Data Handover: Reconciling Message Passing and Shared Memory

Jens Gustedt

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Data Handover: Reconciling Message Passing and Shared Memory

Jens Gustedt

N° 5383

November 2004

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--5383--FR+ENG

Data Handover:
Reconciling Message Passing and Shared Memory

Jens Gustedt

Thème NUM — Systèmes numériques
Projet AlGorille

Rapport de recherche n° 5383 — November 2004 — 17 pages

Abstract: We present a programming paradigm and interface that aims to handle data between parallel or distributed processes that mixes aspects of message passing and shared memory. It is designed to overcome the potential problems in terms of efficiency of both:

- memory blow up and forced copies for message passing and
- data consistency and latency problems for shared memory.

Our approach attempts to be simple and easy to understand. It contents itself with just a handful of functions to cover the main aspects of coarse grained interoperation upon data.

Key-words: efficient data management, message passing, shared memory

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,

Cession de données : réconcilier passage de message et mémoire partagée

Résumé : Nous présentons un paradigme et une interface de programmation qui ont pour objectif de gérer des donées entre processus parallèles ou distrubués. Ce paradigme mélange des aspets de passage de messages et de la mémoire partagée et est conçu pour surmonter les problèmes potentiels liés aux des deux approches en terme d'efficacité :

- Les problèmes d'explosion mémoire et des copies forcées pour le passage de messages ;
- La cohérence des données et le problème de la latence de la mémoire partagée.

Notre approche tente à la fois d'être simple et facile à comprendre. Elle se contente d'une poignée de fonctions pour couvrir les principaux aspets de l'interopération à gros grain sur les données.

Mots-clés : gestion efficace de données, passage de message, mémoire partagée

1 Introduction and Overview

A lot of sophisticated models, systems and programming languages and libraries are nowadays available for parallel and distributed computing. But nonetheless of that multitude of choices, the majority application designers and programmers choose among quite a few interfaces when it comes to implement production systems. These interfaces fall into two major classes, those coming from the world of parallelism (e.g shared segments, different sorts of threads or OpenMP) or the world of distributed computing (e.g PVM, MPI, RPC, RMI or Corba). Although most of them are also available in the other context (e.g MPI on mainframes or threads on distributed shared memory systems) in general the performance of these tools suffer when they are applied in framework for which they have not been designed originally. This reduced performance is not a matter of "lack of good implementations" but is due to conceptual difficulties:

- Message passing applied on shared memory architectures introduces a substantial memory blow up (compared to a direct implementation) and forces unnecessary copies of data buffers.
- The simplified transposal of shared memory algorithms onto distributed platforms provoke a high complexity when it comes to guarantee consistency of data. Latency problems often result in disappointing performance.

So a big performance gap remains when it comes to applications that are supposed to be executed on platforms for which we can't know their nature beforehand. In particular in grid environments inherently will have this property that an application is launched on an *unknown* environment. For various reasons, the client (buying computing power) and the provider (selling computing power) should know as little as possible about each other. They should see each other through a mediator that is able to provide the necessary guarantees to the satisfaction of both sides, e.g mutual trust, performance, availability and many more. It will be almost impossible (or at least expensive) to impose a particular flavor of platforms. Even worse, a client who is demanding a lot of computing power will likely be served by a mixture of these concepts, namely a conglomerate of smaller to middle-sized parallel machines tied together with a virtual network of high bandwidth but with mediocre latency.

Valiant's seminal paper on the BSP model, see [1], has triggered a lot of work on different sides (modeling, algorithms, implementations and experiments) that showed very interesting results on narrowing the gap between the message passing and shared memory paradigms. But when coming to real life code, implementers tend to turn back to the "classical" interfaces, even when they implement with a BSP-like model in mind. Thus again, even though on the modeling side they have in principle overcome the separation of the two, the realization then tends to suffer from one of the performance issues as mentioned above.

With this paper we try for a proposal of an interface that according to our biased experience captures the main and essential features that a library that serves as a base for programming in grids (or, who knows, perhaps once "The Grid") should have:

Simplicity: The interface should be easy to use and not be forcing difficult changes in programming habits. It should also be easy to implement on top of existing interfaces and libraries, making it also easily portable.

