View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by KITopen

RThreads — a Uniform Interface for Parallel and Distributed Programming

Bernd Dreier Markus Zahn
University of Augsburg
Dept. of Mathematics
D-86135 Augsburg, Germany
{dreier,zahn}@Informatik.Uni-Augsburg.DE

Theo Ungerer
University of Karlsruhe
Dept. of Computer Design and Fault Tolerance
D-76128 Karlsruhe, Germany
ungerer@Informatik.Uni-Karlsruhe.DE

Abstract

Several distributed systems and software packages allow the use of workstation clusters as a virtual machine. In general, the interfaces to these environments use different programming paradigms for parallel and distributed computing, e.g. multithreading within a multiprocessor workstation and message passing or remote procedure calls for distributed computing. Porting applications to other distributed systems is a difficult task and many different programming paradigms have to be learned.

We introduce a uniform interface for parallel and *distributed programming based on POSIX Threads. By providing a* global data space *we are able to raise the concept of threads to a higher level of concurrency — threads may be spread over several heterogeneous machines and are therefore called remote threads (RThreads). Up to now, we have implemented the RThread interface on top of PVM and DCE.*

1. Introduction

Today, computer networks grow in size and importance. Workstations coupled by a high-speed network represent an efficient parallel virtual machine. A workstation cluster is often composed of multiprocessor workstations. The processors within a multiprocessor workstation share global memory, whereas a workstation cluster is coupled by a standard network, e.g. Ethernet, FDDI, or ATM. Synchronization of parallel activities among the processors within a workstation is done by access to global variables. In a workstation cluster messages are sent via UNIX ports. Communication within a workstation is much faster than between the workstations of a cluster. As a consequence, medium-grained parallelism can be successfully exploited within a multiprocessor workstation, but only very coarse-grained parallel activities should be distributed over the workstations of a cluster. However, a network of dozens or hundreds of workstations is much more powerful than a multiprocessor workstation with up to four processors, provided that the algorithm is appropriate and a distributed environment is available.

Several distributed systems (e.g. DCE [5]) or software packages (e.g. MPI [3, 8], PVM [7], Linda [1]) allow networked computers to appear as a single concurrent computational resource. As a matter of fact, all these programming environments require to learn a new programming model. DCE supports threads and remote procedure calls, PVM is a message passing system and Linda introduces global data in a tuple space. The necessity to learn and apply completely new paradigms often retards the entry to distributed computing. Furthermore, porting a distributed program from one platform to another often requires a complete redesign of the algorithm.

Most of the distributed programming environments support only a single level of parallelism. Only coarse-grained parallelism between whole UNIX-processes is used in the cases of PVM — the de-facto standard — and of MPI — the future standard in scientific computing. The distribution of the processes over the network is obligatory. One of the urgently wanted improvements is the introduction of medium-grained parallelism using light-weight processes, i.e. threads. DCE already supports medium-grained parallelism by POSIX Threads[1] (PThreads), the distribution of coarse-grained components by remote procedure calls is also possible. However, two completely different programming paradigms for different levels of granularity have to be used.

To address these issues, we define a programming interface, which covers medium- and coarse-grained parallelism in a uniform manner. Medium-grained parallel components are executed within a workstation, and the distribution of coarse-grained components over a workstation cluster is possible. We decided to start from a well-known, already existing programming paradigm. Since we do not

[1] The POSIX series of standards include POSIX.1c, the standard for parallel, multithreaded programming.

want to provide a novel distributed environment, we implemented the uniform programming interface on top of existing distributed systems.

Shared memory models provide an easy entry to parallel programming. POSIX Threads are a wide-spread representative of this class. PThreads are also used in several modern operating systems like Sun Solaris, OS/2, or Windows 95.

Therefore, we decided to base our programming interface on the PThread model. Due to the underlying global address space, POSIX Threads cannot be spread over distributed memory systems. Thus, we have to expand the PThread model to enable distributed execution.

