Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2013

Service-Oriented Programming in MPI

Sarwar ALAM [1], Humaira KAMAL and Alan WAGNER

Department of Computer Science, University of British Columbia, Canada

Abstract. In this paper we introduce a service-oriented approach to the design of distributed data structures for MPI. Using this approach we present the design of an ordered linked-list structure. The implementation relies on Fine-Grain MPI (FG-MPI) and its support for exposing fine-grain concurrency. We describe the implementation of the service and show how to compose and map it onto a cluster. We experiment with the service to show how its behaviour can be adjusted to match the application and the underlying characteristics of the machine.

The advantage of a service-oriented approach is that it enforces low coupling between components with higher cohesion inside components. As a service, the ordered linked-list structure can be easily composed with application code and more generally it is an illustration of how complex data structures can be added to message-passing libraries and languages.

Keywords. service-oriented programming, function-level parallelism, distributed data structures, linked list, MPI-aware scheduler, MPI runtime, FG-MPI, MPICH2

Introduction

Process-oriented languages in general, and communication libraries like MPI in particular, do not have the rich collections of data structures present in today's object-oriented programming languages. The lack of data structure libraries for MPI makes it necessary for programmers to create their own and adds to the overall programming complexity of using message-passing languages. The difficulty in having to explicitly manage the distribution of the data is often pointed to as a major impediment to the use of message passing [1]. One approach that is at the heart of process-oriented systems [2,3] and many distributed systems [4,5] is to implement the data structure using its own collection of processes and provide the distributed data structure as a service. We call this service-oriented programming. A service-oriented approach enforces low coupling, because components interact only through message-passing and cannot directly access one another's data, and leads to higher cohesion inside components, because application and data structure management code no longer needs to be combined together. We investigate the use of a process-oriented design methodology together with a service-oriented programming approach (design pattern) to provide highly scalable, easy to use libraries for complex data structures.

Our interest in a service-oriented approach to programming arose in the context of our work on Fine-Grain MPI (FG-MPI) [6,7]. FG-MPI extends MPICH2 by adding an extra level of concurrency (interleaved concurrency) whereby every OS process comprises of several MPI processes, rather than just one. FG-MPI uses a coroutine-based non-preemptive threading model together with a user-level scheduler integrated into the middleware to make it possible to support thousands of processes inside an OS process to expose finer-grain concurrency with no change to the programming model. The support for having many light-weight

[1]Corresponding Author: *Sarwar Alam, Department of Computer Science, University of British Columbia, Vancouver, B.C. V6T 1Z4, Canada*; E-mail: `sarwar@cs.ubc.ca`.

MPI processes and the flexibility to map processes to functions[1] to execute concurrently inside an OS process, as well as in parallel across the cores and machines in a cluster makes it possible to consider a service-oriented approach to providing a distributed data structure. In this paper we use FG-MPI and our service-oriented approach to implement an ordered linked-list service.

We chose an ordered linked list because it is a common data structure, its linear structure makes it challenging to parallelise in an efficient way, and it highlights how FG-MPI makes it possible to design new types of MPI programs. Since most MPI implementations bind an MPI process to an OS process, one could implement a linked list as a collection of separate OS processes. Previous work by Iancu *et al.* [8] has shown that over-subscription of MPI processes to cores often leads to poor performance and because of this the usual practice is to execute one MPI process per core. This directly binds the number of MPI processes to the number of cores which results in the design of processes to match the machine rather than the available concurrency present in the algorithms. This leads to more coupled, less cohesive programs.

A second drawback to the previous approach is that when executing with one MPI process per core it is difficult to keep the application and service processes constantly busy. One common approach to avoid this problem is to use MPI in combination with Pthreads, or some other runtime, to combine the MPI application process with a thread implementing the concurrent data structure. This provides a fine-grain implementation, but at the expense of portability and predictability because the multiple models and runtimes interact in unpredictable ways. Other than MPI, there are concurrent data structures and APIs designed for multicore, like OpenMP, that can be used, but they do not scale to execute on clusters or efficiently scale to the hundreds and thousands of processor cores which is the intent of our design. Our approach to the design of an ordered linked-list service relies on FG-MPI and avoids the previously mentioned problems to provide a scalable data structure service to applications.

In this paper we give the design, implementation and evaluation of an ordered linked list data structure in FG-MPI. We make the following contributions:

- A demonstration of how the FG-MPI extensions to MPI makes it possible to follow, within MPI, a process-oriented design methodology where the resulting process structure reflects the problem rather than the machine. This makes it possible to add complex data structures to MPI by using a service-oriented approach.
- We provide a novel design of a pure message-passing implementation of an ordered linked list targeted to the type of multicore clusters that are common to high-performance and cloud computing environments.
- We introduce a variety of performance-tuning parameters and investigate the effect of these parameters with regards to tuning the service to the underlying characteristics of the machine and network to achieve good performance across varied machine sizes and communication fabrics.
- Unlike many MPI programs, our mapping of processes to OS processes on a multicore machine, makes it possible to expose a large amount of concurrency that can be predictably scheduled. We also introduce a free process sub-service to allocate and deallocate processes to dynamically adjust to varying loads and demands.
- We introduce shortcuts, a novel technique related to service discovery and relate this to trade-offs between data consistency and performance.

In Section 1 we briefly describe related work and how our approach differs from existing work in the area. A discussion of service-oriented programming and the support for it within the FG-MPI runtime is given in Section 2. In Section 3 we describe the design of the ordered

[1]C functions, named procedures

linked-list service, describing the implementation of the operations, trade-offs with respect to different consistency properties, and the introduction of parameters related to bounded asynchrony, trade-offs between parallelism and interleaved concurrency, adjustments to the granularity, and the mapping of the service onto the system. Shortcuts are discussed in Section 3.6. In Section 4 we describe how the service is mapped and composed with the application. In Section 5 we experimentally show how these parameters affect the behaviour of the service. Conclusions are given in Section 6.

1. Related Work

The focus of MPI is on performance and as a result it supports a wide variety of communication routines to match modern network protocols, interfaces and fabrics. In contrast to synchronous message-passing languages like occam-π [9], MPI programs make frequent use of non-blocking asynchronous communication to overlap computation and communication. The message latency between machines in a cluster is relatively high in comparison to on-chip memory-based messaging and this "send and forget" model of messaging makes it possible to off-load message delivery to the network interfaces and devices. The intent is to increase the network load to better utilise the network hardware and have more messages in flight to provide more work for each machine, thereby improving performance by mitigating the overhead of communication. As shown by Valiant [10], the amount of work needed per machine to overlap with communication depends on the latency and bandwidth characteristics of the network. This suggests that, for scalability, the messaging needs to be able to adjust to the characteristics of the network in much the same way that programs need to take into account the memory hierarchy. The tuning parameters introduced in our design come from a two-level view of the network and the need to adjust the messaging to the fast local and slower non-local communication mechanisms in a cluster of multicore machines. We view this as essential to the design of programs to scale to hundreds and thousands of processors and differentiates our work from languages and systems that execute on a single multicore machine.

A linked list is a simple well-known data structure that has been frequently implemented as a concurrent data structure [11,12,13]. We make use of some of the ideas in [13] (Chapter 9) for the design and correctness of our linked list. In particular the notion of atomic actions, hand-over-hand[2] and forward-only traversal of the list have their counterparts in our design, however, the design is significantly different in that the list elements are active processes where the data structure has control over the operations, not the application processes as in the case of concurrent data structures.

In cloud computing environments hash-based storage structures like Cassandra [14] (and key-values stores) are commonly used to allow for distributed access to the data. Although these systems are based on message passing they are optimised for coarse-grain access to large data. Our distributed ordered linked list provides similar types of access to data as Cassandra but with more of a focus on scalability and low-latency where we are considering finer-grain in-memory access to data rather than large blocks of data residing on disk. As well, as expected, distributed storage systems are designed for availability and reliability, which we do not consider. A notable advantage of our ordered linked-list structure is that because keys remain sorted it would be relatively easy to extend our operations to support range queries, which is inefficient on simple hash-based structures. More generally in Peer-to-Peer computing Distributed Hash Tables (DHT) provide a pure message-based implementation of a look-up service designed for Internet applications [15]. The primary focus of these

[2]The lock for the next item in the list is acquired before releasing the lock for the current list item.

systems is on scalability and availability, particularly in the case where processes in the DHT frequently leave and join (i.e., churn), which is substantially different from our use case.