Performance: The interface should allow for an easy evaluation of the performance of the code that is using it *and* this performance must be competitive to the performance of other implementations that is done with other interfaces and the corresponding libraries.

Interoperability: The interface should be as close as possible to known and accepted standards to ensure an easy interconnection of heterogeneous systems, with different material features, different OS or within different administrative domains.

Overview

First, in the next section, we will review what we think are the principal pros and cons of both base paradigms (message passing and shared memory). Then, in Section 3, we will propose the main features of our interface and exemplify them in Section 4. A brief discussion of a first implementation is followed by some concluding remarks. In an appendix, we give the code of a matrix multiplication example and the C and C++ interfaces.

Notation

In the following we will assume that the reader is familiar with one or several of the commonly used interfaces for message passing and programming shared memory. Since they are the closest to the approach that we develop we will base our examples upon MPI (see [2]) and POSIX shared memory *segments* (see [3])[1]. We will usually denote functions of these APIs by their C language interfaces using a `typewriter` font, such as `shm_open`, and also regular expressions like `MPI_*recv` to denote a group of functions. For the interface as it is proposed here, we often will tend to denote the functions with their C++ name and simply avoiding the `DHO_` prefix if possible.

2 Advantages and Disadvantages of the two base paradigms

2.1 Message Passing

The great success of the message passing paradigm in recent years is certainly due to a multitude of factors. Here we will emphasize on those which seem the most important for us in the following and which must be saved.

Simplified Data Control Data control within message passing environments is conceptually simple. The programmer controls the buffers that hold the data and has precise

[1] The later is not to be confounded with the POSIX THREAD interface, `pthread_*`.

synchronization points after which he may assume that the data has been successfully transmitted. The programmer completely controls the consistency of the data.

Standardization The big success of MPI in particular is also due the fact that it is standardized with very little inherent ambiguity on the semantics. In particular this allows for several concurrent implementations, there are high quality public domain implementations as well as proprietary ones on all major platforms.

Efficiency in distributed environments In distributed environments the message passing paradigm is very efficient. In particular the possibility of having non-blocking communications makes the best of existing architectures when relaxing the synchronization constraints between network hardware and processors. Not only that this efficiently uses the inherent parallelism between the different components of modern architectures but also it allows to overrule one of the fundamental problems, namely that the latency of a network connection is linearly related to its physical distance.

The message passing libraries also have improved a lot in recent years on their efficiency when executed on parallel machines. But inherently due to the message passing paradigm itself they *must* suffer from the following two closely related problems.

Memory blow up: For messages, there is always a sending side and a receiving side. So both, the sending process and the receiving process must allocate memory for a message. At least temporarily, the consumption of memory doubles.

Extra copy operations: To realize the data transfer, the data must be copied from the sender to the recipient. This even thought the sender perfectly knew when preparing the send buffer where the data should have ended up. So for data oriented computation, the running time increases substantially.

2.2 Shared Memory

From the point of view of a programmer, the shared memory paradigm has several advantages, from which we emphasize on the following two:

Random Access The data of all processes is directly accessible from any other process. This avoids implementation of supplementary control ("please send me such and such data") and simplifies the view of the data as a whole for the programmer.

Efficiency on parallel architectures The data transfer can be completely delegated to lower levels of the system architectures, usually a combination of OS and hardware. This helps to make implementations very efficient: data access is mainly bound by hardware parameters such as the bandwidth and latency of the interconnection bus.

