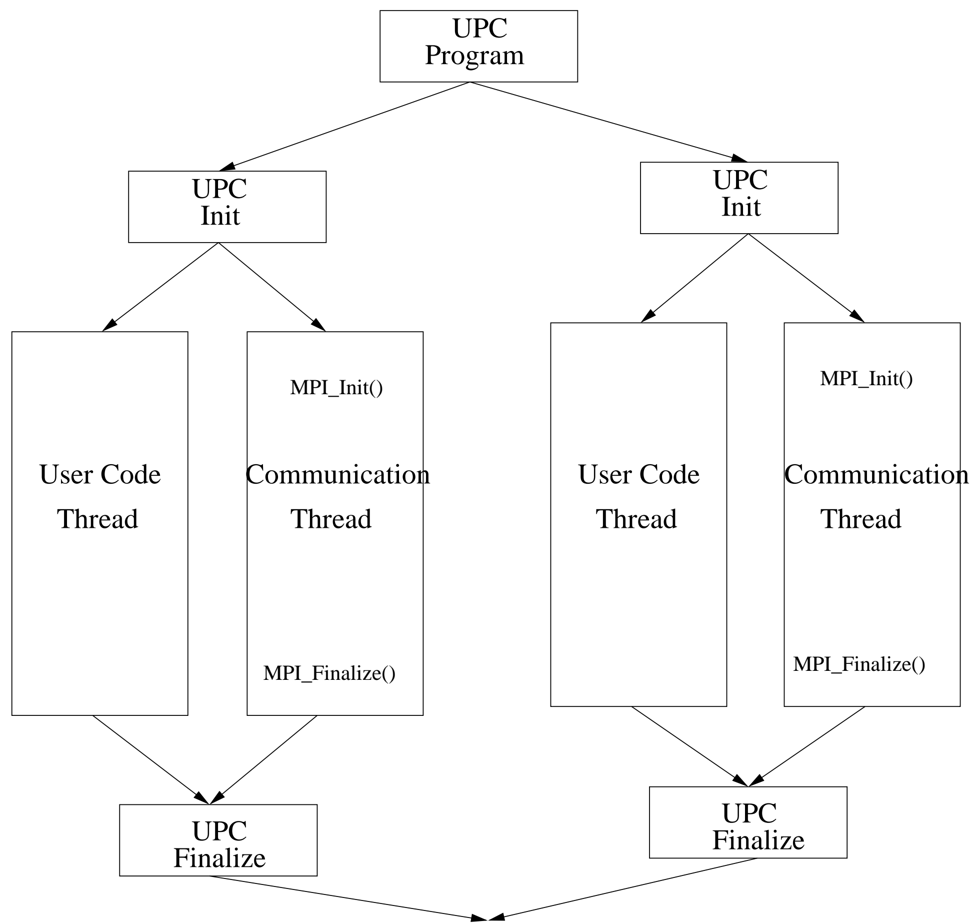


Figure 11: MuPC system model.

enters an infinite loop where it services the incoming and outgoing communication requests. On receiving an incoming request, the communication thread processes it and at the end of the processing posts a new receive request to listen from the processor with which it last communicated. Also, in order to avoid the waste of CPU cycles by looping needlessly in the case of no communication activity, the communication thread yields voluntarily, giving more CPU cycles to the UPC thread. The communication thread terminates after receiving finish requests from all the other processors involved in the execution.

To issue a put/send request the UPC thread locks the send structure, fills in the request parameters and unlocks it for the communication thread to read. The communication thread periodically locks the send structure to check for any outstanding send requests from the UPC thread. In the case of a new send request, the communication thread sends the request to the remote processor using a blocking MPI send. There are 3 types of send requests.

- Send remote: This is a request to write to the remote processor's memory. MuPC implements this by sending the data to the remote processor using the blocking MPI send function.
- Recv remote: This is a request to read data from the remote processor's memory. MuPC implements the remote reads using a two-phase protocol.
- Finish: This is a request to terminate execution.

To communicate the incoming receive requests to the UPC thread, the communication thread uses the receive structure. Not all incoming receive requests are needed to be communicated with the UPC thread. Only the data requested by the UPC thread from remote processors needs to be communicated to the UPC thread. There are 4 types of receive requests.

- Data: This is a request to write to the local processor's memory from the remote processors. This is implemented by writing the data at an appropriate location.
- Recv: This is a request to read the local processor's memory and send the contents to the remote processor.
- Recv reply: This is a request indicating reply to the local processor's request to read the remote memory.
- Finish: This is a termination request from the remote processor.

```
keep_polling = n
Post n non−blocking MPI receives
to listen from n processors

While keep_polling true
        Try locking send_struct
        If locked
        Check request                                     // Outgoing Requests
              If Send_Remote                              // Write request TO remote processor
                Send Data request to remote processor using a Blocking send
                Unlock send_struct
              If Recv_Remote                              // Read request to remote processor
                Send Recv request to remote processor using a Blocking send
                Unlock send_struct
              If Finish                                   // Request to terminate communication thread
                Send Finish request to remote processor using a Blocking send
                Unlock send_struct
        If lock failed or no request
        Check if incoming requests from remote processors  // Incoming Requests
              Lock recv_struct
              If Data request                             // Write request FROM remote processor
                Receive data and store
                Unlock recv_struct
              If Recv request                             // Read request FROM remote processor
                Read data and send Recv_Reply request using a Blocking send
                Unlock recv_struct
              If Recv_Reply                               // Receive Data FROM remote processor
                Receive data and store
                Unlock recv_struct
              If Finish
                Decrement keep_polling                    // Process finish request FROM remote processor
                Unlock recv_struct
          Post a new non−blocking MPI receive for the last processor communicated with
Yield communication thread
Done While
```

Figure 12: Communication thread

The actual send and receive operations are explained in detail in the following sections. The prime feature of the communication thread is, it is designed to be completely non-blocking, with no waiting involved for the completion of a communication request at the remote end. This might sound contradictory looking at the use of blocking MPI sends in the communication thread, but the blocking sends in MPI do not wait for an acknowledgment from the remote processors. The non-blocking characteristic of the communication thread is the essence of the MuPC run time system.

