Data Handover:
Reconciling Message Passing and Shared Memory

Jens Gustedt

LORIA & INRIA Lorraine

Abstract. *Data Handover* (DHO) is 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: (1) memory blowup and forced copies for message passing and (2) 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 inter-operation upon data.
Keywords: efficient data management, message passing, shared memory

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: (1) Message passing applied on shared memory architectures introduces a substantial memory blowup (compared to a direct implementation) and forces unnecessary copies of data buffers. (2) The simplified transposal of shared memory algorithms onto distributed platforms provoke a high complexity when it comes to guaranteeing 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, 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 Valiant [1990], 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. In the follow up of the BSP work it has been claimed and shown (theoretically and by large scale experiments) that applications that stick to certain rules concerning the granularity of the communication pattern will be able to overrule latency problems as they will be typically imposed by grid environments, see e.g Dehne [1999], Gebremedhin et al. [2002], Essaïdi et al. [2004].

With this paper we try for a proposal of a programming paradigm and interface (DHO) 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 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.

The field of applications to which DHO may be applied is probably somewhat broader than what falls under the rules of coarse grained computing as mentioned above. Only, control must be neglectable compared to local computation and communication, i.e programs must organize communication in substantial large units such that the local computation together with the transfer of the data dominates the number of control transfers by orders of magnitude. Applications for which control dominates the resource consumption are not suited for DHO, and, as we think, for grid computing in general.

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) and of the extensions that have been proposed so far. Then, in Section 3, we will propose the main features of our interface. A short example of the central locking and mapping feature is then given in Section 4 followed by a brief discussion of possible steps for an implementation in Section 5. In an appendix, we give a more detailed 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[1]and POSIX shared memory *segments*[2]. 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 The two base paradigms and actual extensions

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 the most important for us.

Simplified Data Control Data control within message passing environments is conceptually simple. The programmer controls the buffers that hold the data and has precise synchronization points after which he may assume that the data has been successfully transmitted. The programmer completely controls the consistency of the data.

[1] Message Passing Interface, see `http://www.mpi-forum.org/`

[2] See "The Opengroup", `http://www.opengroup.org/`. Shared memory segments are not to be confounded with the POSIX THREAD interface, `pthread_*`.

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 optimizes existing architectures in 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 blowup: 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, even if the sender perfectly knows where the data should end 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 two problems, data consistency and latency. But they are also inherently present in the design of shared memory architectures themselves. Data consistency problems contribute much to the difficulties of the interaction between OS and hardware (cache coherence, TLBs,...). A low latency interconnection bus contributes a lot to the cost 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 requires at 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.

2.3 Extensions of the Base Paradigms in Related Work

Numerous proposals have been made to extend each of the two paradigms (and their most prominent realizations) to the respective other setting. We only will be able to name a few of them and will not be able to present them according to their merits.

Distributed Shared Memory[3] systems (DSM) extend the shared memory paradigm into a distributed setting in that they provide an intermediate layer to the application such that memory access is handled transparently in one (emulated) address space. Close to our paradigm are *Object-based Software DSM*, i.e DSM implementations that explicitly expose the DSM concept through the manipulation of shared objects. Probably most of these systems (if they would be suitable for grid computing) could be used to implement the API of this paper relatively easy. DHO uses a data consistency strategy that is similar to *lazy release consistency*, see Keleher et al. [1992], and boroughs part of its interface names from there. It uses ideas similar to Midway[4] to use lock acquisition message for the propagation of knowledge about event ordering in the system. Other systems, a lot that are historical others under current development, can be classified as *Page-based* DSM, *Hardware-based* DSM or as *Single System Image* (SSI) operating systems[5]

Generally, these distributed shared memory systems impose a very close cooperation between the processes, are intrusive with respect to the hosting system and are optimized on a fine grained level of control. These properties make them very effective in physically close, homogeneous clusters of machines that are within the same administrative domain. We think that it is unlikely that this will scale to the opaque, heterogeneous and bandwidth oriented world of the grid.