2. Service-Oriented Programming

Our objective is to take advantage of FG-MPI's support for fine-grain processes to implement an ordered linked-list service. The service consists of a collection of processes with, in its finest granularity, one list node per process along with a message interface for applications to communicate with the service. In the following we give a brief description of FG-MPI and discuss the advantages of providing distributed data structures as a service.

An FG-MPI execution, $[P, O, M]$, can be described in terms of P the number of MPI processes per OS process[3], O, the number of OS processes per machine and M, the number of machines. A typical MPI execution is of the form $[1, O, M]$ where N, the total number of MPI processes as given by the "-n" flag of MPI's `mpiexec` command, equals $O \times M$. In FG-MPI we added an "-nfg" flag to `mpiexec` enabling one to specify $P > 1$, where $N = P \times O \times M$.[4] Figure 1 shows a $[4, 2, 2]$ mapping started with the command:

```
mpiexec -nfg 4 -n 4.
```

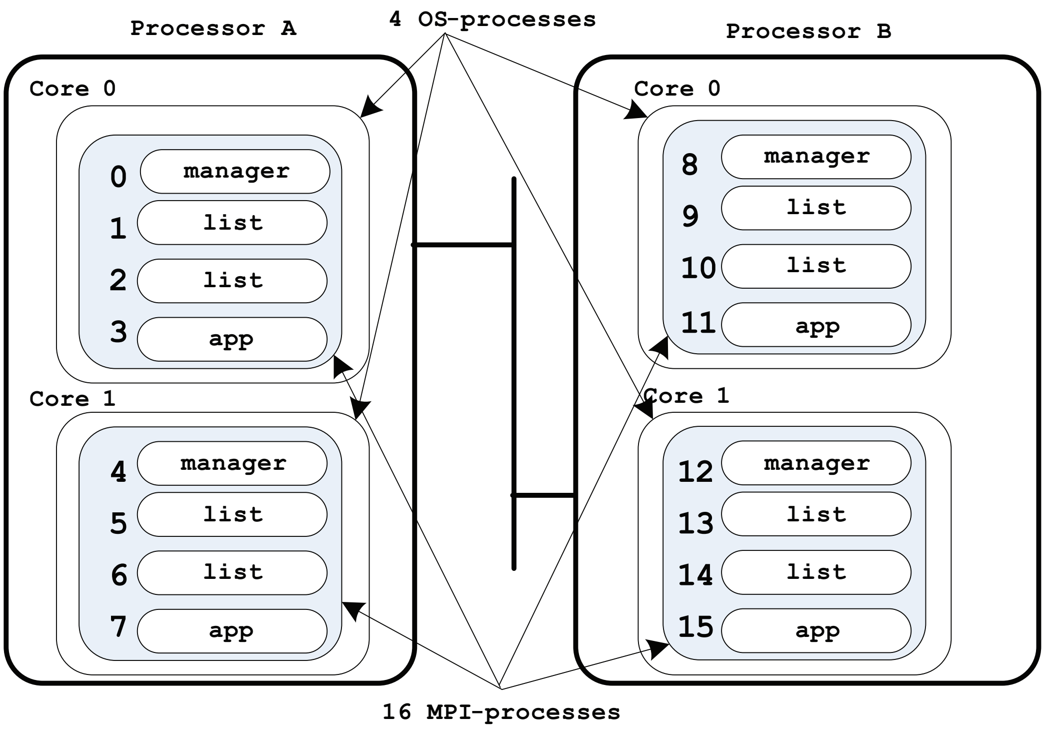


Figure 1. A cluster consisting of 2 machines each with 2 cores and a $[4, 2, 2]$ mapping with one manager, two list nodes, and an application process per OS process and 16 MPI processes in total. For the linked-list service, more list node processes will be added per core by increasing the `nfg` count (currently 4) in the `mpiexec` command.

The P co-located processes are full-fledged MPI processes that share the middleware inside the single address space of an OS process. FG-MPI uses coroutines to implement MPI

[3]We refer to these P MPI processes as *co-located* processes sharing a single address space. The terms *process*, *fine-grain process* and *MPI process* are used interchangeably. The term "OS process" is used to refer to an operating-system process.

[4]Both "`-n`" and "`-nfg`" can be used with `mpiexec` colon notation to create mappings with differing P's, O's and M's for each executable.

processes as non-preemptive threads with an MPI-aware user-level scheduler. The scheduler works in conjunction with the MPI progress engine to interleave the execution of the MPI processes [6]. Each MPI call provides a potential descheduling point for one co-located process to cooperatively yield control to another process. Each process, upon making an MPI call, will try and progress its own request[5], as far as possible, as well as any outstanding requests posted by other co-located processes. If it is unable to complete its call then it yields control to the scheduler, which selects another process to run. FG-MPI enables expression of function-level parallelism due to its fast context-switching time, low communication and synchronisation overhead and the ability to support thousands of MPI processes inside a single OS process [7]. This allows us to map MPI processes to functions instead of `main` programs (see Section 4). In summary, FG-MPI is integrated into the MPICH2 middleware and extends its execution model to support interleaved concurrency of finer-grain MPI processes inside an OS process.

In MPI programmers can, and frequently do, use non-blocking communication to switch between code segments to introduce "slackness" to overlap computation with communication or ensure a better computation to communication ratio. In essence, programmers are doing their own scheduling inside their program which adds to the complexity of the program and all but destroys program cohesion [6]. In FG-MPI programmers can take advantage of the integrated runtime scheduler to decompose their program into smaller processes that better reflect the program rather than code for optimising the messaging. In particular, for distributed data structures FG-MPI makes it possible to separate the data structure code from that of the application using that data structure.

As previously stated, we implement our distributed ordered-list structure as a distributed system service. We distinguish this type of library from the usual type of MPI library where the library computation is done by the same set of processes calling the library. For example, an MPI library for matrix operations first distributes the matrix and then all of the processes call the library routines to collectively perform the computation on their part of the matrix. Implementing more complex data structures on the same set of processes is difficult because of the need to intertwine the application code with the code for managing the data structure. As a result, implementations either use threads, as in implementations of tuple spaces [16] in MPI [17], or simply map the data structure onto its own set of resources as in C-MPI's implementation of a DHT for MPI [18]. FG-MPI provides an alternative way to map the application and data structure processes inside the same OS process.

This approach enhances portability, flexibility, predictability and controllability of the system. With regards to portability it relies on pure message passing in MPI and avoids the resource contention issues that arise in mixing MPI with threads [19]. FG-MPI enables one to expose more fine-grain concurrency and hence more parallelism to be potentially exploited. Each MPI process represents a "unit of execution" that can independently be replicated to scale up the program. As we describe in Section 4, we can flexibly map processes taking in account the size of the machine and to decide, relative to each other, which processes execute in parallel and which interleave their execution within a single core. The added predictability comes from the ability to use one core per OS process mapping to reduce the effect of OS noise [20] and the fact that an MPI-aware scheduler can provide more predictable interleavings. In addition, as described in Section 3, we can introduce a variety of parameters affecting the behaviour to give control over the granularity, number of outstanding messages and load-balancing.

These advantages could also be achieved by careful programming or re-factoring, but

[5]MPI defines `MPI_Request` objects that are used for a number of purposes. Requests here refer to send or receive request handles that are created internally as a result of MPI calls and are used to manage completion of MPI operations.

FG-MPI makes it easier to encapsulate program behaviour and externalise the parameters most affecting performance to match the application and machine. Although in FG-MPI there are some additional programming restrictions with regards to global and static variables[6] and overheads from the extra messaging costs, for many applications this is a reasonable trade-off for the increased programmability, portability and flexibility in exposing more fine-grain concurrency.

