Distributed multiscale computing

Borgdorff, J.

3

Distributed multiscale runtime environment[1]

Abstract We present the Multiscale Coupling Library and Environment: MUSCLE 2. This multiscale component-based execution environment has a simple to use Java, C++, C, Python and Fortran API, compatible with MPI, OpenMP and threading codes. We demonstrate its local and distributed computing capabilities and compare its performance to MUSCLE 1, file copy, MPI, MPWide, and GridFTP. The local throughput of MPI is

[1]The contents of this chapter are based on:

• J. Borgdorff, M. Mamonski, B. Bosak, K. Kurowski, M. Ben Belgacem, B. Chopard, D. Groen, P. V. Coveney, and A. G. Hoekstra. Distributed Multiscale Computing with MUSCLE 2, the Multiscale Coupling Library and Environment. *Journal of Computational Science, in press*, 2014. doi: 10.1016/j.jocs.2014.04.004

• D. Groen, J. Borgdorff, C. Bona-Casas, J. Hetherington, R. W. Nash, S. J. Zasada, I. Saverchenko, M. Mamonski, K. Kurowski, M. O. Bernabeu, A. G. Hoekstra, and P. V. Coveney. Flexible composition and execution of high performance, high fidelity multiscale biomedical simulations. *Interface Focus*, 3(2):20120087, Apr. 2013. doi: 10.1098/rsfs.2012.0087

about two times higher, so very tightly coupled code should use MPI as a single submodel of MUSCLE 2; the distributed performance of GridFTP is lower, especially for small messages. We test the performance of a canal system model with MUSCLE 2, where it introduces an overhead as small as 5% compared to MPI.

3.1 Introduction

The conceptual framework in the previous chapter describes the process of constructing a multiscale model by identifying and separating its scales. The Multiscale Modelling Language (MML) then specifies how the model can be computed [22, 47]. In this chapter we present a means to implement multiscale models: the Multiscale Coupling Library and Environment 2 (MUSCLE 2). It takes a component-based approach to multiscale modelling, promoting modularity in its design. In essence, it treats single scale models as a separate components and facilitates their coupling, whether they are executed at one location or multiple. It is open source software under the LGPL version 3 license and is available at `http://apps.man.poznan.pl/trac/muscle`. MUSCLE 1 [65] generally had the same architecture and it was based on the Complex Automata theory [67, 70] and focussed on multi-agent multiscale computing; however, MUSCLE 1 and 2 share only 4% of their code, and their differences are discussed in Appendix B.3.

Distributed computing is a way to take advantage of limited and heterogeneous resources in combination with heterogeneous multiscale models. There are several motivations for distributing the computation of a multiscale model: to make use of more resources than available on one site; making use of heterogenous resources such as clusters with GPGPUs, fast I/O, highly interconnected CPU's, or fast cores; or making use of a local software license on one machine and running a highly parallel code on a high-performance cluster. Projects such as EGI and PRACE make distributed infrastructure available, and software that uses it is then usually managed by a middleware layer [155]. The QCG-Coordinator [27] middleware, for example, allows users to use computational resources at multiple locations in Europe, and it supports MUSCLE 2 for this purpose.

Quite a few open and generic component-based computing frameworks already

exist, for instance the CCA [8] with CCaffeine [7], the Model Coupling Toolkit (MCT) [83, 89], Pyre [34], or OpenPALM [35]; see the full comparison by Groen *et al.* [62]. The Model Coupling Toolkit has a long track-record and uses Fortran code with MPI as a communication layer so it potentially makes optimal use of high-performance machines. OpenPALM uses TCP/IP as a communication layer and it is packaged with a graphical user interface to couple models. Both frameworks provide some built-in data transformations. MUSCLE 2 uses shared memory for models started in the same command and TCP/IP for multiple commands. An advantage over the other mentioned frameworks is that it provides additional support for distributed computing and for Java. However, it has fewer built-in data transformations available and does not provide tools for implementing the contents of single scale models, so it should be combined with domain-specific libraries.