For the message passing paradigm, the extension from MPI v. 1.2 to MPI 2[6] includes so-called *one sided communication*, i.e communication that may be initiated from one side only, `MPI_Get` and `MPI_Put`. They allow for *Remote Memory Access*, RMA. This is a technique that is similar to DSM with the difference that it doesn't introduce a shared address space for the processes but mutually maps parts of the process memories to each other. MPI 2 leaves the choice upon which parts of memory should be coupled, how long such a coupling shall last and the consistency model almost entirely to the program designer. By that it is more flexible and better adapted to couple processes that are hosted by distant processors with potentially different operating systems. On the other hand this flexibility and openness present a major design difficulty, as well for an application programmer as for a designer of an MPI 2 environment. Perhaps for many of the potential users and implementers the benefits that this paradigm offers have not outperformed these difficulties: usage of the features of MPI 2 is minor compared to v. 1.2 and other than for MPI v. 1.2 a complete reference implementation of MPI 2 in the follow up of MPICH 1.2 took several years and has only been completed recently[7].

3 The proposal of an interface

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.

But unfortunately in the message passing paradigm 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 access to an abstract object, the *data*, and to let handle the library the allocation of memory itself. Thereby we may ensure that the association between the data and its instance is always maintained.

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

[3] see Distributed Shared Memory Home Pages, `http://www.ics.uci.edu/~javid/dsm.html`

[4] `http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/project/midway/WWW/HomePage.html`

[5] see e.g `http://www.kerrighed.org/`

[6] `http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html`

[7] `http://www-unix.mcs.anl.gov/mpi/mpich2/`

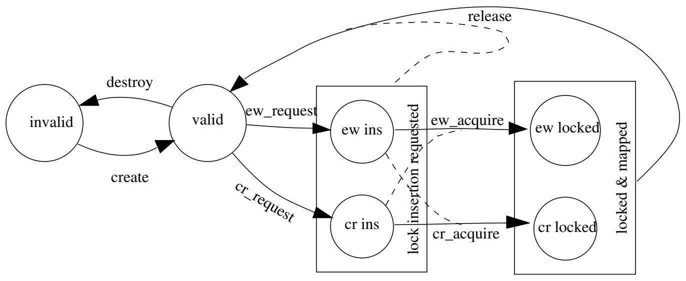

Fig. 1. Essential part of the state diagram of a handle. Less common operations are dashed. Omitted are operations that not encouraged such as direct locking or destructions from other states than “valid”.

name	penalty	returns	description	similar in MPI		similar in POSIX
				sender	receiver	
`DHO_t`			data type representing the user data	buffers and `MPI_Request`		file descriptor
`DHO_create`	none		creation of a handle	buffer creation		{`shm_`}`open`
`DHO_resize`	none		resizing	buffer resizing		`ftruncate`
`DHO_duplicate`	none		duplicate a handle			
`DHO_destroy`	flush		destruction of a handle	buffer deletion		`close` and {`shm_`}`unlink`
`DHO_ew_request`	prefetch		request future exclusive write access, write-prefetch the data		`MPI_Irecv`	
`DHO_ew_acquire`	blocks, insert	`void*`	instantiate data in memory		`MPI_Wait`	`fcntl` for write-locking and `mmap`
`DHO_cr_request`	prefetch		request future concurred (shared) read access, read-prefetch the data		`MPI_Irecv`	
`DHO_cr_acquire`	blocks, insert	`void const*`	instantiate data in memory		`MPI_Wait`	`fcntl` for read-locking and `mmap`
`DHO_release`	flush		relinquish access	`MPI_Isend`		`munmap` and `fcntl` for unlocking
`DHO_test`	none	`int`	test for available data in memory		`MPI_Test`	`fcntl` with try-locking
`DHO_getLength`	none	`size_t`	check size and/or validity			`fstat` for field `st_size`
`DHO_getName`	none	`char const*`	get URI of underlying object			

Table 1. The principal C interfaces. In the column *penalty* a “blocks” indicates that the calling process might block until other processes release the data. A “prefetch” or “flush” indicates that although the process will not be blocked that the system might be busy in doing preparative reading or clean-up writing. Generally the run time penalty may correspond proportionally to the length of the data that is handled. An “insert” indicates that although the process might not be blocked on locks that are placed by others, that it might block until *all* its previously requested lock insertions (`ew_request` or `cr_request`) or link and unlink events (`create`, `duplicate` or `destroy`) are known to be taken into account. The penalty of “insert” should be bound to a factor of the communication latency.
If there is no indication of a return value, the function generally returns an error code. The C++-interface should throw an exception instead.