2. The RThread programming model

The well-known PThread model allows the creation of light-weight processes running in the same address space. There is no hierarchy of threads in this model, i.e. a newly created thread is treated equally to the other threads of the process, including the initiating one. The access to global data can be synchronized by mutexes and condition variables, which are part of the common address space themselves. Since all threads execute in the same address space, global data (including synchronization data) can be accessed directly. Figure 1 illustrates the PThread model; all threads execute in one process, which is represented by the dashed line.

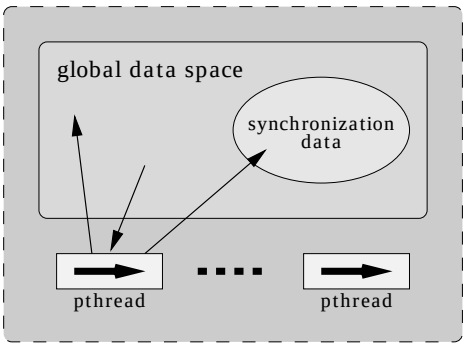

Figure 1. The PThread model

To expand the PThread model to distributed execution, the distribution of parallel components (i.e. threads) must be enabled. Due to the creation of such "remote threads", we call the introduced programming interface *RThread* model. In the RThread model each RThread is running in its separate address space as shown in Figure 2. The different processes (possibly residing on different hosts) are represented by dashed boxes like in figure 1. Since shared memory is not available in distributed heterogeneous computer systems, the RThread model provides a *global data space* for all RThreads. A buffer in each RThread's address space maps to parts of the global data space. All computation in an RThread is done on its buffer. The exchange of data between buffer and global data space is achieved by explicit read/write-operations of the RThread. Each read- or write-operation can affect multiple data items.

Synchronization data is also part of the provided global data space. In contrast to the other part of global data, synchronization data is not buffered. The synchronization operations of the PThread model are expanded to work between several machines.

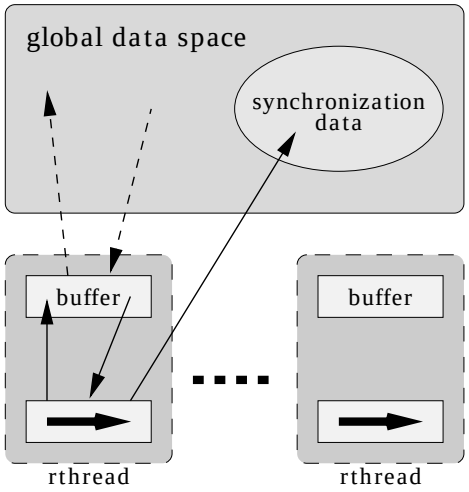

Figure 2. The RThread model

Notice that each RThread in figure 2 may contain several PThreads, i.e. the RThread model introduces a hierarchical view: All PThreads of one RThread run in the same address space and share the same buffer. A common global data space is provided for all RThreads, which have equal rights.

3. A PThread example

To illustrate programming with RThreads and their similarity to programming with PThreads, we start this section with the following PThread example program. In section 5 a solution for the same problem according to the introduced RThread programming interface is described.

The sample program multiplies the two matrices `m1` and `m2`: A pool of five threads computes the result matrix `m3` element by element. The global variables `row` and `col` indicate the next element to be computed. After finishing the computation of an entry, each thread fetches a new job by incrementing `row` and `col`. The fetching of a job is protected by `pthread_mutex_lock()` to ensure mutual exclusion during this process. The main thread waits for the end of all five threads, afterwards the result could be used for further computations.

```
int m1[200][200], m2[200][200], m3[200][200];
int row=0, col=0;

pthread_mutex_t lock; /* protects row and col */
pthread_t worker_threads[5];

void *worker( void *dummy )
{
  int myrow, mycol, i, result;

  while (pthread_mutex_lock( &lock ), row<200 )
  {
    myrow=row;
    mycol=col++;
    if (col == 200)
    {
      col = 0;
      row++;
    }
    pthread_mutex_unlock( &lock );

    result = 0;
    for ( i=0; i<200; i++ )
      result += (m1)[myrow][i] * (m2)[i][mycol];
    (m3)[myrow][mycol] = result;
  }
  pthread_mutex_unlock( &lock );
  pthread_exit(0);
}

void main( void )
{
  int i, j;

  pthread_mutex_init( &lock,
    pthread_mutexattr_default );
  /* initialization of m1 and m2 left out */

  for ( i=0; i < 5; i++ )
    pthread_create( &worker_threads[i],
      pthread_attr_default,
      (pthread_startroutine_t) worker, NULL );
  for ( i=0; i < 5; i++ )
    pthread_join( worker_threads[i], NULL );
}
```