4 MuPC Components

4.1 Initialization

The Compaq UPC run time system interface declares
`int _UPCRTS_init(void)` as the initialization function. This is the first function to be called when a UPC executable is invoked. The initialization task does the job of buffer allocation, MPI initialization and spawning of the UPC and the communication thread. It is designed and implemented to build the system model shown in Figure 11.

4.2 Gets and Puts

4.2.1 `Get` operation

An attempt to read data stored in remote memory results in the invocation of the `Get` operation. MuPC implements the `Get` operation using a two-phase protocol (Figure 13). The motivation for this protocol is to maintain the non-blocking nature of the communication thread. Due to this the communication thread does not need to wait for the completion of the `Get` operation, possibly ignoring incoming requests from the other processors. The `Get` operation proceeds as follows.

- Phase 1: Making the request
  - Step 1: The UPC thread builds the receive request and stores it in the global memory. The UPC thread then waits for the "receive done" flag to be set.
  - Step 2: The communication thread occasionally checks the global memory for outstanding UPC requests. If it detects a request, it processes it as described in the next step.
  - Step 3: The communication thread sends the receive request to the remote processor.
- Phase 2: Receiving the reply
  - Step 4: The remote processor's communication thread detects an incoming receive request. It replies with the requested data.

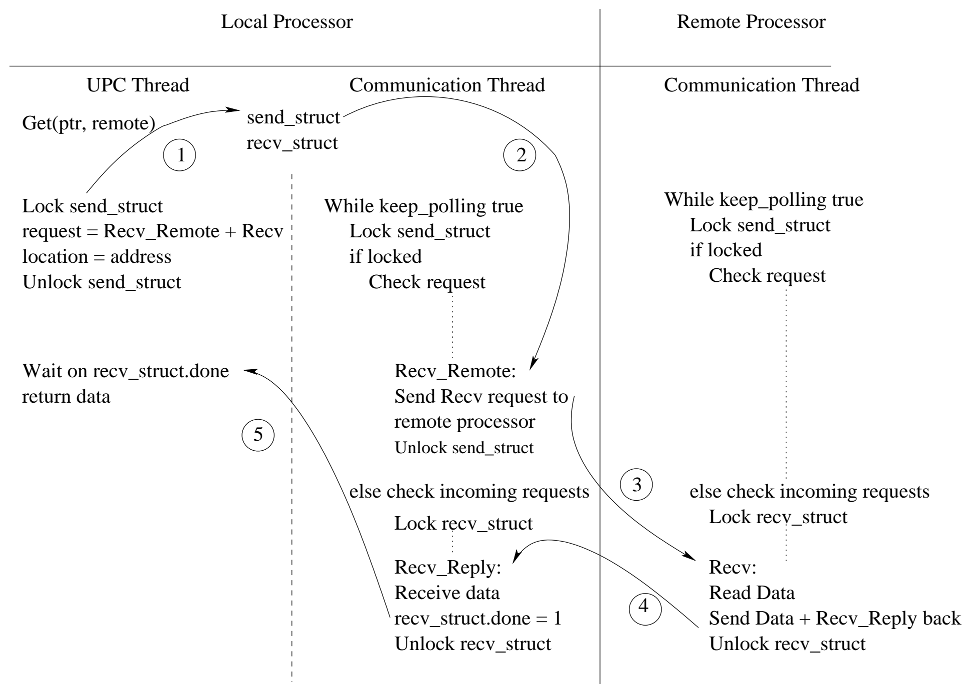


Figure 13: `Get` operation

– Step 5: The local processor's communication thread detects a reply to the receive request and receives the data. It then sets the receive done flag, thus ending the UPC thread's wait.

4.2.2 `Put` operation

The `Put` operation is simpler than the `Get` operation. Writes to remote memory result in the invocation of the `Put` operation (Figure 14).

The UPC thread builds the send request and stores it in global memory. The communication thread detects the request and sends the data to the remote processor. Meanwhile, the UPC thread continues the execution without waiting for the completion of the `Put`.

4.3 Synchronization

UPC provides two types of barrier constructs: a typical barrier and a "split" barrier. `UPC_barrier` is a blocking synchronization construct. A call to the `UPC_barrier`