`destroy`), to gain *Exclusive Write* access to all or part of the data (`ew_request` and `ew_acquire`), to gain *Concurrent Read* access (`cr_request` and `cr_acquire`) and to resign from accessing the data (`release`). The interface *does not* describe a DSM in that it gives no false illusion of presenting the data in a persistent location in address space, not even when restricted to a single process. It is neither message passing, since it never looses track of the conceptual identity of the data when accessed in different processes.

The entire interface as well as an analogous interface for C++ are given in the appendix. The individual parts of the interface are discussed in the following sections.

3.1 Abstracting from memory: data handles and mappings

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), 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.

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 `cr_acquire` (for concurrent (or shared) read mappings) and `ew_acquire` (for exclusive write mappings).

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

Figure 1 describes 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.

The functions `*_acquire` shall return pointers to memory of the necessary size which will be properly initialized with the data. In case of concurrent read mappings the pointer that is return will bear the `const` attribute to hinder the application in writing into the corresponding location. After the application has used the pointer that was returned by `*_acquire` it may call `release` to release the mapping and to free resources that such a mapping might bind.

The objects that DHO should be able to handle may be very big. In fact the totality of all objects, might not fit into the address space of a single processor. So address space by itself might be a scarce resource and DHO has to allow for a re-use of addresses once a mapping has been released. Therefore the pointer that had been returned by `*_acquire` has to be considered invalid after a call to `release` and an application should never keep that pointer beyond an `*_acquire/release` cycle. If the application issues another call `*_acquire` for the same data, the pointer that is returned then may be completely different from the previous one. If released, the data behind a handle itself becomes inaccessible through that handle but shall never be lost as long as any handle on it is alive.

3.2 Separating access: locking

To ensure data consistency, the mappings as introduced above follow the semantics of read-write locks, similar to POSIX' read-write-locks (`pthread_rwlock_*`[8]) and to advisory file-locking with

[8] `http://www.opengroup.org/onlinepubs/007908799/xsh/pthread_rwlock_rdlock.html`

`fcntl`[9]. That is, as long as there is a handle that holds a shared read lock for some data (or some *range* of it, see below), no other handle can acquire an exclusive write lock on it, and as long there is a handle that holds an exclusive write lock no other handle can gain any type of lock on that same data. The programmer may resign from the previously gained lock (and from access to the data) by means of `release`. Thereafter other handles might then gain the lock and access the data.

Calls to `*_acquire` will always block the calling process until the lock is obtained or an unrecoverable error occurred for the data. After a first return from such an `*_acquire` call, as long as there will be no other `DHO_*` call, newly issued calls to the same `*_acquire` function shall return immediately and provide the same return value as the original call. Thereby, handles may effectively be shared between different threads of the same process without wasting resources for renewed locking and mapping attempts.

Calls to `release` shall never block. For interfaces that conditionally lock a handle without blocking see Section 3.4 below.

3.3 Gluing data together: ranges

Although similar at a first sight, the locking mechanisms of POSIX threads' read-write-locks and of advisory 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 the following two features of file-locking are important for cooperative computation concerning large amounts of data, namely (1) strict association to an object, and (2) the possibility of individually locking parts of the object by different processes. By that different processes may safely work on different parts of the same large data object.

A call to `duplicate` creates a new handle for 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. The parameters *offset* and *length* indicate the start and length (in bytes) 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 `*_acquire` will deliver pointers to memory that contains the data from byte 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`.

3.4 Forethought: requesting future access

Another advantage of the message passing paradigm is its asynchronicity and in particular that it allows for so-called non-blocking receives. By that it is possible to distinguish the announcement of a future take of control (`MPI_Irecv`) from the effective instantiation of the data in memory (`MPI_Wait`)[10]. 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 processes, 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 idea of DHO is to systematically invite the programmer to provide such announcements and thereby to increase her or his awareness for the non-triviality of the environment for which the program is designed.