4. The RThread programming interface

As mentioned above, programming with RThreads is very similar to PThread-like programming. We provide an RThread equivalent for each PThread function (e.g. `rthread_create()`, `rthread_mutex_lock()`) and for each PThread type (e.g. `rthread_t`, `rthread_mutex_t`).

`rthread_create()` spawns a thread on a possibly remote host. The RThread synchronization operations behave like their PThread equivalents. However, the synchronization is performed between threads on different machines.

We introduce two additional functions to exchange global data between the RThread's buffer and the global data space: `rthread_read()` and `rthread_write()`. For example, the single data item `row` is copied from the global data space into the RThread's buffer (and vice versa) by the following function calls:

```
rthread_read( RTHREAD_long, RTHREAD_row, 0, 0, 1,
  RTHREAD_DATA_DONE );
rthread_write( RTHREAD_long, RTHREAD_row, 0, 0,
  1, RTHREAD_DATA_DONE );
```

The first parameter defines the data type, followed by the variable name, a first index, a last index and a stride, finished by `RTHREAD_DATA_DONE`. The data type is specified to allow data conversion in heterogeneous networks. Data types and variables are named with an `RTHREAD_` prefix to use labels defined by the RThread package (see section 6). First index, last index and a stride can be used to access parts of an array. For example, the following statement reads the k-th column of the $n \times n$ matrix `m2`.

```
rthread_read( RTHREAD_long, RTHREAD_m2, k,
  (n-1)*n + k, n, RTHREAD_DATA_DONE );
```

Due to a variable argument list, multiple read accesses can be combined in a single `rthread_read()` statement:

```
rthread_read( RTHREAD_long, RTHREAD_row, 0, 0, 1,
  RTHREAD_long, RTHREAD_col, 0, 0, 1,
  RTHREAD_DATA_DONE );
```

Instead of terminating the data access by `RTHREAD_DATA_DONE`, multiple `rthread_read()`s can be grouped using `RTHREAD_DATA_CONTINUE`.

Writing to the global data space is done with `rthread_write()` accordingly. The initializiation of the RThread package is performed by the functions `rthread_main_init()` and `rthread_remote_init()`. They are described in section 5.

5. An RThread example

For an RThread-implementation of the matrix multiplication algorithm described in section 3, two programs must be created: The "main thread program" and the "remote thread program". Both have to define buffer space for global data and initialize the RThread package by calling `rthread_main_init()` respectively `rthread_remote_init()`. The main thread program has to pass the file name of the remote thread program to the initialization function. According to the PThread example program given above, it starts and joins the remote threads. The function `worker()` from the former example program is left out, because it is not used as a start function in a local PThread.

```
/* main thread program */
#include "rthread.h" /* additional includes */
#include "matmul_rthread.h"
```

```
/* buffer */
int m1[200][200], m2[200][200], m3[200][200];
int row=0, col=0; /* buffer */

rthread_mutex_t lock; /* protects row and col */
rthread_t worker_threads[5];

void main( void )
{
  int i, j;

  /* initialize rthread package */
  rthread_main_init( "matmul_remote" );

  rthread_mutex_init( RTHREAD_lock,
    rthread_mutexattr_default );
  /* initialization of m1 and m2 left out */

  for ( i=0; i < 5; i++ )
    rthread_create( &worker_threads[i],
      rthread_attr_default, RTHREAD_worker,
      (rthread_addr_t) NULL );
  for ( i=0; i < 5; i++ )
    rthread_join( worker_threads[i], NULL );
}
```

In the "remote thread program" buffer space for all or part of the global data is allocated similar to the "main thread program". The `worker()` function of the "remote thread program" corresponds to the start function of the PThread example program given in section 3. In addition, explicit read or write statements have to preserve the consistency of buffer and global data space.