There are many libraries for local and wide-area communications, apart from MPI implementations and the ubiquitous TCP/IP sockets. MPWide [59], for instance, is a lightweight library that optimises the communication speed between different supercomputers or clusters; ZeroMQ [74] is an extensive communication library for doing easy and fast message passing. To use them for model coupling these libraries have to be called in additional glue code. MUSCLE 2 optionally uses MPWide for wide area communication because of its speed and few dependencies.

So far MUSCLE 2 is being used in a number of multiscale models, for instance a collection of parallel Fortran codes of the Fusion community [53], a gene regulatory network simulation [134], a hydrology application [16], and in a multiscale model of in-stent restenosis [20, 32, 60].

In this chapter, we introduce the design of MUSCLE 2 in Section 3.2, including its conceptual background, MUSCLE 2's API (Application Programming Interface) and runtime environment. The performance and startup overhead of MUSCLE 2 is measured in Section 3.3 in a number of benchmarks. Finally, in Section 3.4 two applications that use MUSCLE are described, principally a multiscale model of a complex canal system, for which additional performance tests are done.

3.2 Design

MUSCLE 2 is a platform to execute time-driven multiscale simulations. It takes advantage of the separation between the submodels that together form the multiscale

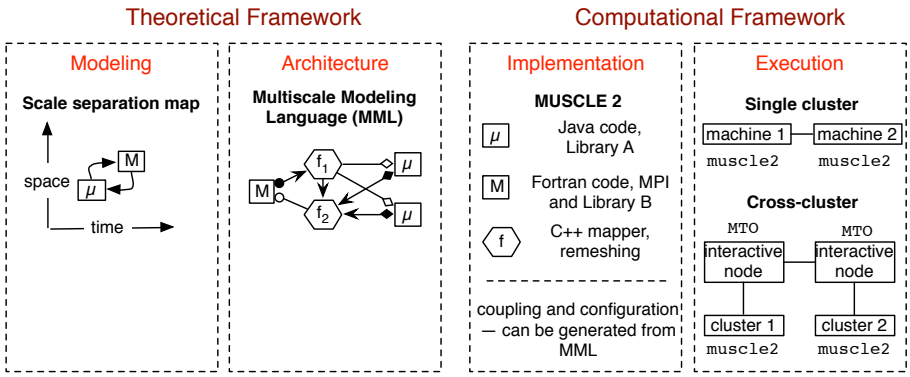


Figure 3.1: Overview of the approach presented in this paper. First, use the multiscale modelling and simulation framework to characterise a multiscale model: create a scale separation map (for example with macro model M and micro model μ), and translate it to the computationally oriented multiscale modelling language, including any mappers f_1 and f_2 to do scale bridging and/or data conversions (Sec. 3.2.1). The architecture is implemented and coupled with MUSCLE 2 (Sec. 3.2.2 and 3.2.3), and finally, executed on a single machine or cluster with plain MUSCLE 2 (Sec. 3.2.4), or cross-cluster using the MUSCLE Transport Overlay (MTO, Sec. 3.2.5).

model, by treating each submodel as a component in a component-based simulation. The submodels individually keep track of the local simulation time, and synchronise time when exchanging messages.

A strict separation of submodels is assumed in the design of MUSCLE 2, so the implementation of a submodel does not dictate how it should be coupled to other submodels. Rather, each submodel sends and receives messages with specified ports that are coupled at a later stage. When coupling, modellers face their main scientific challenge: to devise and implement a suitable scale bridging method to couple single scale models. MUSCLE 2 supports the technical side of this by offering several functional components, described in Sec. 3.2.1.

The runtime environment of MUSCLE 2 executes a coupled multiscale model on given machines. It can run each submodel on an independent desktop machine, local cluster, supercomputer, or run all submodels at the same location. For instance, when one or more submodels have high computational requirements or require alternate resources such as GPU computing, these submodels can be executed on a suitable machine, while the others are executed on a smaller cluster. A requirement is that a

connection can be established between submodels, and that a message can only be sent to currently running submodels. For some models a local laptop, desktop or cluster will suffice; MUSCLE 2 also works well in these scenarios. Technical details about the runtime environment can be found in Appendix B.