Especially when realizing the shared memory paradigm on distributed architectures this approach suffers from at least from two problems, data consistency and latency. But they are also inherently present when realizing shared memory architectures. Data consistency

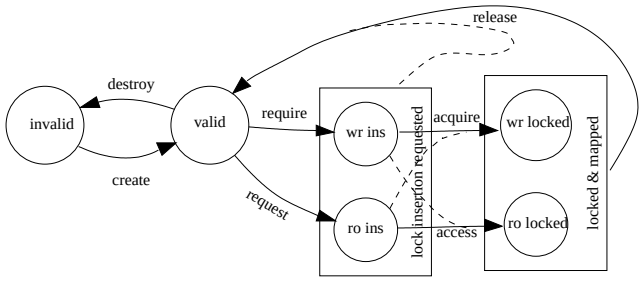

Figure 1: Essential part of the state diagram of a handle. Less common operations are dashed. Omitted are operations that not encouraged such as direct locking or destructions from other states than "valid".

problems contribute much to the difficulties of the interaction between OS and hardware (cache coherence, TLBs,...). A low latency interconnection bus takes a major part in the prize of a high end parallel machine.

Data consistency A major danger (and programming difficulty) of random access is concurrent access to memory. Who wins when writing at the same moment to the same location? Is the data that a process reads still valid?

Latency To realize shared memory and cover the consistency problems the realization of a fine grained synchronization between the processes is important. E.g the fact that some data has a new value must be propagated to the readers of that data. This needs at least time of the (hardware) latency of the interconnection. In a distributed setting, latency is a major obstacle since it is limited by the quotient of the distance and the speed of light.

3 The proposal of an interface

The main functionalities of the interface that we are proposing are summarized int Table 1. It contains interfaces to access the data structure `DHO_t` as a whole (`create`, `duplicate` and `destroy`), to gain write access to all or part of the data (`require` and `acquire`), to gain read-only access (`request` and `access`) and to resign from accessing the data (`release`). The entire interface as well as an analogous interface for C++ are given in the appendix.

3.1 Abstracting from memory: data handles

At first, we want to combine the simplicity of control of the message passing interface with the random access of memory. Therefore it is important to introduce a level of abstraction between *memory* and *data*. Whereas data is an ideal concept (e.g a Shakespeare sonnet),

name	blocks	returns	description	similar in MPI sender	similar in MPI receiver	similar in POSIX
`DHO_t`			data type representing the user data	buffers and `MPI_Request`		file descriptor
`DHO_create`	list		creation of a handle	buffer creation		`{shm_}open`
`DHO_resize`	no		resizing	buffer resizing		`ftruncate`
`DHO_duplicate`	no		duplicate a handle			
`DHO_destroy`	list, wr		destruction of a handle	buffer deletion		`close` and `{shm_}unlink`
`DHO_require`	rd		require future exclusive (read-write) access, write-prefetch the data		`MPI_Irecv`	
`DHO_acquire`	yes	`void*`	instantiate data in memory		`MPI_Wait`	`fcntl` for write-locking and `mmap`
`DHO_request`	rd		request future shared (read-only) access, read-prefetch the data		`MPI_Irecv`	
`DHO_access`	yes	`void const*`	instantiate data in memory		`MPI_Wait`	`fcntl` for read-locking and `mmap`
`DHO_release`	list, wr		relinquish access	`MPI_Isend`		`munmap` and `fcntl` for unlocking
`DHO_test`	no	`int`	test for available data in memory		`MPI_Test`	`fcntl` with try-locking

Table 1: *The principal C interfaces.* In the column *blocks* a "yes" indicates that the calling process might block until other processes release the data. A "rd" or "wr" indicates that although the process will not be blocked that the system might be busy in doing preparative reading or clean-up writing. A "list" indicates that although the process will not be blocked on locks that are placed by others, that it might block until *all* its previously posed announcements (**`require`** or **`request`**) are known to be taken into account. If there is no indication of a return value, the function generally returns an error code. The C++-interface should throw an exception instead.

memory is the (sensible, sharable) instantiation of such a concept (e.g a print of Shakespeare's sonnets). In that sense the message passing paradigm handles data whereas the shared memory paradigm handles memory:

- A message is data with two different instantiations, one at the sender and one at the receiver.
- The question what the actual data in a shared memory is, points just to the consistency problem as mentioned above.

Both, memory *and* data, may change in time, e.g we may speak of a sonnet as printed in such and such edition.