3. Ordered Linked-List Service

As a first step in the design of our ordered linked-list data structure service, we take a fine-grain approach where we represent each list node as an MPI process and store a single data item in that list node. Later in Section 3.5 we discuss a performance optimisation of the design to allow for multiple data items per list node, which lets the user relax granularity. By using FG-MPI we can develop and test this initial implementation with thousands of list items on one machine and even with millions of processes on a small cluster.

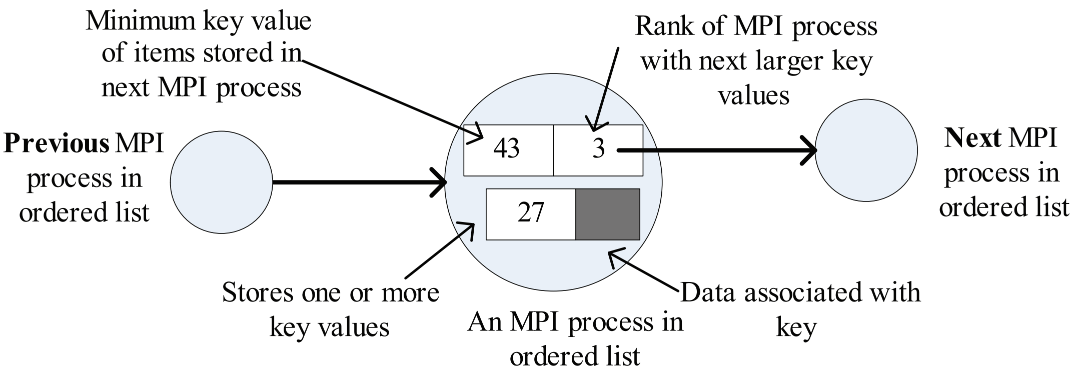

Minimum key value of items stored in next MPI process
Rank of MPI process with next larger key values
Previous MPI process in ordered list
43
3
27
Next MPI process in ordered list
Stores one or more key values
An MPI process in ordered list
Data associated with key

Figure 2. A node process (part of an ordered list)

As shown in Figure 2 each process (i.e., list node) in the list stores the following values:

a) the MPI process rank (i.e. identifier) of the next process in the list,
b) the key (or ordered list of keys in the multi-key case),
c) the data associated with the key, and
d) the minimum key (`min-key`) associated with the next process in the list.

The data stored inside each process is almost identical to that of the standard implementation of a linked list in a procedural language. The MPI process rank plays the role of a pointer to memory where a process can only communicate with the next process in the list. The only difference is item (d), the minimum key of the next process. This value is kept for correctness and performance of our list operations, which will be discussed in Section 3.1.

Our intent was to design a list potentially supporting millions of nodes with thousands of operations simultaneously accessing, adding and deleting from the list. In order to reason about the implementation and in particular to ensure freedom from deadlock we imposed the following two conditions:

a) processes initiate communication only in the forward direction, and
b) operations on the list process are atomic.

[6]The privatisation of global variables is commonly used in systems that allows multiple user-level threads to share the same address space and a number of tools are available to facilitate their privatisation [21,22].

Given a list of N processes, one can view the list as being comprised of a collection of nested servers where the i-th process is the client of the service provided by processes $i+1$ to N. According to our two conditions client i can only make requests to the service provided by process $i+1$. As well, the operation is atomic, implying that once a request has been made it completes and neither client i or the server at $i+1$ will accept new requests. In order to perform the request the client and server may engage in a series of communications as part of the request but this is assumed to terminate and all communications are local between a small fixed set of processes.

We claim that any request made to the head of the list eventually completes and thus the service is deadlock-free. The base case for server N is true because it depends only a series of local communications which are assumed to complete.[7] More generally, since list operations are atomic, we have that all requests by client $N-i$ to server $N-i+1$ complete and by induction it follows that requests by client $i=N-1$ (the first list process) complete. It also follows from the two conditions that requests to the list cannot jump over another request.

We take advantage of MPI message-passing semantics to enforce these two conditions. Every list process blocks on an `MPI_Recv()` with `MPI_ANY_SOURCE` (i.e., ready to receive a message from any source). Since a process's predecessor is the only process that knows its MPI rank, the predecessor is the only process that can initiate communication. Once this initial communication is made, the process learns the identity of its predecessor and the two can communicate back and forth to perform all local changes needed to complete the operation. Once both processes have completed the local operation they both block again on their respective `MPI_Recv()` routine. In addition MPI guarantees FIFO delivery of messages between two processes, which ensures that one message (operation) cannot overtake another inside the message queues; however special handling is necessary for operations where the process's predecessor changes (Sections 3.1.2, 3.1.3). In this case we need to ensure that all outstanding messages to a list process are received before switching that list node's predecessor.

3.1. List Operations

There are three types of requests that can be sent to our ordered linked-list service: FIND, INSERT and DELETE. Other operations such as UPDATE can be added but these three operations are sufficient to illustrate the basic properties of the service. Requests are of the form: operation type, source process rank, key-value, sequence number, and data (in the case of an insert operation). Application processes initiate requests and the process in the list performing the operation is the process that replies to the application process making the request.

In the following discussion we assume process A is one of the processes in the linked-list service and it is initiating contact with process B, the next process in the list. A third process, named C, with different roles for different operations is also present. We describe the implementation of the list operations in Sections 3.1.1, 3.1.2, and 3.1.3 together with Figure 3.

3.1.1. FIND Operation

As shown in step 1 of Figure 3(a), suppose that B has just received a FIND operation message from A. The message from A contains the operation, the key, the rank of the application node which made the request along with the application's sequence number. After receiving the message, B searches locally for the key in the message. Success is relayed back to the

[7]Server N acts as a sentinel and is configured with the NULL service, and any request generated by N signals failure which is relayed back to the application process.