MUSCLE 2 is separated into an API, which submodel code uses, a coupling scripting environment that specifies how the submodels will be coupled, and a runtime environment, that actually executes the multiscale model on various machines. The library is independent from the coupling, which is in turn independent from the runtime environment. As a result, a submodel is implemented once and can be coupled in a variety of ways, and then executed on any suitable set of machines. Additionally, future enhancements to the runtime environment are possible without changing the library.

3.2.1 Conceptual background

To generally couple multiscale models, a framework describing the foundations of multiscale modelling [19, 36, 75, 76] (see Ch. 2) and its repercussions on multiscale computing [22, 47] was conceived. It starts by decomposing a phenomenon into multiple single scale phenomena using a scale separation map as a visual aid. Based on these single scale phenomena, single scale models are created; see Fig. 3.1. Ideally, these single scale models are independent and rely only on new messages at specific input ports, while sending messages with observations of their state at output ports. By coupling output ports to input ports using so-called conduits, a multiscale model is formed. Assuming a time-driven simulation approach, each message is associated with a time point, which should be kept consistent between single scale models.

The conceptual framework distinguishes between acyclically and cyclically coupled models. In the former, no feedback is possible from one submodel to the other, while in the latter a submodel may give feedback as often as needed. This distinction has many computational implications, such as the need to keep submodels active in cyclically coupled models, or the recurring and possibly dynamic need for computing resources. MUSCLE 2 focusses on cyclically coupled models by keeping submodels active during the entire simulation, whereas workflow systems tend to focus on acyclically coupled models.

To facilitate consistency, submodels each have a fixed submodel execution loop: consisting of initialisation, a loop with first an observation and then a solving step,

Listing 3.1: Submodel execution loop in MUSCLE 2

```
do {
    init()
    while (!endCondition()) {
        intermediateObservation()
        incrementTime()
        solvingStep()
    }
    finalObservation()
} while (restartSubmodel())
```

and then a final observation. This loop can be restarted as long as a submodel with a coarser time scale provides input for the initial condition. During initialisation and solving steps, only input may be requested, and during the observations, only output may be generated. Although this is the general contract, submodel implementations in MUSCLE 2 may diverge from this loop, for example if it would increase performance.

Submodels should remain independent and as such the data expected by a submodel will not automatically match the observation of another. For this purpose data can be modified in transit, thus implementing scale bridging methods, either by lightweight conduit filters, which change data in a single conduit, or by mappers, which may combine the data of multiple sources or extract multiple messages from a single observation. Finally, the input for a submodel may not be available from another submodel but rather from an external source, or conversely, an observation might only be used outside the model. In that case, terminals may be used: sources to provide data and sinks to extract data.

Each of the concepts mentioned in this paragraph is defined in the multiscale modelling language (MML). In MML these concepts are well-defined and accessible for automated analysis, for example to predict deadlocks or the total runtime of a simulation. The configuration file of MUSCLE 2 and a code outline can be generated from MML.

3.2.2 Library

The library part of MUSCLE 2 consists of Java, C, C++, Python, Fortran APIs and coupling definitions in Ruby. The API's for these languages can each: send and receive arrays, strings, and raw bytes; do logging; and stage in- and output files. Other

Table 3.1: Sending (first row) and receiving (second row) a message in MUSCLE 2 in programming languages Java and C++.

Java	C++

`double[] dataA=new double[100];`
`out("portA").send(dataA);`
	`double* dataA=new double[100];`
	`muscle::env::send(`
	`  "portA", dataA, 100, MUSCLE_DOUBLE`
	`);`
	`delete [] dataA;`

`double[] dataB = (double[])`
`  in("portB").receive();`
	`size_t len;`
	`double* dataB = (double*)`
	`  muscle::env::receive(`
	`  "portB", (void*)0, len, MUSCLE_DOUBLE`
	`  );`
	`muscle::env::free_data(dataB, MUSCLE_DOUBLE);`

programming languages and additional libraries may use the MUSCLE 2 API as long as it can interface with one of the listed programming languages. Send calls have non-blocking semantics whereas receive calls are blocking by default but may be used as non-blocking instead. An example of sending and receiving data with MUSCLE 2 is shown in Table 3.1. The formal submodel loop in Listing 3.1 is advised but not enforced.