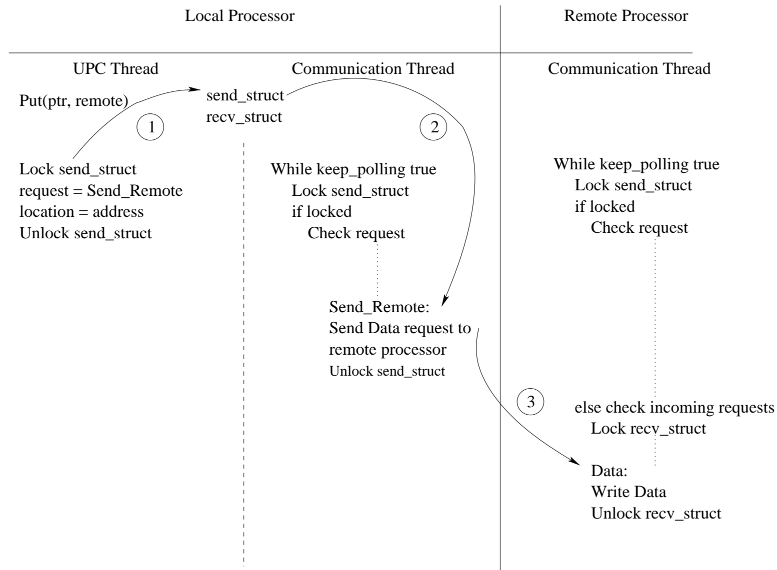


Figure 14: `Put` operation

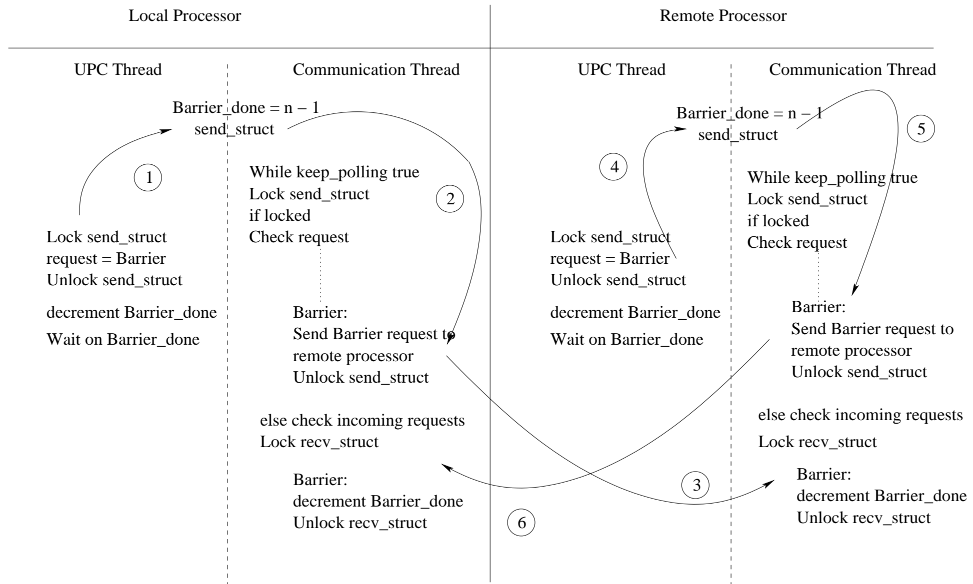


Figure 15: `Barrier` operation

blocks until all the other processors executing the program also call `UPC_barrier`. The "split" barrier is a sequence of two constructs, `UPC_notify` and `UPC_wait`. The Notify construct announces a processor's intent to synchronize with other processors. The Wait construct blocks until all the other processors have called Notify in their corresponding Notify-Wait sequence. The two-step synchronization procedure is designed to improve efficiency by allowing local work to be done between the Notify and the Wait constructs.

4.3.1 Barrier operation

MuPC implements the `UPC_barrier` operation using an all-to-all token exchange protocol (Figure 15). Each processor initializes a variable with a value equal to the total number of UPC threads(processors) - 1. Whenever the UPC code calls `UPC_barrier`, the run time system sends a barrier request to all the processors involved in the computation. On the receipt of a barrier request at the remote processor, the run time system decrements the value of a barrier variable by one. The UPC thread at the local processor waits until the value of the barrier variable is not equal to one *i.e.*, until all the other processors encounter their barriers. Once this is achieved the wait completes and the UPC threads on all the processors can continue their execution.

- Step 1: The UPC thread on the local processor calls the barrier. The barrier function builds the barrier request and stores it memory. The barrier function sends this barrier request to all the processors. The barrier variable is initialized to the number of processors. The UPC thread waits until this barrier variable value is not equal to zero.
- Step 2: The communication thread detects a request from the UPC thread and sends the barrier request to all the remote threads.
- Step 3: The remote processor's communication thread detects an incoming barrier request and decrements its barrier variable by one.
- Step 4: The remote processor's UPC thread hits its own barrier function call. It repeats the same procedure in step 1.
- Step 5: Same as step 2.
- Step 6: Same as step 3.

As a result, by the time the barrier variables on all the processors are zero, the user code on all the processors have hit their barriers, thus achieving synchronization.

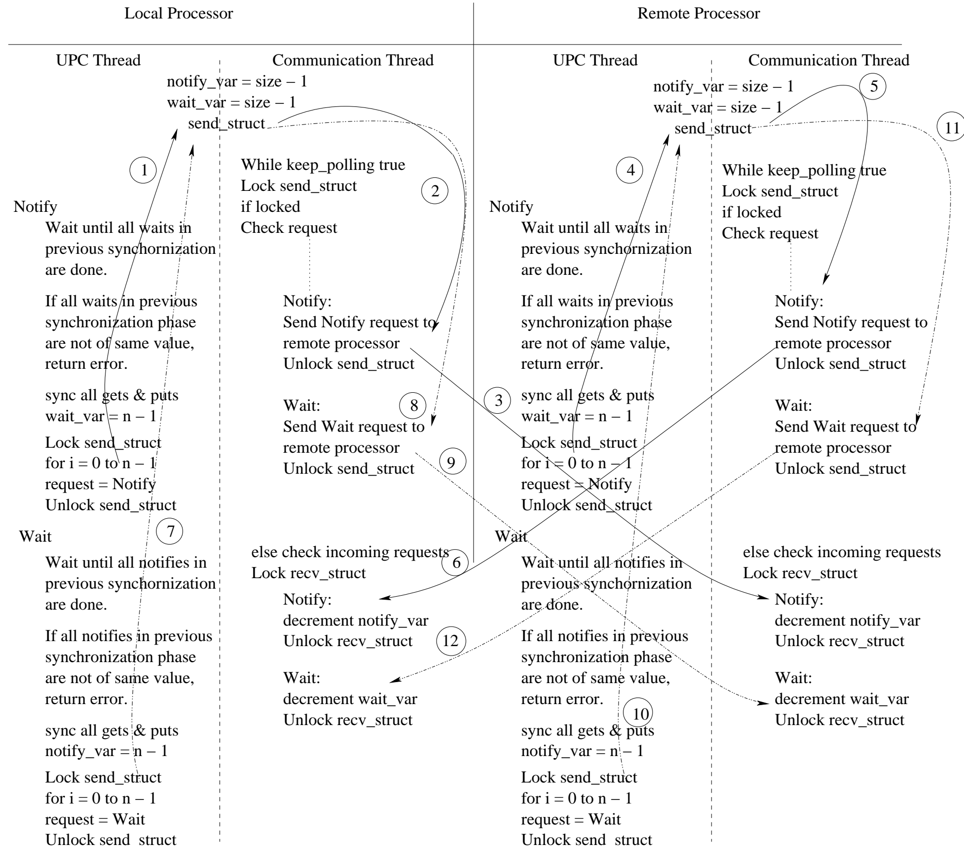


Figure 16: Notify operation

4.3.2 Notify/Wait operation

MuPC's algorithm for implementing the Notify/Wait operation is shown in Figure 16.