Therefore, in the following example program the first read access to the variable `col` in the local buffer is preceded by `rthread_read( ..., RTHREAD_col, ...)`. The modified value is written to global data space by `rthread_write( ..., RTHREAD_col, ...)` afterwards. Mutual exclusion of threads accessing `col` concurrently is ensured by `rthread_mutex_lock()` corresponding to `pthread_mutex_lock()` in the PThread program.

```
/* remote thread program */
#include "rthread.h"
#include "matmul_rthread.h"

/* buffer */
int m1[200][200], m2[200][200], m3[200][200];
int row=0, col=0; /* buffer */

rthread_mutex_t lock; /* protects row and col */
rthread_t worker_threads[5];

void *worker( void *dummy )
{
  int myrow, mycol, i, result;

  while (rthread_mutex_lock( RTHREAD_lock ),
         rthread_read( RTHREAD_long, RTHREAD_row,
           0, 0, 1, RTHREAD_DATA_DONE ),
         row<200 )
  {
    myrow=row;
    rthread_read( RTHREAD_long, RTHREAD_col, 0,
      0, 1, RTHREAD_DATA_DONE );
    mycol=col++;
    rthread_write( RTHREAD_long, RTHREAD_col, 0,
      0, 1, RTHREAD_DATA_DONE );
    if (col == 200)
    {
      col = 0,
      rthread_write( RTHREAD_long, RTHREAD_col,
        0, 0, 1, RTHREAD_DATA_DONE );
      row++;
      rthread_write( RTHREAD_long, RTHREAD_row,
        0, 0, 1, RTHREAD_DATA_DONE );
    }
    rthread_mutex_unlock( RTHREAD_lock );

    rthread_read( RTHREAD_long, RTHREAD_m1,
      myrow*200, myrow*200 + 200-1, 1,
      RTHREAD_long, RTHREAD_m2, mycol,
      (200-1)*200 + mycol, 200,
      RTHREAD_DATA_DONE );
    result = 0;
    for ( i=0; i<200; i++ )
      result += (m1)[myrow][i] * (m2)[i][mycol];
    (m3)[myrow][mycol] = result;
    rthread_write( RTHREAD_long, RTHREAD_m3,
      myrow*200 + mycol, myrow*200 + mycol, 1,
      RTHREAD_DATA_CONTINUED );
  }
  rthread_write( RTHREAD_DATA_DONE );
  rthread_mutex_unlock( RTHREAD_lock );
  rthread_exit(0);
}

int main ( void )
{
  rthread_remote_init();
  exit( 0 );
}
```

6. Implementation

The RThread-functions are implemented in libraries. For compilation of RThread-programs an additional application-specific header file will be created automatically. Each label beginning with `RTHREAD_` which occurs in a RThread function call is added to the header file. The header contains a mapping between the global data space and the buffers. A label represents an address of an element in the global data space. The header file maps these addresses to local addresses in the buffer of the given RThread. The `RTHREAD_` prefixed variable and function names are required to get unique identifiers valid in the main *and* the remote threads, where the identifiers naturally have different addresses.

In our current implementation the main thread program behaves like a master program. It contains the global data space, serves the read and write accesses to the global data space, executes the synchronization operations and starts processes embedding the remote threads.