[9] `http://www.opengroup.org/onlinepubs/009695399/functions/fcntl.html`

[10] See e.g `http://www.mpi-forum.org/docs/mpi-11-html/node44.html#Node44`

The two interfaces that we propose are `cr_request` for future `cr_acquire`, and `ew_request` for future `ew_acquire`. Combining non-corresponding calls (`cr_request` and `ew_acquire`, `ew_request` and `cr_acquire`) will not have the desired effect and should be avoided, see below. After a `*_request` has successfully been placed, the presence of the requested mapping and lock can be tested with `test`. Such a call will never be blocking. After an affirmative answer of `test`, the pointer to the data may then be obtained by a call to the corresponding `*_acquire`, also without blocking the caller.

3.5 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 processes share a global clock or other external resource for event synchronization. On the other hand, the order in which processes treat data is important, and so we need a minimal event model that helps the programmer to establish an order among the data access that he designs. Again, this model intends to claim the obvious combined with the most possible slackness for the system: (1) The ordering of events as perceived by any individual processor is conserved by the system when handling the data. (2) The effective ordering of the events upon the data may be any ordering that observes the constraints that are imposed by the ordering upon the processors.

All events (concerning a handle) are produced internally by each process by exactly one of the five functions `create`, `destroy`, `cr_acquire`, `ew_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 `cr_request` and to `ew_request` do not constitute events for the calling process and are never blocking.

Data in turn knows about seven types of events which are not identical to the ones perceived by the processes. They are (1) *link* and *unlink* events, corresponding to `create` and `destroy`, (2) concurrent read and exclusive write *lock insertion* events, corresponding to `cr_request` and `ew_request`, (3) concurrent read and exclusive write *locking* events, that correspond to an effective locking of the resource for the corresponding handle, and (4) *unlocking* events, corresponding to `release`.

To allow for asynchronicity, *locking* events do not directly correspond to calls of `*_acquire`. In fact they are *internal* events that may not directly be triggered by an external source. Such a *locking* event may happen any time between the `lock insertion` and the successful return from the `*_acquire` call (or a `test`).

Locking events are guaranteed to respect the arrival order of the corresponding *lock insertion* events, the data is supposed to handle these events according to an event queue. There may be conflicts in the set of requests that are already locked (but not yet unlocked) and those that are in the queue. We say that a request R *directly blocks* another request S if (1) the ranges of the two request intersect, (2) R was placed before S and (3) at least one of the two requests is for exclusive write.

It is easy to see that this relation defines an acyclic graph and (by transitivity) a partial order of events. We say that R_0 is *blocking* R_k if there is a chain of requests $R_1, \ldots, R_{k-1}$ such that R_{i-1} directly blocks R_i for all i.

After the handling of a particular request R is finished, the queue of pending lock requests is checked for requests that can be served now. This processing has to give two guarantees:

consistency: A locking requests in the queue may be served if there are no earlier requests that block it.

progress: A request that is now first in the event queue and that is not blocked by any lock shall always be served.

In other words, all requests that are minimal in the blocking order *may* be served, and the one among them that is also minimal with respect to the arrival order *must* be served if possible.

3.6 Data persistence, integrity and awareness of references

To allow for persistence and access from any unspecified environment all data shall be identified by an URI (plus an eventual *fragment identifier*) as of RFC 2396[11], its *name*. This name can either be given explicitly as an argument to `create` or, if left `NULL`, will be generated by the system. The *name* that is passed to `create` defines the type of persistence of the object. If name refers to some valid file or memory segment on the system the object is persistent and will survive the destruction of the last handle to it. If the choice of `name` was left to the system, the object is a *temporary* (initially of length 0) that ceases existence when the last handle to it is destroyed.