- Step 1: The upc_notify routine is called with value n. Calling upc_notify marks the beginning of a new synchronization phase. Inside the routine, MuPC first checks if any upc_waits from the previous synchronization phase are pending. If yes, then MuPC blocks in upc_notify until all previous upc_waits are complete. Once all pending waits are complete, MuPC checks if all the previous waits had the same value. If no, then MuPC returns an error. Following these two checks MuPC calls _UPCRTS_GetAllSync and _UPCRTS_PutAllSync (which are noop's for MuPC). After this MuPC builds a "NOTIFY" request to be broadcast to all remote processors.

- Step 2: The communication thread on the local processor occasionally checks the global memory for outstanding requests from the main thread. In this case it finds a "NOTIFY" request.

- Step 3: The communication thread extracts the remote processor number from the request. Then it sends the request to the remote processor via MPI Isend call. The communication thread on the remote processor receives the "NOTIFY" request and processes it by decrementing the notify_var variable which is set to value size - 1, for every new synchronization phase.

- Step 4, 5 and 6: These steps are same as Steps 1, 2 and 3. They occur when upc_notify is called on the remote processor.

- Step 7: The upc_wait routine is called with value n. MuPC first checks if any upc_notifies from the current synchronization phase are pending. If yes, then MuPC blocks in upc_wait until all previous upc_notifies are complete. Once all pending notifies are complete, MuPC checks if all the previous notifies had the same value. If no, then MuPC returns an error. Following these two checks MuPC calls _UPCRTS_GetAllSync and _UPCRTS_PutAllSync (which are noop's for MuPC). After this MuPC builds a "WAIT" request to be broadcast to all remote processors.

- Step 8: The communication thread on the local processor occasionally checks the global memory for outstanding requests from the main thread. In this case it finds a "WAIT" request.

- Step 9: The communication thread extracts the remote processor number from the request. Then it sends the request to the remote processor via an MPI Isend call. The communication thread on the remote processor receives the "WAIT" request and processes it by decrementing the wait_var variable which is set to value size - 1, for every new synchronization phase inside upc_notify.

- Step 10, 11 and 12: These steps are same as Steps 7, 8 and 9. They occur when upc_notify is called on the remote processor.

4.4 Memory Management and Locks

The Compaq UPC run time system interface declares a set of functions for dynamic memory allocation and functions for locking and unlocking the memory.

4.4.1 Dynamic Memory

UPC declares three functions for dynamic memory allocation and a function to free memory.

```
shared void *upc_global_alloc(size_t nblocks, size_t nbytes);
shared void *upc_all_alloc(size_t nblocks, size_t nbytes);
shared void *upc_local_alloc(size_t nblocks, size_t nbytes);
void upc_free(shared void *);
```

The three allocation functions return a shared pointer. Internally a shared pointer is implemented as a structure with 3 fields:
`va` - Virtual Address
`th` - Thread Number
`ph` - Phase

The main requirement for dynamic memory allocation is "A block of memory allocated across threads should start at the same va(virtual address)".

This requirement guides the design of dynamic memory allocation in MuPC. The approach taken by MuPC is simple. At initialization MuPC reserves a pool of memory on all threads to serve UPC dynamic memory allocation requests. The default size of this pool is 8 MB. The constant that sets this value is MEM_BLOCK. Any memory allocation request made by any UPC thread via upc_global_alloc or upc_all_alloc is routed through thread 0. Thread 0 maintains a variable base_va which points to the first free location in the pool. upc_local_alloc is implemented using malloc and is done locally on the calling thread. upc_free is a noop for MuPC.

upc_global_alloc

upc_global_alloc is a non-collective operation, i.e. any thread can call it independently and allocate memory across all threads. Assume a UPC application with 2 threads numbered 0 and 1 as shown in Figure 17. Thread 1 calls upc_global_alloc. The sequence of events that occurs is:

- Step 1: upc_global_alloc is called on thread 1.
- Step 2: the request goes to MuPC RTS on thread 1.
- Step 3: MuPC RTS on thread 1 posts a request to thread 0 to allocate memory.

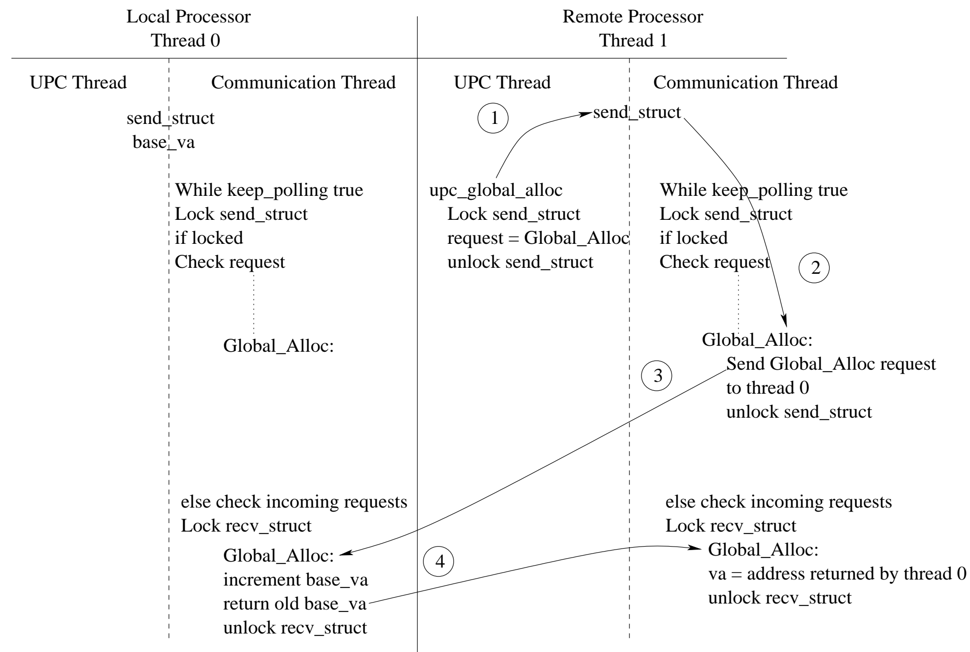


Figure 17: Global Alloc operation

- Step 4: MuPC RTS on thread 0 increments the base_va value by the amount of memory requested and returns the old base_va value. Thread 0 also checks if requested value does not exceed the available memory, determined by the MEM_BLOCK constant.
- Step 5: MuPC RTS on thread 1 sets va equal to the address returned from thread 0 and it sets, th = 0 and ph =0.
- Step 6: upc_global_alloc returns.

upc_all_alloc

upc_all_alloc is a collective operation, i.e. all threads must call it and there is implied synchronization. Memory is allocated across all threads. The sequence of events that occurs is (Figure 18).