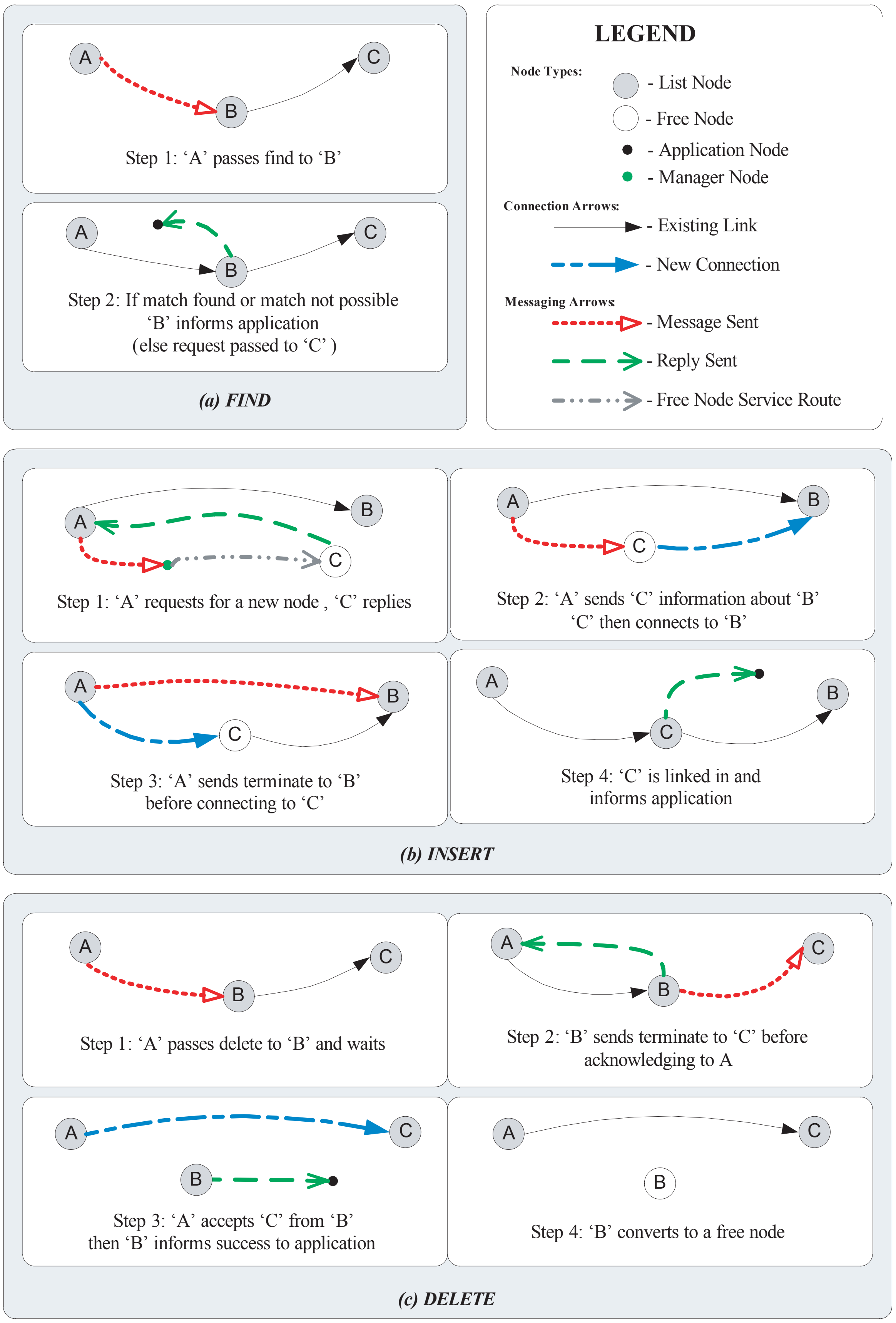

A
B
C
Step 1: 'A' passes find to 'B'
Step 2: If match found or match not possible 'B' informs application (else request passed to 'C')
(a) FIND
LEGEND
Node Types:
- List Node
- Free Node
- Application Node
- Manager Node
Connection Arrows:
- Existing Link
- New Connection
Messaging Arrows:
- Message Sent
- Reply Sent
- Free Node Service Route
Step 1: 'A' requests for a new node , 'C' replies
Step 2: 'A' sends 'C' information about 'B' 'C' then connects to 'B'
Step 3: 'A' sends terminate to 'B' before connecting to 'C'
Step 4: 'C' is linked in and informs application
(b) INSERT
Step 1: 'A' passes delete to 'B' and waits
Step 2: 'B' sends terminate to 'C' before acknowledging to A
Step 3: 'A' accepts 'C' from 'B' then 'B' informs success to application
Step 4: 'B' converts to a free node
(c) DELETE

Figure 3. Implementation of the ordered linked-list operations FIND, INSERT, and DELETE. The arrows denote the different communications among processes A, B and C; three consecutive processes that participate in the operations.

application process, while failure makes B compare the key in the message with the `min-key` value. Recall that the `min-key` is the minimum key (or just the key, in the single key case we are describing) stored at B's successor C. If the key is less than `min-key` and the key is not the key stored at B, then the key does not exist in the list and B sends a message to signal failure back to the application process. If the key is the key stored at B, then B replies to the application process with a message containing the data associated with the key. Finally, if the key is greater than `min-key`, then B simply forwards the original message onto C.

3.1.2. INSERT Operation

As in the case of FIND, the INSERT message passes from list process to process until it reaches the location to be inserted. Assume we are currently at A where the key to be inserted is greater than the key inside A and is less than `min-key` of successor B. We need to insert a new process for this key between A and its successor B. Notice that `min-key` allows A to detect that a process has to be inserted after A, whereas without `min-key` it is process B that would need to insert a new process before itself, and need to communicate with A. For this to occur, while at same time satisfying the forward-only and atomicity conditions, every forwarded request would need a reply. Storing `min-key` makes it possible to asynchronously forward requests without having to wait for a reply. This ensures that the common case, forwarding requests, is done as quickly as possible.

As shown in step 1 of Figure 3(b), A sends a free-node request to a manager process (see Figure 1) for the free-node service (see Section 3.3). The free-node service, if successful, replies back to A with the MPI rank of an available free list process (process C). If there are no free processes, then the free-node service reports failure to process A, and in turn process A reports failure back to the application process making the INSERT request.

In step 2, after A receives the rank of C, A sends the INSERT operation, the `min-key` and rank information it has about B to C. In step 3 A synchronously sends[8] a terminate connection message to B which signals that there are no outstanding messages from A, and B can now accept a message from some process other than A. A updates its own information with C as its successor. Finally, in step 4, once C has received the message from A it can store the data and signal success back to the application process. A no longer has information about B and can only forward requests to C. As well, C is the only process that knows about B and thus the only one that can forward requests to B.

It is possible to overlap steps 1 with steps 2 and 3 if the free node service is able to receive the data for C from A and have node C instantiated with all of the needed data. In this case A only needs to communicate with the free-node service and with B, after which it can continue to process requests.

3.1.3. DELETE Operation

In the case of DELETE, as long as the key is less than `min-key` the request is forwarded. When the key equals `min-key` then A knows that the next process, B, is to be deleted. Like INSERT, `min-key` helps to maintain our two conditions since it makes it possible to detect when the next process needs to be deleted.

As shown in step 1 of Figure 3(c), A sends a delete message to B, the node to be deleted, and waits for a reply from B. In step 2, using synchronous send, B sends a disconnect message to its successor C, which signals to C that there are no outstanding messages from B. B notifies A that it is finished by replying to A with the information about C. In step 3, A updates itself so it now points to C and continues. During steps 3 and 4, B replies back to the application and notifies the free node service that it is now free, and B now behaves as a free

[8]`MPI_Ssend` is a synchronous send, and completes only after a matching receive has been posted.

process. In step 2, having A wait for the reply from B ensures that no other request can jump ahead of A (equivalent to hand-over-hand).

3.2. Application Programming Interface

An important part of the design is the implementation of the service interface. Applications first need to know how to discover the service. We configure the application processes with a pointer (i.e., MPI rank) to the head of the list, which does not store any list items. Later in Section 3.6, to improve response time, we augment this with local service discovery, but for now we only consider the case when application processes send all requests to the head process.

Although the list communication is deadlock-free, the composition of the list with the application processes introduces the following deadlock scenario. It is possible for there to be a communication cycle starting from (a) the application process making the request, to (b) the list nodes traversed, to (c) the final list process that sends the reply message back to (a). The message interface to the service must enforce the condition that the application be ready to receive all replies whenever they have outstanding requests to the service. This ensures that for this deadlock scenario, while all the list processes may be trying to send, the application process will first receive before sending; thus avoiding deadlock. There are also important performance consequences to receiving replies quickly, since at steady-state the in-flow of requests equals the out-flow of replies; each new request depends on the receipt of a reply.

As discussed further in Section 3.4, each application process can have a fixed number of outstanding requests. The replies to these requests can return out of order because the order depends on an item's location in the list; replies by processes towards the end of the list are likely to return after a later request for a list item near the head of the list. As mentioned in Section 3.1 request messages contain sequence numbers. The application service interface adds the sequence number. Sequence numbers are carried along as part of the request and returned to the process inside the reply message. We use MPI's `MPI_Irecv` to pre-post receive buffers for the replies for each outstanding request. The pre-posted receive buffers can be used, if necessary, as a hold-back queue [23] which can be managed by the application to re-order and return replies in the order of the requests. When all initial requests are sent to the head of the list, the list structure has the property that, since all requests are received in-order from the application process and no request can jump ahead of another one as requests traverse the list, we can maintain sequential consistency.

Moreover, although not done, if the head process replaced the sequence number of the request with the message-order timestamp, then all requests are totally ordered. The timestamp action can be viewed as a linearization point which implies the linearizability of executions, again assuming all requests are sent to the head process. Later in Section 3.6 we relax this condition.