In our proposed paradigm the processes don't share memory but *data handles* called `DHO_t`. Every data shall correspond to a common data handle that can be accessed by the processes. The processes shall negotiate *control* over the data via such a handle. They shall request an instantiation of the data in their individual memory, a *mapping*, via such a handle by means of the functions `DHO_access` (for read-only mappings) and `DHO_acquire` (for read-write mappings). The functions shall return pointers to memory of the desired size which will be properly initialized with the data.

Currently in several different contexts this abstraction exists and is well mastered. Examples for interfaces that implement features similar to such handles are `MPI_Requests` for MPI and file descriptors for POSIX files and POSIX shared memory segments.

Figure 1 resumes the essential parts of the state diagram of an individual handle. Essential here indicates that these are the parts for which the interface is designed and which should correspond to the most efficient usage of a DHO-library. Others are possible, see below, but should correspond to exceptional cases in usage.

3.2 Separating access: locking

An interesting feature of the message passing paradigm is the way control over the data passes from one process to the other. By issuing an `MPI_*send` operation the sender passes control over to the receiver. By issuing an `MPI_*rec` operation (and an eventual `MPI_*wait`) the receiver takes over control. Every process knows exactly over what data it has control. This particular feature is what we call *data handover*, a well defined protocol for the passage of responsibility over data from one process to another.

On the other hand, once the message is finally received, the association of the receive buffer with the abstract concept "message" is lost, the data has no approved identity anymore, the programmer has to keep track of that identity himself. So instead of asking the programmer to temporarily associate some memory (buffer) to a message, we propose to provide the allocation of memory by the library. Thereby we may ensure that the association between the data and its instance is always maintained.

We propose to use a concept that is similar to POSIX' read-write-locks (`pthread_rwlock_*`[2]) and to mandatory file-locking with `fcntl`[3]. Multiple handles might `access` the same data simultaneously for reading, placing a read-lock that will prevent any other handles from modifying it. At most one handle at a time may `acquire` a data to be able to modify it. While it holds this acquirement all processes (including the one that obtained the lock) that attempt access to the same data through another handle will be blocking.

The programmer may temporarily resign from the previously gained access to the data by means of `release`. A call to that function shall invalidate the pointer that was previously return by `access` or `acquire`. The data itself becomes inaccessible through that handle but shall never be lost as long as any handle on it is alive. The process might newly regain access to the data (calls to `access` or `acquire`) but the mapping then might use a different address than before.

3.3 Forethought: timely announcing changes on control

Another advantage of the message paradigm is that it allows for so-called non-blocking receives, that is it allows to distinguish the announcement of a future take of control (`MPI_Irecv`) from the effective instantiation of the data in memory (`MPI_Wait`)[4]. This distinction is useful to circumvent latency problems, and in fact it is very often used to implement overlapping of communication and computation with MPI.

Such an announcement (and the related reservation of resources) is a very valuable information for a run-time system. It may allocate memory, prefetch data into the location, perform communication between distant processs, and whatever may be necessary to fulfill the request of the program. So in essence such feature helps the programmer to thoughtfully overrule distance and latency in a potentially unknown platform.

The two interfaces that we propose are `request` for future read-only `access`, and `require` for future read-write `acquires`. Combining non-corresponding calls (`request` and `acquire`, `access` and `require`) will not have the desired effect and should be avoided.

3.4 Controlling concurrency: ordering events

Since we propose a tool that should be applicable in a wide context and in particular in a distributed setting we may not assume that the processs share a global clock or other external resource for event synchronization.

All events (concerning DHO) are produced internally by each process by exactly one of the five functions `create`, `destroy`, `access`, `acquire`, and `release`. From the point of view of the process they can be considered as being atomic and guarantee that the considered operations are finalized upon return. Calls to `request` and to `require` do not constitute events for the calling process and are never blocking.