- Step 1: Assume upc_all_alloc is called first on thread 1 and then on thread 0.
- Step 2: MuPC RTS on thread 1 posts an all_alloc request across all threads. upc_all_alloc blocks until it has heard all_alloc requests from all other threads

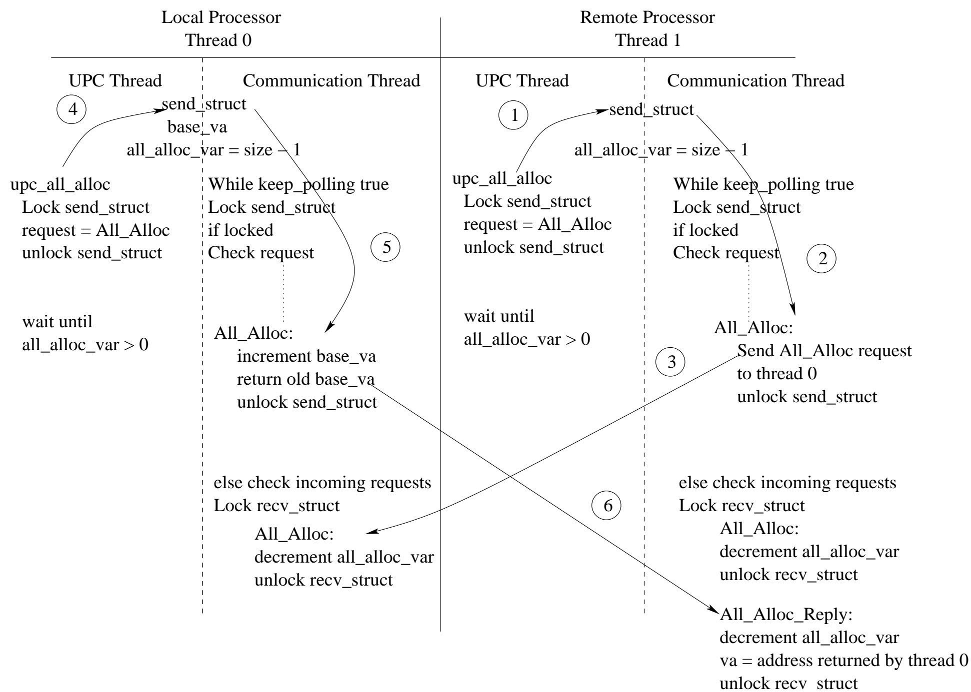


Figure 18: All Alloc operation

and thread 0 has returned a valid va. The MuPC RTS uses a variable all_alloc_var = THREADS - 1 to keep track of the number of all_alloc requests it has heard.

- Step 3: MuPC RTS on thread 0 gets the all_alloc request from thread 1 and it decrements its all_alloc_var. Subsequently, when the user code on thread 0 calls its upc_all_alloc function, MuPC RTS on thread 0 posts an all_alloc request to all threads along with the va.
- Step 4: Once MuPC RTS on each thread has heard all_alloc_var all_alloc requests and has a valid va, each of them sets va equal to the address returned from thread 0 and sets, th = 0 and ph =0.
- Step 5: upc_all_alloc returns.

upc_local_alloc

upc_local_alloc is a non-collective operation. MuPC implements local alloc using the malloc() function. The shared pointer fields are set to the following values: va = value returned by malloc, th = calling thread number and ph = 0 (Figure 19).

upc_free

upc_free is a noop for MuPC.

4.4.2 Locks

UPC provides the following lock functions

```
shared void * upc_global_lock_alloc(void);
shared void * upc_all_lock_alloc(void);
void upc_lock_init(shared void *);
void upc_unlock(shared void *);
void upc_lock(shared void *);
int upc_lock_attempt(shared void *);
```

The two lock allocation functions return a shared pointer to an object of type upc_lock_t. An important distinction between the memory allocation functions and lock allocation functions is that memory is allocated only on thread 0 in the case of lock allocation functions.

MuPC makes use of Pthread mutexes to implement locks. The upc_lock_init function is used to initialize Pthread mutexes on thread 0.

upc_lock and upc_unlock are implemented using Pthread lock and unlock routines on thread 0. upc_lock_attempt is the non-blocking version of upc_lock

upc_global_lock_alloc

upc_global_lock_alloc is a non-collective operation (Figure 19). Its operation is similar to upc_global_alloc. Assume a UPC application has 2 threads numbered 0 and 1. Thread 1 calls upc_global_lock_alloc. The sequence of events that occurs is:

- Step 1: Thread 1 calls upc_global_lock_alloc.
- Step 2: MuPC RTS on thread 1 posts a global_lock_alloc request to thread 0.
- Step 3: MuPC RTS on thread 0 updates the base_va by the size of upc_lock_t and returns the old base_va.
- Step 4: MuPC RTS on thread 1 receives the base_va from thread 0 and sets va = base_va, th = 0 and ph = 0.
- Step 5: upc_global_lock_alloc returns.