3.3. Free Process Sub-service

MPI creates a static namespace at the initialisation of the MPI execution environment (`MPI_Init`), and assigns consecutive integer ranks to MPI processes in the range 0 to $N-1$. FG-MPI decouples the namespace of the MPI processes from that of the OS processes to allow for multiple MPI processes per OS process. We take advantage of FG-MPI's ability to expose large-scale concurrency to create a pool of free processes at the beginning of the program. This pool of free processes can be used to dynamically allocate and deallocate processes in response to the application load. As we discuss in this section, a free process is reincarnated as a list node process in case of an INSERT operation and released back to the pool on DELETE. This can be viewed as spawning of processes on a fixed namespace. The free processes are all blocked waiting for an allocate request and do not impose an overhead on the scheduler (see Section 4). The manager processes are able to dynamically load balance the allocation of free processes by communicating with each other. We discuss the details of the free process sub-service next.

At the start of a $[P, O, M]$ execution for our list service, three different types of MPI processes are launched per OS process, see Figure 1 and see Section 4 for the details on mapping. One of them is an application process, the second is a manager process and the remaining $P-2$ processes are ordered-list processes. Under this mapping the maximum size of the list is $(P-2) \times O \times M$ with $(P-2)$ nodes (i.e. ordered-list processes) per OS process. Initially the ordered list processes are enqueued in a local pool of $(P-2)$ free processes. Other mappings are possible and the only constraint is that whenever an OS process has an ordered-list process there must also be a manager process. This allows us to define the size of the list as well as control the distribution of list nodes to OS processes to balance the communication and work between co-located processes (interleaved) and non co-located processes (parallel).

On start-up each of the manager processes has a list of the list processes local to its OS process. It also has information about the ranks of manager processes in other OS processes. Managers use a pre-defined load-balancing strategy whose goal is to evenly distribute list processes to OS processes while trying to keep communication local between co-located processes. The trade-off between these two conflicting goals depends on the distribution of keys and overall workload and thus should be configurable.

In our experiments the keys are uniformly random from a 32-bit key space. We tested with varying workloads but all cases used the following simple load-balancing strategy. In the case of a free node request because of an INSERT operation the manager first attempts to satisfy the request locally. If there are no local free nodes, then the manager passes the allocation request onto the next manager where all managers have been configured into a ring and each knows the next manager in the ring. The manager who succeeds in allocating a free node replies directly to the list node making the request, allowing the INSERT to complete. If no free node is found by any of the managers after one round robin cycle, then the local manager signals failure back to the list node who made the request. In turn, the list node notifies the application process that the INSERT operation failed because the list is full. When the list is full or almost full, guaranteeing liveness is up to the application and the list service does not guarantee fairness. In the ring of manager processes deadlock is avoided by ensuring that each manager posts a receive buffer for receiving a message from the previous process before attempting to send a message to the next manager in the ring. Because the receive buffer is always pre-posted, MPI middleware can always deliver a message sent by one manager process to the next manager in the ring.

In the case of a DELETE, the free node registers itself back on the free node list. As described further in Section 3.6, locally we maintain a look-up table that processes can post to and consult. DELETE operations are fast, since they only require local operations, whereas INSERT can incur much larger delays when non-local communication is needed to satisfy the request.

Load-balancing is a good example of a strategy that impacts performance and needs to be configurable. The cost can depend on the workload, the relative cost of communication between local and non-local processes, the distribution of keys, and also the size of the list and P (the number of local processes per OS process). An advantage of this service-oriented approach is that load-balancing code is not scattered through the program but in one place and adding a new strategy only requires extending the manager processes inside the free node service.

3.4. Flow Control (W and K)

We define a configuration parameter W in the application API to control the flow of requests to the list. W is the maximum number of outstanding requests to the list service that an application process can have at any time. Let cw_i denote the current number of outstanding requests (i.e. number of requests minus number of replies) from process i. As previously mentioned, to avoid deadlock and improve performance, we require that whenever $cw_i > 0$, the application process must be able to receive a reply. The sum of all cw_i's at time T is the total load on the service. In the experiments in Section 5 we set W to be a constant for all application processes and increase the rate of requests together with W to reach steady-state where globally the flow of requests to the service matches the flow of replies from the service. This is used to measure the peak performance of the service. W cannot be set arbitrarily high, particularly when all requests go to the head of the list, since the saturation point tends to occur towards the head of the list and there are overheads in the middleware that increase, which result in decreased throughput. Thus peak throughput is achieved by carefully setting W. This parameter can be extended so as it can be changed at run time per application to maintain peak throughput.

In MPICH2, as in most MPI implementations, short messages are sent eagerly and buffered at the destination until matched by a receive. MPICH2, and as a result FG-MPI, has a global buffer pool that allocates space for eager messages in addition to other dynamic MPI objects. There is no flow control on eager sends and MPI simply aborts the execution when the buffer pool in the middleware of an OS process is exhausted. There is no simple technique for detecting these types of buffer problems in MPI, but it is possible to avoid them without resulting to completely synchronous communication [24]. To avoid these errors we implemented a *bounded asynchronous* scheme where for every K standard MPI sends (potentially eager sends) we send a synchronous `MPI_Ssend()` to ensure that there are no outstanding matching messages between the two processes. In the case of $K = 0$, every send becomes synchronous. The synchronous case is interesting because it is one technique to test for potential deadlock situations that are masked when buffering is available. Because MPI cannot guarantee the amount of buffering available for asynchronous communication the MPI standard defines a *safe* program as one that can execute synchronously. However, as was evident in our experiments, it is important to increase K as large as possible to overlap the computation and communication, while also ensuring there is always sufficient space in the buffer pool. This is all the more important in FG-MPI because there are more MPI processes sharing the middleware's buffer pool and thus a greater chance of exhausting it.

3.5. Granularity (G)

An obvious extension to our ordered list is to allow each list process to store more than one item. As previously described each process stores its local set of keys and the minimum key of its successor. It is simple to modify the list process to perform a FIND, INSERT or DELETE to search its local set of items to perform the operation. The only added complication occurs for INSERT since we can either add the item to the current node's set of items or insert the item by adding a new process. The INSERT operation was changed so that now the process at the insertion point (process A in Section 3.1.2) can decide to add the item to its own set of items or split the items between itself and a new process. This introduces another opportunity to add a load balancing strategy to attempt to optimise for the number of items to store locally. We used a simple strategy which split the items whenever the number of items exceeds a fixed threshold. For DELETE operations, if the delete key matches the `min-key` stored in a process, this implies either the next process is to be deleted, or when there are multiple items in the process, `min-key` needs to be updated. Both these cases can be handled by the reply from B to A in step 2 of the DELETE operation shown in Figure 3(c).

We call the maximum number of list items that a list node process can hold the *granularity*, G, of the list service. This can be adjusted dynamically. It is possible to set G initially to be small when the list is small and then increase it as the list size increases. Both G and P are helpful in reducing the idle time. Increasing G reduces the amount of messaging and therefore the amount of work done in performing requests. However, to maximise the opportunity for parallelism we need to distribute the list to the $O \times M$ processing cores.

3.6. Shortcuts

As mentioned earlier, the application processes send requests to the head of the linked list. This has an obvious disadvantage of creating congestion at the head of the list. When P is greater than one there are potentially active list processes co-located in the same address space. Rather than always send a request to the head, one optimisation is to have the application process first check locally. If the application process can look-up the key-values of the local list processes along with the associated rank, then it can send the request to the process whose key-value is closest to, but not greater than, the requested key-value. When none exist, it can always by default send the result to the head. One can view the local list processes as providing a set of possible shortcuts that the application process can use to reduce response time as well as reduce congestion for the head and processes towards the front of the list.

The disadvantage to the indiscriminate use of shortcuts is that there is no longer any guarantee on the order of operations since a later operation, on even the same item, could find a new shortcut allowing it to get ahead of an earlier request. However, for every outstanding request the application process can store the key-value of any shortcut that was used. This allows it to always choose the shortcut less than or equal to the earliest shortcut used by an outstanding request, thereby guaranteeing requests are performed in order. Likely the application process only cares about the order of requests to a given data item and gives more opportunity to use shortcuts since now earliest can be defined with respect to each individual key.