The use of URLs (as a subset of URIs) makes it possible to re-use pre-existing protocols to access the data, e.g "file" for file mapping, "shm" for shared segments, "http" for read-only data, "ftp" or "scp" for remote copies etc. and in addition allows for a future integration of new protocols. Using URIs (and not only URLs) make it possible to refer to objects (resources) regardless of the protocol to access them, e.g an URI like "urn:ISBN:0-300-02495-9" could refer to a resource for some text processing program using DHO. Clearly, not all possible URIs make sense in the context of DHO, and implementations of DHO may also differ much in what kind of URIs they accept. The function `getName` shall return a string containing the URI. If the handle refers to a sub-range of the object this name shall be followed by a *fragment identifier*, of the form "#offset,length" where *offset* and *length* are the decimal values of the absolute byte position in the object. If the object is temporary and the length has not be `resize`d to a non-zero value (see below), `getName` shall return a `NULL` pointer. This is because such an object can not be referenced by two different handles and be `resize`d simultaneously in a consistent way.

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 `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.

Whilst a handle is the unique handle that refers to the whole object (e.g because it was just freshly created) a process may call `resize` to assign it a new length. The semantics of the system interface `ftruncate`[12] apply, i.e: (1) If the call shortens the object the data beyond the new length is definitively lost. (2) If the call extends the object the newly created part will appear as filled with zero's. 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. If the conditions for `resize` (uniqueness and referring to the whole object) are not fulfilled, `resize` should simply do nothing. The application may use `getLength` to know whether the `resize` succeeded.

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 `cr_request` followed by `ew_acquire`, or `ew_request` followed by `cr_acquire`) could result in some run-time penalty because of badly used resources and because of re-insertions of locking requests but not in an error condition of the program.

A handle is erroneous iff `getLength` returns zero. This may e.g be the case when the object is empty after it was created by the call to `create` and no resize action has yet been undertaken or if a previously created handle had been `destroy`ed. Observe that no valid handles that refer to a sub-range of size zero can be created. Reclaiming a parameter `length` of zero is interpreted as defining a valid range starting at `offset` and extending to the end of the range of the "parent" handle.

[11] `http://www.ietf.org/rfc/rfc2396.txt`

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

4 Example

Due to space limitations we are not able to give examples for all aspects of DHO. For the effect of the combined locking and mapping mechanism that takes an important part in DHO consider the code in Figure 2. It shows the loop of a matrix multiplication routine that uses a column block "Bpart" that is cyclically shifted around the processes.[13] A more complete version of such a function is given in the appendix.

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. When executed in a distributed environment this code will behave very similar as if it would be implemented with e.g MPI in replacing request with MPI_Irecv, ew_acquire with MPI_Wait, and destroy with MPI_Isend. The processes 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.

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

Fig. 2. The for-loop of a matrix-multiplication

5 Implementation

The proposal of DHO is based upon the experience with our prototype library for coarse grained parallel algorithms, SSCRAP[14]. It already successfully implements parts of the components of DHO. In particular the mapping/unmapping technique (called chunks in SSCRAP) has shown to be a valuable tool for developing parallel programs that leave enough slackness to the memory management system. By that SSCRAP programs run efficiently without modification on large mainframes and clusters. In an out-of-core context, we are able to handle data that is larger than the address space of the machine. The run time control of events (based on BSP-like supersteps) is stricter than what we imaging for DHO, though. But this should not be a major issue in a future extension, since the model that we propose here (as being a variant of lazy release consistency) is well experimented in the DSM context and well mastered on a system level by the means of advisory file locking.

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, advisory file locking and mmap) and MPI for communication and handling processes themselves. Efficiency of shared memory should only depend on the efficiency of the shm_open implementation. This is already quite good e.g on Linux systems, but not yet satisfactory on other platforms that sometimes implement POSIX norms only verbally. For message passing, the efficiency should be easier to achieve due to the maturity of the MPI implementations for all platforms. There are some technicalities that remain to be solved: (1) The function mmap is page oriented and not byte oriented. (2) Advisory file locking is process-oriented and not well suited for objects that lock. (3) The POSIX standard for advisory file locking does not impose an order in which locks are obtained.

[13] For the notation: We realize the product $C = AB$, where A, B and C are nn 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.

[14] http://www.loria.fr/~gustedt/sscrap/

We think that these restrictions can be overcome relatively easy and that it might be in fact possible to extract much of the necessary from `SSCRAP`. On the other hand we are hoping that the present work will provide us with more valuable feedback and we will be able to combine energies in a community effort to implement a prototype of a DHO or similar system.