The Java API, in addition to the API's of the other languages, allows implementing formal MML constructs such as formal submodels, filters, and mappers, and sending and receiving advanced data structures like Java classes. Because MUSCLE 2 allows multiple languages in a multiscale model, filters and mappers can also be used in models that otherwise do not use Java. In all cases, the API is non-invasive and does not force a certain programming paradigm, which makes it straightforward to incorporate in existing code.

3.2.3 Configuration

The configuration of a multiscale model and the coupling is done in a Ruby file. In this file, submodels and their scales are specified, parametrised, and coupled to each other. Mappers, conduit filters, sources and sinks are also added to the coupling topology here. A conduit can be configured with multiple filters; predefined-filters such

as data compression, or custom filters such as data transformations or conversions. Because the configuration is a Ruby script the coupling topology can be automatically generated, for instance to set up a ring or grid topology, or to read a network from a file.

Below is an example of the configuration of a multiscale model with one macro-model and one micro-model, with a single coupling from macro to micro.

```
# Add the classpath of the submodels, using
# environment variable $MODEL_HOME
add_classpath ENV['MODEL_HOME'] + '/build/classes'

# Create a submodel instance 'macro' with
# implementing Java class 'mypackage.Macro'
macro = Instance.new('macro', 'mypackage.Macro')

# Set the macro timestep to 1 hour, and
# the total simulation time to 1 day
macro['dt'] = '1 hour'
macro['T'] = '1 day'

# For 'micro', use a predefined MUSCLE MPI submodel and
# specify the executable
micro = Instance.new('micro',
                     'muscle.core.standalone.MPIKernel')
micro['command'] = ENV['MODEL_HOME'] + '/build/micro'

# Couple the port 'macroscopicVariable' of macro to
# port 'environmentValue' of micro
macro.couple(micro,
             'macroscopicVariable' => 'environmentValue')
```

3.2.4 Runtime environment

The runtime environment of MUSCLE 2 is designed to be light-weight and portable, and to provide high performance. MUSCLE 2 is supported on Linux and OS X. Data

is transmitted between submodels and mappers, both called instances, using direct and thus decentralised message passing.

Each MUSCLE 2 simulation has a single Simulation Manager and one or more Local Managers, as shown in Fig. 3.2. The Simulation Manager keeps track of which instances have started and what their location is. The Local Manager starts the instances that were assigned to it in separate threads and listens for incoming connections. Instances will start computation immediately but they will block and become idle as soon as they try to receive a message that has not yet been sent, or try to send a message to an instance that has not been started.

A so-called native instance is a compiled instance that runs as a separate executable. Its controller is still implemented in Java and therefore the executable will try to establish a TCP/IP connection with this controller, which will then do all communication with other instances and with the Simulation Manager. A native instance may be serial or use threading, OpenMP, or MPI.

Message-passing mechanisms that are used are shown in Fig. 3.3. Messages within a Java Virtual Machine are sent using shared memory. To insure independence of data between instances, the data is copied once before it is delivered from one instance to the other unless otherwise specified. Messages between Local Managers and between the Local and Simulation Managers are sent over TCP/IP, which is available everywhere and inherently allows communication with between machines. The MessagePack serialisation library [104] is used for these communications because of its efficient packing. The connection between a native instance and its controller uses the XDR [131] serialisation library because it is already installed in most Unix-like systems.

3.2.5 Cross-cluster computing

Because MUSCLE 2 uses TCP/IP for message passing between instances, it can communicate over the internet and within clusters. However, most HPC systems prevent direct communication between submodels running on different clusters. Therefore, MUSCLE 2 provides the user space daemon MUSCLE Transport Overlay (MTO). It runs on an interactive node of an high-performance cluster and will forward data from MUSCLE 2 instances running on the cluster to the MTO of another cluster, which will forward it to the intended MUSCLE 2 instance. By default, it does this with plain non-blocking TCP/IP sockets, but it can also use the MPWide 1.8 [59] library.