Given the existence of these shortcuts one can go further and make them available to the list processes themselves. There is no longer any guarantee on the order in which concurrent requests are done, even requests to the same key-value. Of course, the application process always has the option to wait for the reply before submitting the next request. This use of shortcuts by the list introduces an interesting type of probabilistic data structure where the shortcuts help balance requests and significantly reduce response time. The number of shortcuts, and hence the likelihood of finding good ones, depends on the size of the list and P. On average, assuming a uniform distribution, one expects a cost of N/P rather than $N/2$ communications to access N list processes distributed across all of the OS processes. As well, shortcuts help distribute the requests across the entire list of processes rather than at the front of the list. The managers also influence the quality of shortcuts. If managers only allocate free processes locally, then the shortcuts will tend to be local as well, whereas allocating non-locally results in better shortcuts but higher communication cost. The use of shortcuts, even without ordering, is good for the important use case where inserts and deletes are rare and the common operation is find.

Shortcuts are implemented by the manager process and service discovery is done by having each application process contact the local manager. We take advantage of the single address space of the OS process and the non-preemptive scheduling (i.e. atomicity) of processes to implement a shared look-up table. The list processes update the table when they are inserted or deleted and the application can query the table to find a suitable shortcut. Although this violates pure message-passing semantics, it can be viewed as part of the local execution environment of the processes similar to the `mpiexec` environment variables passed to the program at launch time or OS environment variables that can be read during execution.

Note that using shortcuts this way can lead to race conditions, where a message may be sent to a shortcut but the shortcut may be deleted or changed by the time the message reaches it. We are able to detect when a shortcut no longer exists and we modified the list and free processes to report shortcut failure back to the application when this occurs. We are also able to turn off the use of shortcuts and the application can re-submit the request with shortcuts off for that particular request.

We use the terms *total-ordering* for the implementation that does not use shortcuts, *sequentially-consistent* for the one where only application processes use shortcuts, and *no-consistency*[9] for the version that allows use of shortcuts by both the application and the list processes. We extended our service discovery to include both these shortcut semantics.

4. Mapping Issues

In this section we describe how the application, manager and list processes are mapped to OS processes (as an example see Figure 1). The `mpiexec` command performs a 1-dimensional mapping of MPI processes to different cores and machines. FG-MPI provides a second dimension to the mapping of multiple MPI processes within the same OS process. Just as `mpiexec` binds OS processes to cores, we define a function `FGmpiexec`, called from `main`, that binds co-located MPI processes to different functions. The binding of processes to functions can be defined in a general way through a user-defined function that takes a process's `MPI_COMM_WORLD` rank as its input parameter and returns a pointer to the function that MPI process will be executing. Note that each of these functions, which the processes are bound to, are written as regular MPI programs, beginning with `MPI_Init` and ending with `MPI_Finalize` calls. The number of co-located processes is specified by the `nfg` parameter to `mpiexec` and as shown in Figure 1, the MPI process ranks are assigned in consecutive block ranges of size `nfg`.

Listing 1 shows an example of binding MPI processes to three different types of functions per OS process: `LocalManagerNode` function for the manager process, `AppNode` function for the application process and `ListNode` function for the list processes. Listing 1 contains two user-defined mapping functions; `binding` and `map_lookup`. The `binding` function, as previously mentioned, takes the process rank as input and returns the corresponding function pointer. The `map_lookup` mapper function takes a string as its third parameter and maps that to a binding function. In this example the `str` parameter is not being used, however, its purpose is to allow selection of different binding functions at runtime through the `mpiexec` command line. The FG-MPI runtime system inside each OS process is initialised through a call to the `FGmpiexec` function, which takes `map_lookup` as a parameter and spawns the MPI processes. For the linked-list example this allows us to interleave the application and data structure node processes within the same OS process.

Depending on the operations on the list service, the list processes may change state from "free" to "active" and vice versa. As mentioned in Section 3.3, initially all the $P-2$ list processes add themselves to the local pool of free list processes. Adding to the free list is done by each of the list processes executing a `MPI_Recv` call. Since this is a receive call for which no matching messages have arrived, the receive call cannot complete and FG-MPI's scheduler places all these list processes in its blocked queue. In the free state, these list processes do not take up any additional resources other than the small amount of memory used to store state. These list processes move to "active" state and become part of the list service when they are allocated by the free process service, which unblocks them by sending a message to them.

[9]No-consistency means there is no guarantee on the order in which requests are performed. Note, the list itself always properly orders the items.

```
/******* FG-MPI Boilerplate begin *********/
#include "fgmpi.h"
/* Forward declarations of the three
 * functions that the MPI processes are bound to. */
int ListNode(int argc, char** argv);
int LocalManagerNode(int argc, char** argv);
int AppNode(int argc, char** argv);
FG_ProcessPtr_t binding(int argc,char** argv,int rank)
{
    if (rank == starting_rank_of_co-located_processes)
      return (&LocalManagerNode);
    else if (rank == last_rank_of_co-located_processes)
      return (&AppNode);
    else
      return (&ListNode);
}
FG_MapPtr_t map_lookup(int argc,char** argv,char* str)
{
    return (&binding);
}

int main( int argc, char *argv[] )
{
    FGmpiexec(&argc, &argv, &map_lookup);
    return (0);
}
/******* FG-MPI Boilerplate end *********/
```

Listing 1. An example of MPMD mapping for the linked list.

FG-MPI makes it possible to combine those processes implementing the service with the application processes using the service. As a service this makes it possible to separate the data structure code from the application code as well as having the ability to flexibility map application and service processes in a variety of ways.

5. Experiments

In this section we experiment with different parameters of the ordered linked-list service and show how varying them affects the performance. Recall that operations on a linked list are inherently non-scalable and our objective here is to demonstrate the working of a service-based approach and the flexibility in tuning of its parameters. In all the experiments reported below we report the steady-state throughput (i.e., number of operations per second) of the ordered linked list. The INSERT operation was initially used to build the list to capacity and then two different workloads were used to measure the throughput. One workload consisted of an equal number of INSERT, DELETE and FIND operations and the second consisted of FIND operations only. The two workloads produced similar results with a slight difference in performance, therefore, we report the results for the second workload only.

For the experiments, the test setup consisted of a cluster with 24 machines connected by a 10GigE Ethernet interconnection network. Each of the machines in the cluster is a quad-core, dual socket (8 cores per machine) Intel Xeon® X5550, 64-bit machine, running at 2.67 GHz. All machines have 12 GB of memory and run Linux kernel 2.6.18-308.16.1.el5.

5.1. Consistency Semantics

Figure 4 shows the throughput achieved with the three consistency semantics described in Section 3.6, i.e., total-ordering, sequentially-consistent and no-consistency. The size of the