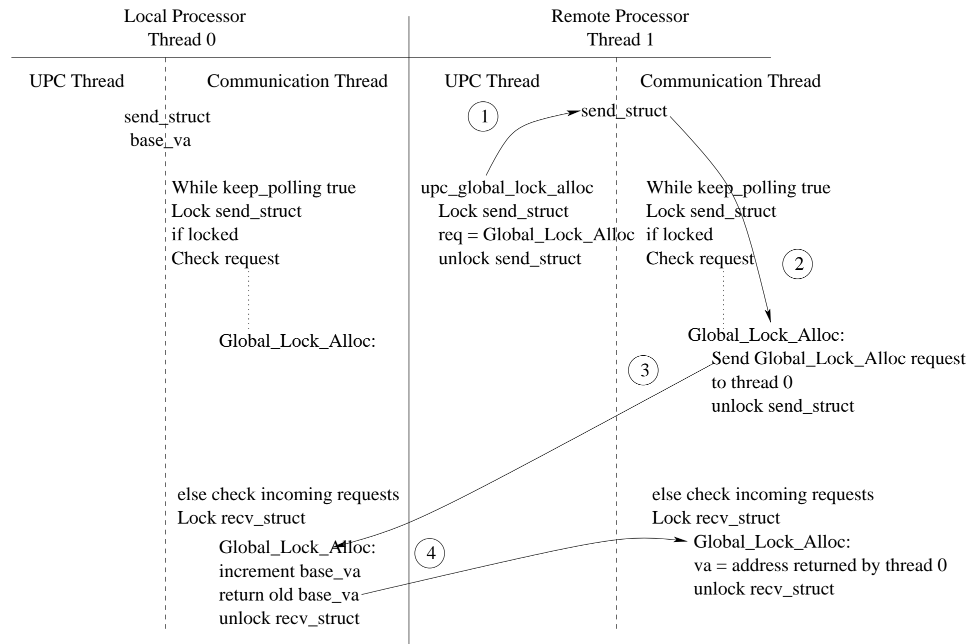


Figure 19: Global Lock Alloc operation

upc_all_lock_alloc

upc_all_lock_alloc is a collective operation (Figure 20). The sequence of events that occurs is:

- Step 1: Thread 1 calls upc_all_lock_alloc.
- Step 2: MuPC RTS on thread 1 posts a all_lock_alloc request across all threads. upc_all_lock_alloc blocks until it has heard all_lock_alloc requests from all other threads and thread 0 has returned a valid va. The MuPC RTS uses a variable all_lock_alloc_var = THREADS - 1 to keep track of the number of all_lock_alloc requests it has heard.
- Step 3: MuPC RTS on thread 0 gets the all_lock_alloc request from thread 1 and it decrements its all_lock_alloc_var. Subsequently, when the user code on thread 0 calls its upc_all_lock_alloc function, MuPC RTS on thread 0 posts an all_lock_alloc request to all threads along with va.
- Step 4: Once MuPC RTS on each thread has heard "all_lock_alloc_var" all_lock_alloc requests and has a valid va, each of them sets va equal to the base_va returned from thread 0 and it sets, th = 0 and ph =0.

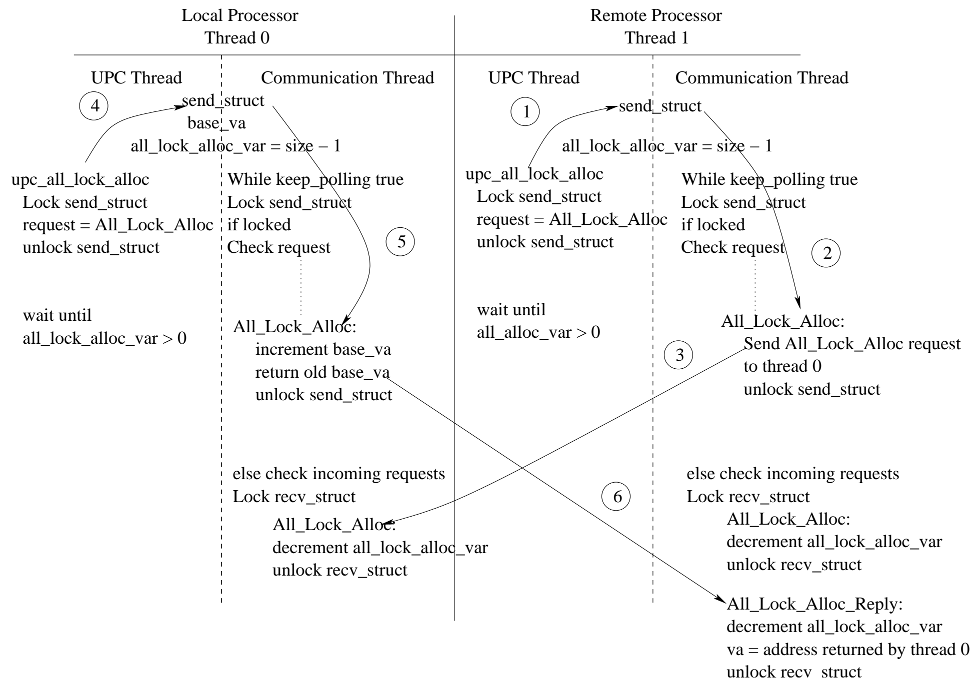


Figure 20: All Lock Alloc operation

- Step 5: upc_all_lock_alloc returns.

upc_lock_init

upc_lock_init is a non-collective operation (Figure 21). This function initializes a lock allocated by any of the UPC lock allocation functions. It is mandatory to initialize each lock before using it for the first time under MuPC.

- Step 1: Thread 1 calls upc_lock_init.
- Step 2: MuPC RTS on thread 1 posts a lock_init request to thread 0.
- Step 3: MuPC RTS on thread 0 checks if the lock is already initialized, in which case it returns immediately.
- Step 4: If the lock is not initialized it is initialized using the Pthread lock initialization function.
- Step 5: Once the lock is initialized thread 0 tells thread 1 that lock initialization is complete.

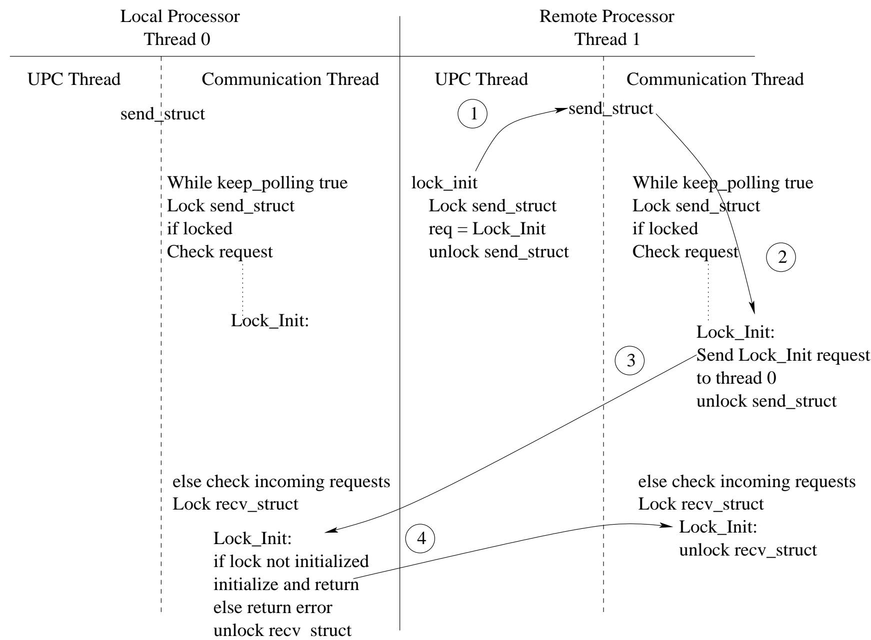


Figure 21: Lock Init operation

- Step 6: upc_lock_init returns.

upc_lock

upc_lock is used to lock a lock allocated by a UPC lock allocation routine (Figure 22).