[2] `http://www.opengroup.org/onlinepubs/007908799/xsh/pthread_rwlock_rdlock.html`
[3] `http://www.opengroup.org/onlinepubs/009695399/functions/fcntl.html`
[4] See e.g `http://www.mpi-forum.org/docs/mpi-11-html/node44.html#Node44`

Data in turn knows about seven types of events which are not identical to the ones perceived by the processes. They are

- *link* and *unlink* events, corresponding to `create` and `destroy`,
- read-only and read-write *lock insertion* events, corresponding to placed calls `request` and `require`,
- read-only and read-write *locking* events, that correspond to an effective locking of the resource for the corresponding process, and
- `unlocking` events, corresponding to a call of `release`.

Observe that *locking* events do not directly correspond to calls of `access` or `require`. In fact such a *locking* event may happen any time between the `lock insertion` and the corresponding `access/require`. The return from the later calls (or the positive return of a `test`) can only be seen as a notification that the *locking* event had happened, but do not indicate *when* it happened. *Locking* events are guaranteed to happen in arrival order of the corresponding *lock insertion* events, the data is supposed to handle these events according to an event queue.

- If upon arrival of a *lock insertion* event for a specific handle a locking request is already present in the event queue of the corresponding data and the *locking* event has not yet taken place the old request is removed from the queue and the new one is appended at the end of the queue.
- If upon arrival of a *lock insertion* event for a specific handle the corresponding data is already locked for this handle the lock is released and the new request is appended at the end of the queue.
- If upon arrival of a read-only *lock insertion* event the locking queue for the corresponding data is empty and the data is not write-locked, the *locking* event takes place immediately.
- If upon arrival of a read-write *lock insertion* event the locking queue for the corresponding data is empty and the data is not locked (neither read-only nor write), the *locking* event takes place immediately.
- On reception of an *unlock* event, an eventual lock for the handle is released and an eventual locking request for the handle is removed from the event queue.
- At the end of treating any kind of event, the next locking requests in the queue that can be fulfilled are handled until a request is reached, for which the lock can not be obtained. In particular, a locking request that currently cannot be fulfilled blocks all other requests in the queue, even though they might feasible at that time.

As there is no global source of synchronization, events happening on different processors and on data may not be supposed to appear in a specific (total) order. The only assumption that an application may take is partial and related to control flow:

- For all calls on a particular handle that trigger an event on the corresponding data, the events arrive at the data in the same order as the calls are issued by the process.
- After return from `create` the corresponding *link* event for the data has either been registered or the call was erroneous and the handle is not valid.
- After a return from `destroy` any lock that the calling handle had held eventually is released with an *unlock* event and the *unlink* event has taken place.
- After a successful return from any of the event constituting functions (even for other data objects and handles) all *lock insertion* events have taken place.

From these informations, an application may only conclude by transitivity of this partial order relation that certain events on other processors have already taken place.

3.5 Gluing data together: ranges

Although similar at a first sight, the locking mechanisms of POSIX threads' read-write-locks and of mandatory file-locking with `fcntl` have quite different semantics concerning the scope to which they may apply. Whereas the first is a general tool to protect *one* unspecified resource, the second is more specific and dedicated to *ranges* of memory of one well defined object. We think that in our context these two features of mandatory file-locking, namely

- strict association to a well defined object, and
- the possibility of individually locking parts of the object by different processes

are important for cooperative computation concerning large amounts of data. By that different processes may e.g safely work on different parts of the same large data.

The interface `duplicate` creates a new handle that refers to the same object as given by the input parameter *other*. The parameters *offset* and *length* indicate the start and length of the desired part of the data to which the newly created handle refers. The newly created handle only refers to the "window" defined by *offset* and *length* and there is no possibility to access data outside that range via this object. Any calls to `access` or `require` will deliver pointers to memory that contains the data from position *offset* on and is guaranteed to include the next following *length* bytes.

In consistence with the `fcntl` interface a value of 0 for *length* has the special interpretation of defining a range going from *offset* to the end of the effective range that is accessible through `other`.

A call to `duplicate` creates a new handle of the same object as *other* refers to. This handle becomes completely independent from the *other*, in particular the new handle stays valid when *other* is destroyed.