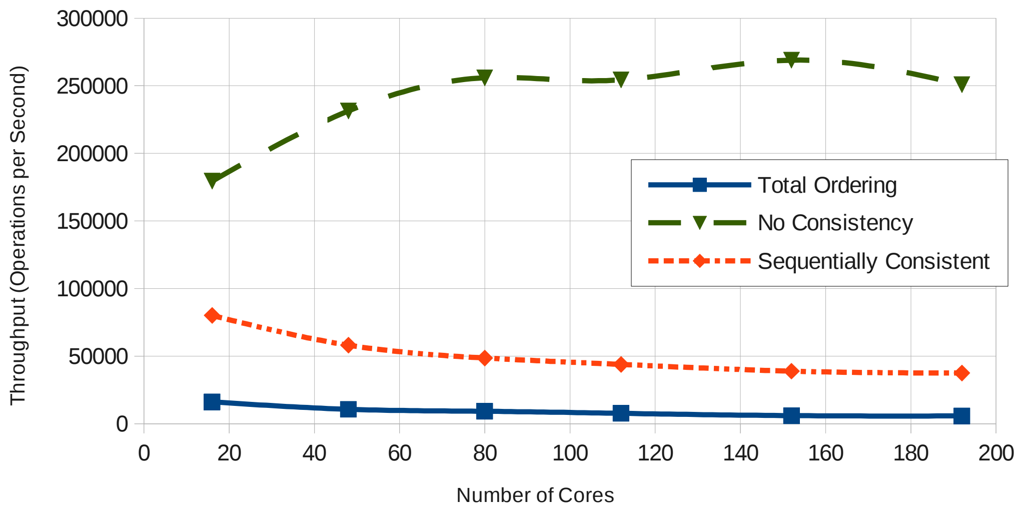


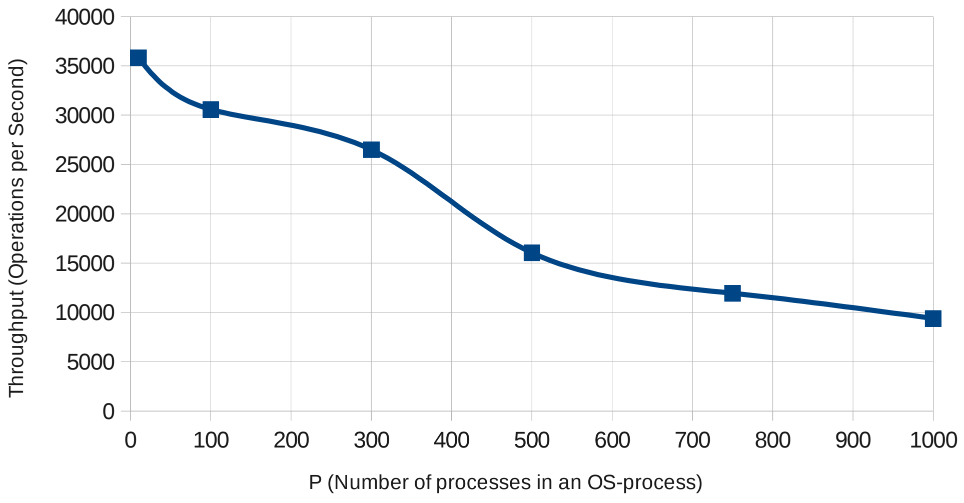
Figure 4. Throughput achieved with different consistency semantics for a fixed list size, evenly distributed over the number of cores=$(O \times M)$.

linked list was set to 2^{20} in this experiment and the number of list nodes were evenly distributed over the number of cores ($O \times M$), by setting $ListSize \div (O \times M) = P \times G$, with P set to 10. As expected, the throughput with no-consistency is the highest followed by that for sequential-consistency and then total-ordering. In Figure 4, as we move from left to right along the x-axis, the list is becoming more distributed and the number of items being stored per list node is decreasing, while keeping the list size constant. The no-consistency semantics benefits from more distributed nature of the list because the requests are spread over more cores. One could argue that, in the probabilistic sense, selecting the optimal shortcut for the no-consistency semantics takes a maximum of $\mathcal{O}(O \times M)$ steps. Operations with the other two types of semantics i.e., total ordering and sequential consistency have to traverse the list to reach the correct list node and on the average take $\mathcal{O}(O \times M \times P)$ steps assuming G is small enough to neglect its cost. Total ordering also suffers from congestion in the head of the list, which is why its curve is, as expected, the worse of the three.

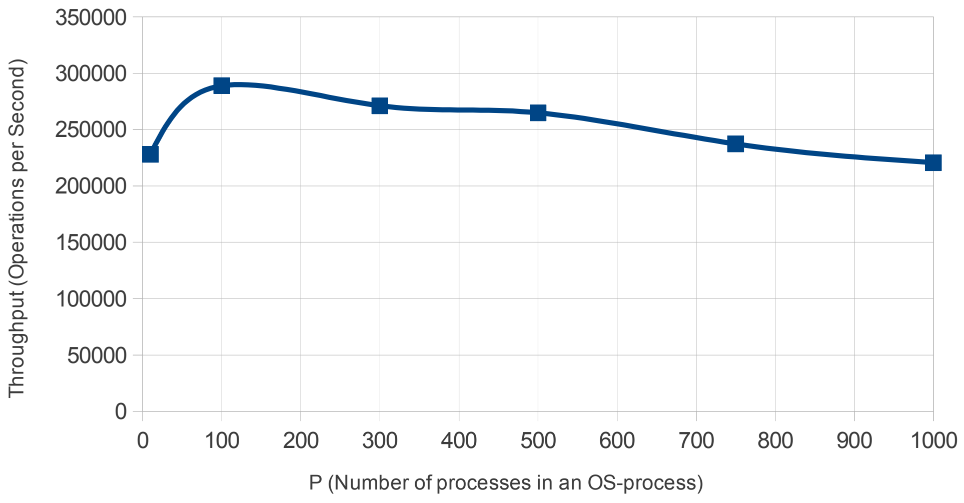
5.2. *G versus P*

Figure 5 shows how performance changes when interleaved concurrency (P) is increased while reducing granularity (G), keeping the list size and the number of cores fixed. Since the number of cores is fixed, the overall amount of parallelism is fixed and the figure shows the trade-off between interleaved concurrency versus coarser-grain processes with more list items inside a process. Figures 5(a) and (b) show the trade-off between G and P with sequentially-consistent and no-consistency shortcut semantics. In Figure 5(a), as expected, it is better to reduce P and increase the granularity, G.

Parameters G and P define the first two levels in the communication hierarchy of the machine. In terms of traversing the list, from fastest to slowest, there is traversal (a) inside a process, (b) between processes inside the same OS process, (c) between processes on the same machine, (d) between processes on separate machines. There is a significant difference in communication costs between these levels (a) tens of nanoseconds, (b) less than a microsecond, (c) a few microseconds, and (d) on the order of 50 microseconds. Making G large and P small is analogous to keeping all data in the primary cache rather than the secondary cache. Figure 5 is a simple illustration of the trade-offs. Since the focus of the paper is on the design of the library we did not explore the use of different applications or additional load-sharing strategies for the manager processes to better exploit the hierarchy.



(a) Sequentially-consistent.



(b) No-consistency.

Figure 5. Effect of changing P, while keeping $P \times G$ constant. The list size and the number of cores are fixed. Semantics used are (a) sequentially-consistent and (b) no-consistency. (ListSize = $G \times P \times O \times M$=$2^{20}$ and cores=$O \times M$=176).

There are interesting questions that arise with regards to efficiently managing communication costs across the communication hierarchy. On a list of size N, assuming uniformly distributed keys, we have that $N = G \times (P \times O \times M)$ where the expected number of hops is $\frac{1}{2}(G + P \times O \times M - 1)$. With regards to latency (response time) the optimal distribution is to have $P = 1$, $G = N/(O \times M)$ and map OS processes consecutively from one machine to the next to keep communication as local as possible. But for throughput, increasing G limits the rate at which requests enter the system, since when viewed as a pipeline the search inside the list of size G at the FIND location delays all requests behind the current one. Although this depends on the location of the FIND, in steady-state it is likely to occur often and have more of an effect closer to the head of the list where the load is higher. Intuitively, larger G reduces the capacity of pipeline and speeds up requests while increasing P increases the capacity of the pipeline but slows down requests. We believe the value of G in Figure 5(a) was never large enough to see this effect and provide a rationale for increasing P in the sequentially-consistent case.

The throughput shown in Figure 5(b), with the use of shortcuts inside the list of processes, is significantly larger than that of Figure 5(a). P determines the number of possible shortcuts inside each OS process. Once there are more than a few shortcuts there is a high probability of the existence of a shortcut to a closer OS process. As a result, the expected number of hops begins to depend on the number of OS processes rather than the size of the list. There is a diminishing return for adding more shortcuts since there is less chance of it being of value. As Figure 5(b) shows the first 100 shortcuts increase performance after which more shortcuts is adding more overhead than benefit as P increases.