The introduced RThread programming interface is independent of the underlying distributed systems. These are used for network transport and start of remote programs only. Up to now, we have successfully implemented the RThread-functions on DCE and PVM platforms. The use of different libraries enables the execution of the same program either in a PVM or DCE environment. In the DCE implementation the remote threads execute read, write and synchronization operations by remote procedure calls to the master. In the case of PVM this communication is realized by its message passing facilities. PVM already supports dynamic creation of remote processes. To accomplish this task in the DCE environment, we developed a runtime system [2], which allows for load balancing additionally.

7. How to get efficient programs

An important aspect to get efficient programs is the reduction of network traffic caused by read and write operations. Usually, performance can be increased by transferring a higher amount of data in a single network transaction instead of several transactions with less data. `rthread_read()` and `rthread_write()` are designed for combining multiple data accesses in a single network transaction. The parameters "first index", "last index" and "stride" allow the access to parts of arrays. A variable argument list supports grouping of multiple data requests in a single function call. Furthermore, the use of `RTHREAD_DATA_CONTINUED` enables the collection of data accesses across several function calls, which is terminated by `RTHREAD_DATA_DONE`. This feature is especially useful in loops (see example program).

The programmer should carefully consider data dependencies between different RThreads. Consistency of buffer and global data space is only ensured by explicit use of `rthread_read()` and `rthread_write()`. The programmer is able to set his own level of consistency [4, 6]. Therefore, the knowledge of the program semantics should be employed by the programmer to group or collect data accesses, thereby increasing performance.

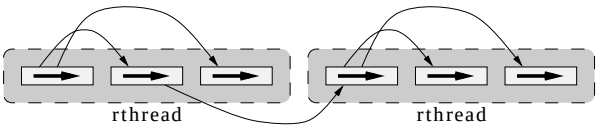

Figure 3. Combining RThreads and PThreads

It is possible to mix RThreads and PThreads in a parallel program, i.e. RThreads may contain several PThreads which are executed in the RThread's process (see figure 3). Therefore, tasks with high communication needs may run simultaneously without being distributed. This is a great improvement compared to systems like PVM where parallelism forces distribution. Calling functions of the RThread-libraries is allowed to the PThreads, too: In Figure 3 a PThread within an RThread creates another RThread. The RThreads themselves are possibly spread over a heterogeneous workstation cluster. Thus, they should obviously be constrained to computation intensive tasks with few communication needs.

8. Conclusions and further work

We introduced a model for parallel and distributed programming as extension of the well-known POSIX-Threads. The RThreads provide a uniform programming interface for medium- and coarse-grained parallel programs. Coarse-grained tasks can be distributed to remote hosts. Our current implementations are based on PVM and DCE, as representatives of a message-passing environment respectively of a RPC-based distributed operating system. Implementations on other distributed platforms (e.g. MPI) are in progress. Testing of the DCE- and PVM-based implementations with several sample programs is finished. The examples show good speedup. More complex applications will show the ease of programming and give realistic results on the efficiency of the implementations.

Our programming environment relies on read and write accesses explicitly set by the programmer. In future, a precompiler may help optimizing global data exchange. Moreover, we survey automatic transformation of multithreaded programs into our RThread model.

References

[1] N. Carriero, D. Gelernter, T. Mattson, and A. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20:633–655, April 1994.

[2] B. Dreier and M. Zahn. Entwicklung einer verteilten Programmierumgebung für das DCE. Master's thesis, Universität Augsburg, October 1993.

[3] M. P. I. Forum. Mpi: A message-passing interface standard. Technical report, University of Tennessee, June 1995.

[4] K. Hwang. *Advanced Computer Architecture*. McGraw-Hill, New York, 1993.

[5] H. W. Lockhart Jr. *OSF DCE Guide to Developing Distributed Applications*. McGraw-Hill, Inc., 1994.

[6] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, pages 52–60, August 1991.

[7] V. Sunderam, A. Geist, J. Dongarra, and R. Mancheck. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20:531–546, April 1994.

[8] D. Walker. The design of a standard message-passing interface for distributed memory concurrent computers. *Parallel Computing*, 20:657–673, April 1994.