3.6 Data persistence, integrity and awareness of references

When an initial handle to an object is created, the *name* that is passed to `create` defines the type of persistence of the object. If name is some valid name of a file or memory segment on the system the object is persistent and will survive the destruction of the last handle to it. DHO should use URLs to describe the data. Thereby it will be possible to re-use different types of protocols to access to the data, e.g "file" for file mapping, "shm" for shared segments, "http" for read-only data, "ftp" or "scp" for remote copies etc. If `name` is `NULL`, the object is a temporary (of length 0) that ceases existence when the last handle to it is destroyed.

The function `destroy` cuts all links between the *handle* and the object for which it was created. A call do `destroy` should succeed under all normal circumstances. If *handle* has a valid mapping (or announced some) the corresponding memory will first be unmapped before the new link to the *other* object is established.

If `create` or `duplicate` are called on a valid (i.e already created) *handle*, the first action will be to implicitly `destroy` *handle* and thus unmap, too, if necessary.

```
dho A(1024);              /// a tmp of size 1024
if (something) {
    dho B(A, 0, 0);       /// referencing whole
    B.resize(2048);       /// to many refs!
} else {
    dho B0(A, 0, 512);    /// first half
    {
      dho B1(A, 512, 0); /// second half
      A.destroy();        /// renouncing control!

      void* addr = B1.require();
      // Do something with addr.
    }
    B0.resize(2048);      /// bytes 512 to 2048 are
                          /// initialized to zero
}
```

Figure 2: An example for `resize`

Whilst a handle is the unique handle that refers to the data a process may call `resize` to assign a new length to the object. The semantics of the system interface `ftruncate`[5] apply, i.e:

- If the new length is smaller than previously the data beyond the new length is lost.

[5] `http://www.opengroup.org/onlinepubs/007908799/xsh/ftruncate.html`

- If the new length is larger than previously the newly created part will appear as filled with zero's.

This filling rule will also be applied if the object by itself previously extended beyond *offset+length*. Under no circumstances resize will provide access to parts of the data that were not accessible when the *handle* has been created.

The system will keep track of the handles that are alive for a given object and will be able to decide whether or not a given call to `resize` is valid. In the code of Figure 2, the "if" part shows an example to illustrate this. The "else" part shows the perhaps surprising feature that an extension of the handle may delete data.

If the uniqueness condition for `resize` is not fulfilled, `resize` should simply do nothing. The application may use `getLength` to know whether the `resize` succeeded.

Even if the handle points to non-persistent data, `getName` should return a valid name for the object. This name can e.g be used by other processes to create handles to the same object.

3.7 Run time errors and robustness

In general, programs that deviate from the desired flow of control between valid states as it is suggested by Figure 1 should not produce errors. The system should try to cope with such deviations as long as possible, instead. E.g the wrong pairings of announcements and mappings (i.e `request` followed by `acquire`, or `require` followed by `access`) could result in some run-time penalty because of badly used resources but not in an error condition of the program.

The only interface that is mandatory is `create`. Once destroyed, all memory of a handle about the data is lost and any call to the other interfaces of Figure 1 is an error. The application may test for such a situation with `getLength` which is non-zero otherwise.

4 Example

For the effect of the locking mechanism consider the code in Figure 3. It shows the loop of a matrix multiplication routine that uses a column block "`Bpart`" that is cyclically shifted around the processes. A more complete version of such a function is given in the appendix. For the notation: We realize the product $C = A \times B$, where `A`, `B` and `C` are $n \times n$ matrices. We have `p` processes, `mynum` is the number of the actual processor and for simplicity of the example `p` divides `n` and `cols` = `n`/`p`. Pointers `apoint` and `cpoint` hold addresses of the column block of `A` and `C`, `DO_MULT` is a sequential matrix multiplication routine.