5.3. W and K

Parameter W specifies the number of outstanding requests that each application can submit to the list service. Increasing the value of W increases the load on the system. K is a throttling parameter that each list node uses to specify how many requests can be forwarded down the list, before having to wait for a previously forwarded message to complete. In the perfectly balanced case, when W equals P, each list node receives exactly one application request. When W exceeds P, each list node progresses multiple application requests up to a maximum of K. The smaller the value of K, the higher the throttling effect on the flow of requests through the list service. Figure 6 shows the effect of different values of K on the throughput.

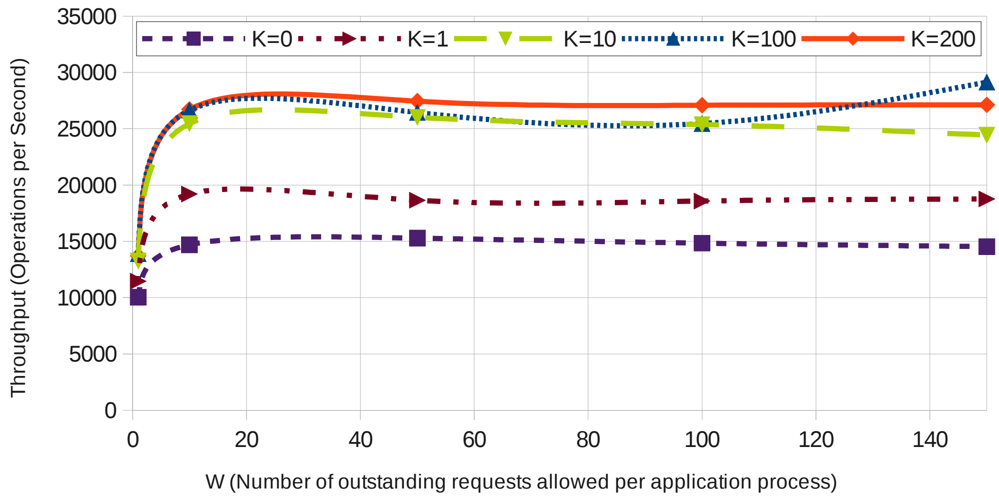


Figure 6. Throttling effect of K on the throughput while increasing the outstanding requests (W) by the application processes. List size used is 2^{20}. $P = 100$. $O \times M = 192$ cores. Sequentially-consistent semantics are used in this experiment.

The effect of small K clearly limits the throughput. Increasing K increases fluidity in the network for request propagation. There is a limit to the rate at which a process can forward request, and once K exceeds this limit it no longer has any effect (it's at full throttle). This limit depends on computation and communication characteristics of the machine. K gives us a control over the flow inside the service and it should be tuned to the communication characteristics of the underlying network.

6. Conclusions

In this paper we discussed a service-oriented approach to the design of a distributed data structures and presented an ordered linked-list service as an example. The implementation of a distributed data structure as a service uses its own collection of processes which makes it possible to separate the data structure code from the application code resulting in reduced coupling and increased cohesion in the components of the system.

In parallel computing, more so than in other areas, there is the additional challenge that the performance of the library must be portable across a wide range of machines sizes and communication fabrics. Scalability across machines and communication fabrics requires the addition of tunable parameters to control resources based on the characteristics of the machine and networks. In our design we introduced a variety of these parameters to control computation/communication overlap (K), interleaving versus parallel (P), load on the data structure (W) and process granularity (G).

The service-oriented approach to the design of an ordered linked list is substantially different from existing work. It borrows ideas from work on concurrent linked lists but is based on pure message-passing and does not require support for shared memory. There is some commonality with distributed systems in cloud environments but with a focus on scalability and latency, and not availability and reliability. Our ordered linked list implementation can execute on a single multicore but more importantly it can also potentially scale to machines with thousands of cores to support sorting and querying for large datasets that can take advantage of the large amounts of available memory.

The experiments demonstrated the ability to scale to an ordered linked list with over a million data items with up to 192,000 MPI processes executing on 192 cores. As expected, for a list of size N, the $\mathrm{O}(N)$ communications per request makes the one item per process impractical for large lists. However, we showed that better performance is possible and depends on the level of consistency needed and trade-offs between G, the number of list items stored inside a process, and P, the number of processes inside an OS process. The values of P, O, G and N are all runtime or dynamic parameters that do not require re-compilation of the program making it possible to flexibly map an application onto varying types of cluster architectures without a need to re-design or recompile the library.

In conclusion, by using a service-oriented approach we are able to create an ordered linked-list library that can be used within MPI programs. The linked list is a relatively complex data structure and our design decouples the code needed to implement the list from the application and encapsulated the code related to performance inside the library while exposing a simple set of parameters for the application programmer or user to flexibly tune the performance of the system. This supports our view that a service-oriented approach to distributed data structures provides an interesting design space for large scale, high performance, distributed systems. It lies at the intersection of parallel computation and distributed system design, and well-suited for the types of cluster of multicore machines used in high performance and cloud computing environments.

Acknowledgements

The authors wish to thank the anonymous reviewers for their valuable comments. We gratefully acknowledge NSERC Canada for their support of this research.

References

[1] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, August 2007.

[2] Carl G. Ritson and Peter H. Welch. A process-oriented architecture for complex system modelling. *Concurrency and Computation: Practice and Experience*, 22:965–980, March 2010.

[3] P.H. Welch. Process Oriented Design for Java: Concurrency for All. In H.R.Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000.

[4] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[5] Daisuke Tsukamoto and Takuo Nakashima. Implementation and evaluation of distributed hash table using MPI. *Broadband, Wireless Computing, Communication and Applications, International Conference on*, 0:684–688, 2010.

[6] Humaira Kamal and Alan Wagner. An integrated fine-grain runtime system for MPI. *Computing*, pages 1–17, 2013.

[7] Humaira Kamal and Alan Wagner. Added concurrency to improve MPI performance on multicore. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 229 –238, Sept. 2012.

[8] C. Iancu, S. Hofmeyr, F. Blagojevic, and Yili Zheng. Oversubscription on multicore processors. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –11, april 2010.

[9] Peter H. Welch and Frederick R. M. Barnes. Communicating mobile processes: Introducing occam-pi. In Ali Abdallah, Cliff Jones, and Jeff Sanders, editors, *Communicating Sequential Processes. The First 25 Years*, volume 3525 of *Lecture Notes in Computer Science*, pages 712–713. Springer Berlin / Heidelberg, April 2005.

[10] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[11] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, PODC '04, pages 50–59, New York, NY, USA, 2004. ACM.

[12] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[13] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[14] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[15] R Morris IonStoica, David Karger, M Frans Kaashoe, and Hari Balakrishnan. Chord: A scalable peer-topeer lookup service for internet applications. In *SIGCOMM01*, pages 27–31.

[16] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.

[17] L. M. Silva, J. G. Silva, and S. Chapple. Implementing distributed shared memory on top of MPI: The DSMPI library. In *Proceedings of the 4th Euromicro Workshop on Parallel and Distributed Processing (PDP '96)*, PDP '96, pages 50–, Washington, DC, USA, 1996. IEEE Computer Society.

[18] J. M. Wozniak, B. Jacobs, R. Latham, S. Lang, S. W. Son, and R. Ross. C-MPI: A DHT Implementation for Grid and HPC Environments. (Preprint ANL/MCS-P1746-0410), April 2010.

[19] William Gropp. MPI 3 and beyond: Why MPI is successful and what challenges it faces. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *EuroMPI*, volume 7490 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2012.

[20] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 19:1–19:12, Piscataway, NJ, USA, 2008. IEEE Press.

[21] Photran. An Integrated Development Environment and Refactoring Tool for Fortran. Available from `http://www.eclipse.org/photran`.

[22] Elsa: An elkbound based C++ parser. Available from `http://www.cs.berkeley.edu/~smcpeak/elkhound`.

[23] G.F. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. International computer science series. Addison-Wesley, 2011.

[24] Alex Brodsky, Jan Bækgaard Pedersen, and Alan Wagner. On the complexity of buffer allocation in message passing systems. *Journal of Parallel and Distributed Computing*, 65(6):692 – 713, 2005.