- Step 1: Thread 1 calls upc_lock.
- Step 2: MuPC RTS on thread 1 posts a lock request to thread 0.
- Step 3: If lock is free, MuPC RTS on thread 0 locks the lock using the Pthread lock function and returns success to thread 1. If the lock is already locked then thread 0 returns failure to thread 1.
- Step 4: MuPC RTS on thread 1 returns status to the upc_lock function. If status is success then the upc_lock function returns. In case of failure, the upc_lock function blocks on a variable that indicates a change in the status of the lock. (This variable is modified whenever any thread unlocks some lock, thus upc_lock detects the status change and tries to lock again, repeating the above cycle.)

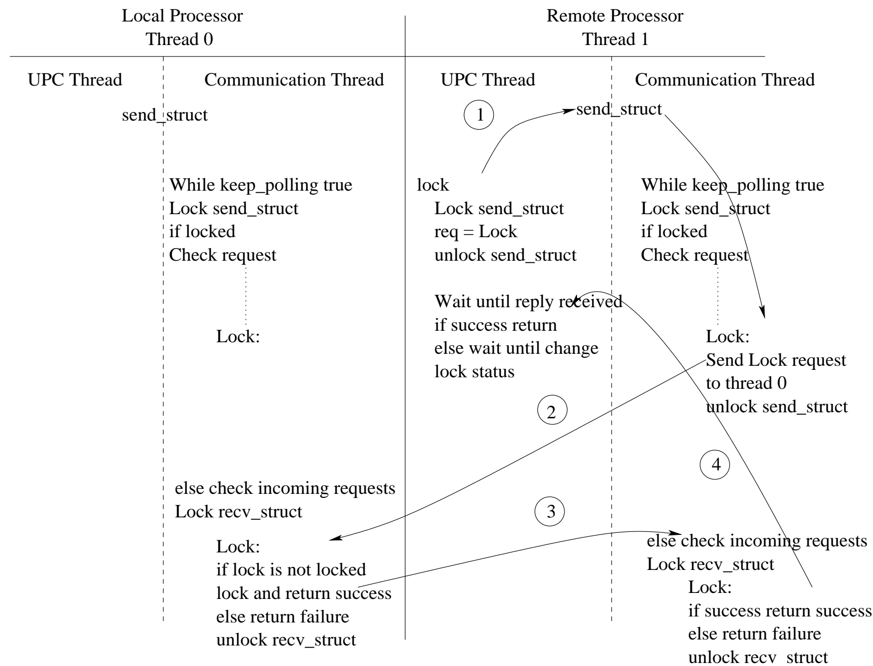


Figure 22: Lock operation

- Step 5: upc_lock returns error if the lock is not initialized.

upc_unlock

upc_unlock is used to unlock a lock allocated by an UPC lock allocation routine (Figure 23).

- Step 1: Thread 1 calls upc_unlock.
- Step 2: MuPC RTS on thread 1 posts an unlock request to thread 0.
- Step 3: MuPC RTS on thread 0 receives the unlock request and checks if the requesting thread owns the lock. If the thread does not own the lock, then an error is returned.
- Step 4: If the requesting thread owns the lock then MuPC RTS on thread 0 unlocks the lock using the Pthread unlock function and tells thread 1 about successful unlocking.
- Step 5: MuPC RTS on thread 1 reports status to the upc_unlock function.

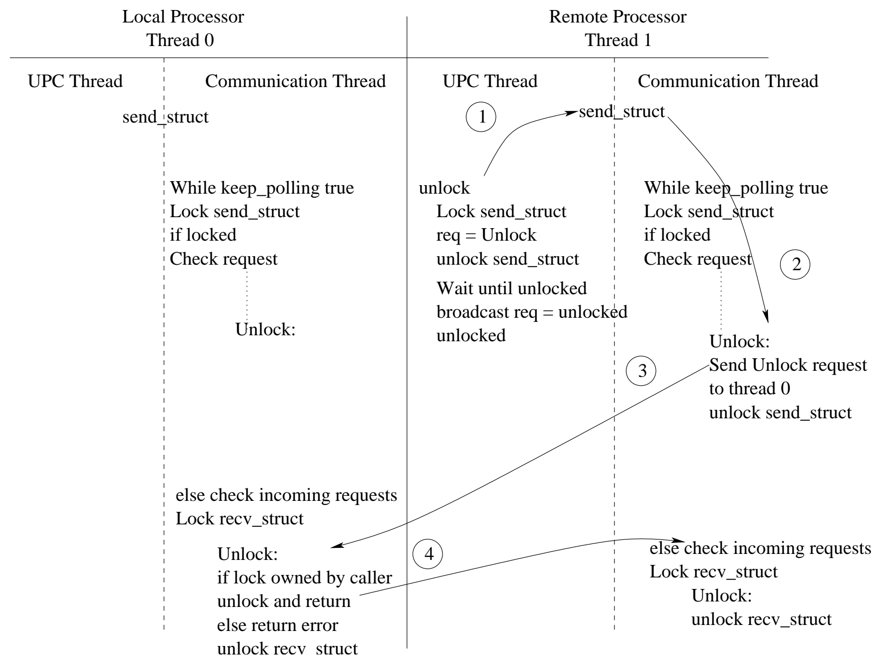


Figure 23: Lock unlock operation

- Step 6: The upc_unlock function then posts a lock_unlocked message to all threads so that all threads waiting to lock some lock can retry to lock.
- Step 7: upc_unlock returns.

upc_lock_attempt

upc_lock_attempt is the non-blocking version of upc_lock function (Figure 24).