It uses a pair of data handles `Bpart[]` which will be used alternated from iteration to iteration depending of whether the loop index `i` is even or odd. For this code it is easy to prove the properties that are summarized in the following lemma.

```
for (int i = 0; i<p; ++i) {
 Bpart[(i+1)%2].require();
 void const* bpoint = Bpart[i%2].acquire();
 DO_MULT(apoint, n, cols,
         bpoint, n, cols,
         cpoint, (mynum+i)%p);
 Bpart[i%2].release();
 Bpart[i%2].destroy();
 Bpart[i%2].create(
              B, len*((mynum+i+2)%p), len);
}
```

Figure 3: The `for`-loop of a matrix-multiplication

Lemma 1 *Suppose that the code from Figure 3 is executed by p processs such that each process* `mynum` *has inserted a lock on* `Bpart[0]` *for the range of* `len` *bytes starting at offset* `len`$\cdot$`mynum` *in* `B` *and that these locks are the only ones hold on* `B`*. Then at any moment of the* `for`*-loop the number of locks that are inserted or hold for any of these ranges are between one and two. In particular, on exit of the* `for`*-loop each processor has inserted exactly one lock for the handle* `Bpart[`$p \bmod 2$`]` *which corresponds to exactly the same range of* `B` *as was locked on entry.*

This shows, that when executed in a distributed environment this code will behave very similar as if it would be implemented with e.g MPI. The processs receive the blocks of `B` one after another and never hold more than two blocks at a time. On the other hand, executed on shared memory the processor just timely receive pointers to the data, no copy operation between the processors is taking place. So on both types of architectures such a code will execute efficiently.

5 Outlook

Implementation of a prototype

We think that a prototype for a DHO-library could be implemented very quickly on top of POSIX file descriptors (with `open` or `shm_open`, mandatory file locking and `mmap`) and MPI for communication and handling processes themselves. Apart from one subtle technicality, namely that the function `mmap` is page oriented and not byte oriented, we think that it will be in fact possible to extract the necessary tools from `SSCRAP`, the tool box for coarse grained parallelism that we developed in recent years, see [4, 5, 6].

Such a prototype should be already competitive when compared to implementations using only one of the other paradigms. Efficiency of shared memory should only depend

on the efficiency of the `shm_open` implementation but which is not yet satisfactory on all platforms. For message passing, the efficiency should be very good, due to the maturity of the MPI implementations for these platforms.

Extensions to mixed environments

Then an important step will be to test mixed environments, in particular clusters of multiprocessors, and to go to networks with heterogeneous latency between different parts.

References

[1] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[2] "The message passing interface." [Online]. Available: http://www.mpi-forum.org/

[3] "The open group." [Online]. Available: http://www.opengroup.org/

[4] M. Essaïdi, I. Guérin Lassous, and J. Gustedt, "SSCRAP: An environment for coarse grained algorithms," in *Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, 2002, pp. 398–403.

[5] J. Gustedt, "Towards realistic implementations of external memory algorithms using a coarse grained paradigm," in *International Conference on Computational Science and its Applications (ICCSA 2003), part II*, ser. LNCS, no. 2668. Springer, 2003, pp. 269–278.

[6] I. Guérin Lassous and J. Gustedt, "Portable list ranking: an experimental study," *ACM Journal of Experimental Algorithmics*, vol. 7, no. 7, 2002. [Online]. Available: http://www.jea.acm.org/2002/GuerinRanking/

[7] M. T. Goodrich, "Randomized fully-scalable BSP techniques for multi-searching and convex hull construction," in *Proceedings of the eighth annual ACM-SIAM Symposium on Discrete Algorithms*, M. Saks *et al.*, Eds. SIAM, Society of Industrial and Applied Mathematics, 1997, pp. 767–776.

[8] F. Dehne, A. Fabri, and A. Rau-Chaplin, "Scalable parallel computational geometry for coarse grained multicomputers," *International Journal on Computational Geometry*, vol. 6, no. 3, pp. 379–400, 1996.

[9] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas, "Towards efficiency and portability: Programming with the BSP model," in *8th Annual ACM symposium on Parallel Algorithms and Architectures (SPAA'96)*, 1996, pp. 1–12.

[10] A. V. Gerbessiotis and L. G. Valiant, "Direct bulk-synchronous parallel algorithms," *Journal of Parallel and Distributed Computing*, vol. 22, pp. 251–267, 1994.