Bibliography

F. Dehne. Coarse grained parallel algorithms. *Algorithmica Special Issue on "Coarse grained parallel algorithms"*, 24(3/4):173–176, 1999.

F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3):379–400, 1996.

Mohamed Essaïdi, Isabelle Guérin Lassous, and Jens Gustedt. SSCRAP: An environment for coarse grained algorithms. In *Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, pages 398–403, 2002.

Mohamed Essaïdi, Isabelle Guérin Lassous, and Jens Gustedt. SSCRAP: Soft synchronized computing in rounds for adequate parallelization. Rapport de recherche, INRIA, May 2004. URL `http://www.inria.fr/rrrt/rr-5184.html`.

Assefaw Hadish Gebremedhin, Isabelle Guérin Lassous, Jens Gustedt, and Jan Arne Telle. PRO: a model for parallel resource-optimal computation. In *16th Annual International Symposium on High Performance Computing Systems and Applications*, pages 106–113. IEEE, The Institute of Electrical and Electronics Engineers, 2002.

A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Int'l Symp. on Computer Architecture (ISCA'90)*, pages 15–26, 1990. URL `citeseer.lcs.mit.edu/gharachorloo90memory.html`.

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)*, pages 1–12, 1996.

Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, 1992. URL `citeseer.lcs.mit.edu/keleher92lazy.html`.

Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

Appendix A Matrix Multiplication, complete example

```
// Supposes that each processor has inserted a cr locking request on B
// upon entry.  Will focus this lock ew 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, plen*mynum, plen);
    Apart.cr_request();
    // Output, prepare for writing.
    dho Cpart(C, plen*mynum, plen);
    Cpart.ew_request();
    // Two column blocks, that will be circulating.
    dho Bpart[2];
    Bpart[mynum%2].duplicate(B, plen*mynum, plen);
    Bpart[(mynum+1)%2].duplicate(B, plen*((mynum+1)%p), plen);
    Bpart[mynum%2].ew_request();
    // Get the lock for Apart and Cpart, and place the lock insertion
    // for B[0].
    void const* apoint = Apart.cr_acquire();
    void* cpoint = Cpart.ew_acquire();
    // We know that our lock for B[mynum%2] is inserted, now synchronize on
    // the latest process that releases B.
    B.release();

    for (int i = mynum; i< mynum+p; ++i) {
        // Everybody had been able to place their lock for Bpart[i%2].
        // We now may insert the exclusive locking request for
        // B[(i+1)%2] without blocking process (mynum-1)%p.
        if (i < mynum+p-1) Bpart[(i+1)%2].ew_request();
        else B.cr_request();
        // Lock Bpart[i%2] and place the locking request for
        // Bpart[(i+1)%2]
        void const* bpoint = Bpart[i%2].ew_acquire();
        DO_MULT(apoint, n, cols,
                bpoint, n, cols,
                cpoint, i%p);
        if (i < mynum+p-2) Bpart[(i+2)%2].duplicate( B, len*((i+2)%p), len);
    }
}

int main(int argc, char* argv) {
    // Three different types of uri to refer to the data.
    dho A("https://top.secret/A.dat");
    dho B("ssf://toto.my.company/B.dat");
    dho C("file://tutu.my.company/C.dat");
    B.cr_request();
    matrix_mult(A, B, C);
}
```

Appendix B 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[MAXNAME]);
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_ew_request(DHO_t* handle);
void* DHO_ew_acquire(DHO_t* handle);

/** concurrent read access to the data **/
int DHO_cr_request(DHO_t* handle);
void const* DHO_cr_acquire(DHO_t* handle);

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

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

/** state inquiry */
DHO_getName(DHO_t const* handle,
            char const name[MAXNAME]);
off_t DHO_getLength(DHO_t const* handle);

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

Appendix C 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[MAXNAME]);
    void dho(char const name[MAXNAME]);
    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 ew_request(void);
    void* ew_acquire(void);

    /// concurrent read access to the data
    void cr_request(void);
    void const* cr_acquire(void);

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

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

    /// state inquiry
    void getName(char const[MAXNAME])const;

    size_t getLength(void)const;

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