- Step 1: Thread 1 calls upc_lock_attempt.
- Step 2: MuPC RTS on thread 1 posts a try_lock request to thread 0.
- Step 3: If the lock is free, MuPC RTS on thread 0 locks the lock using the Pthread lock function and returns success to thread 1. If the lock is already locked then thread 0 returns failure to thread 1.
- Step 4: upc_lock_attempt returns.
- Step 5: upc_lock_attempt would return in error if the lock is not initialized.

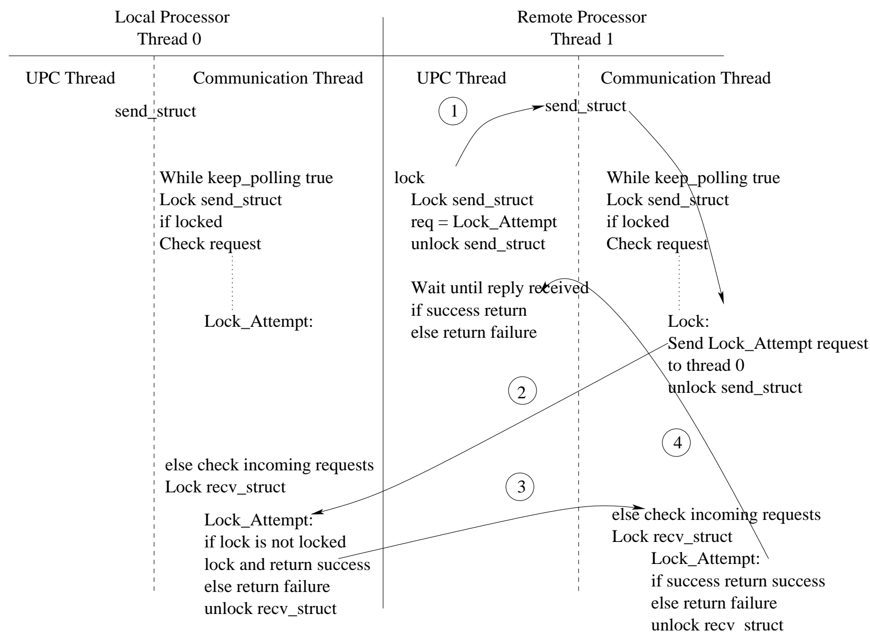


Figure 24: Lock attempt operation

4.5 Termination

The Compaq UPC run time system interface declares
`int _UPCRTS_fini(void)` as the termination function. This function does the job of memory deallocation and MPI finalization.

4.6 MuPC Restrictions

4.6.1 Strict and Relaxed

MuPC currently does not support the relaxed consistency model. All operations in MuPC are strict. All operations in MuPC are blocking by design. This means that any reference to remote memory is completed before next one starts, thus making all references strict. Its important to note that this design is in conformance to the UPC standard.

4.6.2 upc_global_exit

MuPC currently does not support the upc_global_exit library function. It is a multi exit function. The calling thread causes all threads to exit, which would start their final barrier processing. The reason MuPC does not support this function is because this function is not part of the UPC standard. Also it is not a significant language feature.

4.6.3 Final Barrier Processing

MuPC does not support final barrier processing. A final barrier matches all other barriers and blocks until all other threads reach their final barrier. Lack of support for final barrier processing is due to the absence of final barrier processing in the UPC standard.

5 Summary

5.1 Current state of MuPC

UPC offers itself as a simple yet powerful parallel programming language. The success of UPC depends on its availability across a wide variety of platforms. Compaq's run time system interface is designed with the goal of portability. MuPC adopts the same goal and has based its design and implementation around publicly available libraries for MPI and Pthreads. We have successfully implemented MuPC conforming to Compaq's run time system interface. The system has been tested successfully on the Compaq Alpha platform.

5.2 MuPC Testing

In this section we briefly describe the testing effort done for MuPC. MuPC testing was carried out as an independent project by a Michigan Tech graduate student Kian Giap Lee. MuPC testing was carried out in three phases.

- Unit testing: This involves testing individual MuPC functions.
- Functional testing: This involves testing to see if MuPC implements all the UPC constructs correctly. This was done by testing MuPC with the George Washington University test suite.
- Integration testing: This involves stress and volume testing MuPC. Volume testing implies, finding out the upper limit on datasets and number of threads MuPC can support.

5.3 MuPC Porting

One of the design goals of MuPC was to port it across a variety of platforms. As mentioned previously MuPC has been successfully implemented and tested on Alpha platforms. Currently we are testing MuPC on Sun and Beowulf clusters.

5.4 Release Information

MuPC is developed as an open source project. MuPC's source code and design documentation are available at `www.upc.mtu.edu`

References

[1] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference.* The MIT Press, 1999.

[2] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and Language Reference. CCS-TR-99-157, May 13 1999.

[3] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing:Deisgn and analysis of algorithms.* Benjamin/Cummings Publishing House, Inc., Redwood city, California, November 1993.

[4] Vincent W. Freeh. A comparision of Implict and Explicit parallel programming. *Journal of Parallel and Distributed Computing*, (34(1):50-65), April 1996.

[5] Mark P. Jones and Paul Hudak. Implicit and explicit parallel programming in Haskell. Technical Report YALEU/DCS/RR-982, Department of Computer Science, Yale University, 1993.

[6] William W. Carlson and Jesse M. Draper. Distributed Data Access in AC. In *Fifth ACM Sigplan Symposium on Principles and Practices of Parallel Programming*, pages 39–47, 1995.

[7] Guy L. Steele Jr. and J. Rose. C*: An Extended C Language for Data Parallel Programming. In *Proceedings of the Second International Conference on Supercomputing*, volume 2, pages 2–16, May 1987.

[8] ANSI-Programming languages-C. ISO/SEC 9899, May 2000.

[9] David Culler, Andrea Dusseau, Seth Copen Goldstien, Arvind Krishnamurthy, Steven Lumetta, Thirsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing 93*, pages 262–273, November 15-19 1993.