Matrix Multiplication, complete example

```
// Supposes that each processor has placed a lock on B upon entry.  Will focus this
// lock while processing onto ranges of B that will circulate between the processors.
// The lock on B as a whole will then be placed again upon return form this function.
void matrix_mult( dho& A, dho& B, dho& C ) {
    off_t n = sqrt(A.getLength());
    off_t plen = A.getLength()/p;
    off_t cols = plen/n;
    off_t blen = plen/p;
    // Only input, prepare for reading
    dho Apart(A, len*mynum, len);
    Apart.request();
    // Output, prepare for writing.
    dho Cpart(C, len*mynum, len);
    Cpart.require();
    // Two column blocks, that will be circulating.
    dho Bpart[2];
    Bpart[0].create(B, len*mynum, len);
    Bpart[1].create(B, len*((mynum+1)%p), len);
    Bpart[0].require();
    // Get the lock for Apart and Cpart, and place the lock insertion for B[0].
    void const* apoint = Apart.access();
    void* cpoint = Cpart.acquire();
    // We know that our lock for B[0] is inserted, now synchronize on the latest
    // process that releases B.
    B.release();

    for (int i = 0; i<p; ++i) {
        // Everybody had been able to place their lock for Bpart[i%2].  We now may
        // insert the lock for B[(i+1)%p] without blocking process mynum+1.
        if (i==p-1) B.request();
        else Bpart[(i+1)%p].require();
        // Lock Bpart[i%2] and place the lock for Bpart[(i+1)%p]
        void const* bpoint = Bpart[i%2].acquire();
        DO_MULT(apoint, n, cols,
                bpoint, n, cols,
                cpoint, (mynum+i)%p);
        Bpart[i%2].release();
        Bpart[i%2].destroy();
        if (i<p-2) Bpart[i%2].create(B, len*((mynum+i+2)%p), len);
    }
    Cpart.release();
    Apart.release();
}
```

Proposed C-interface

```
/*  This may look like nonsense,
    but it is really -*- c -*- */


struct DHO_t;

/** construction and destruction */

int DHO_create(DHO_t* handle,
               char const* name);
int DHO_resize(DHO_t* handle,
               off_t length);
int DHO_duplicate(DHO_t* handle,
                  DHO_t const* old,
                  off_t offset,
                  off_t length);


int DHO_destroy(DHO_t* handle);


/** exclusive write access to the data **/
int DHO_require(DHO_t* handle);
void* DHO_acquire(DHO_t* handle);

/** inclusive read access to the data **/
int DHO_request(DHO_t* handle);
void const* DHO_access(DHO_t* handle);

/** test if data has arrived **/
int DHO_test(DHO_t const* handle);

/** release any data previously required,
    acquired, requested or accessed. **/
int DHO_release(DHO_t* handle);

/** state inquiry */
char const* DHO_getName(DHO_t const* handle);
off_t DHO_getLength(DHO_t const* handle);

/** system inquiry **/
off_t DHO_maxLength(void);
off_t DHO_minLength(void);
```

Proposed C++-interface

```
//  This may look like C code,
//  but it is really -*- c++ -*-
#include "DHO.h"

class dho : private DHO_t {
public:
    /// construction and destruction
    void dho(void);
    void create(char const* name);
    void dho(char const* name);
    void resize(off_t length);
    void dho(off_t length);
    void duplicate(dho const& other,
                   off_t offset=0,
                   off_t length=0);
    void dho(dho const& other,
             off_t offset=0,
             off_t length=0);
    void destroy(void);
    void ~dho(void);

    /// exclusive write access to the data
    void require(void);
    void* acquire(void);

    /// inclusive read access to the data
    void request(void);
    void const* access(void);

    /// test if data has arrived
    bool test(void)const;

    /// release any data previously required,
    /// acquired, requested or accessed.
    void release(void);

    /// state inquiry
    char const* getName(void)const;
    size_t getLength(void)const;

    /// system inquiry
    static size_t DHO_maxLength(void);
    static size_t DHO_minLength(void);
};
```

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399