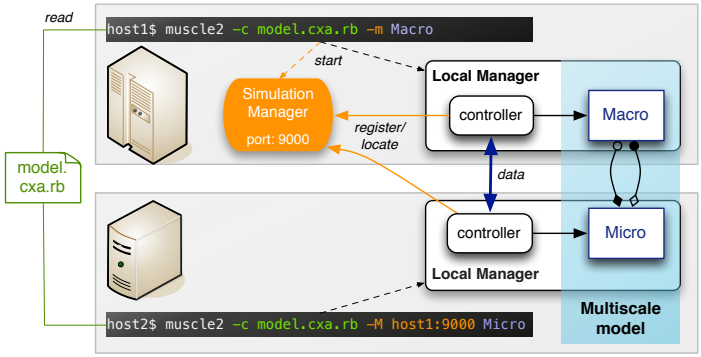
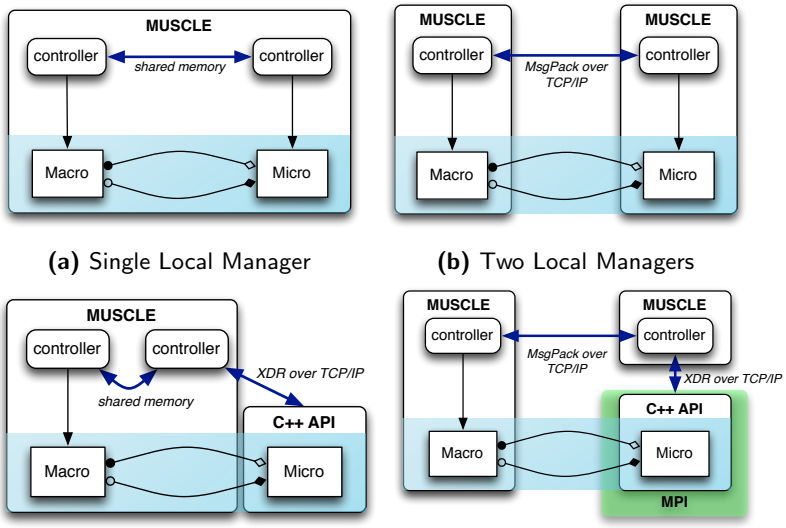


Figure 3.2: An example of the MUSCLE runtime environment, explained in Section 3.2.4, doing a distributed execution of the multiscale model described in file `model.cxa.rb` on machines `host1` and `host2`. The register and data arrows are both TCP/IP connections. The Macro and Micro rectangles make up a multiscale model.



(a) Single Local Manager

(b) Two Local Managers

(c) C++ with single Local Manager

(d) MUSCLE 2 with MPI

Figure 3.3: Different ways in which MUSCLE 2 can be executed. A box with the MUSCLE label indicates a Local Manager. Blue arrows indicate communication between instances. They are labeled with the means of communication.

Table 3.2: Computing resources used in performance testing. Cores are listed per node.

Name	Infrastructure	Location	Processor	Cores
iMac	local desktop	Amsterdam, The Netherlands	Intel i3 3.2 GHz	2/4*
Zeus	PL-Grid, EGI	Krakow, Poland	Intel Xeon 2.4 GHz	4
SuperMUC	PRACE Tier-0	Munich, Germany	Intel Xeon 2.7 GHz	16/32*
Huygens	PRACE Tier-1	Amsterdam, The Netherlands	IBM Power6 4.7 GHz	32/64*
Cartesius	PRACE Tier-1	Amsterdam, The Netherlands	Intel Xeon 2.7 GHz	12/24*
Gordias	local cluster	Geneva, Switzerland	Intel Xeon 2.4 GHz	8
Scylla	local cluster	Geneva, Switzerland	Intel Xeon 2.4 GHz	12

*Two HyperThreads per core

MPWide's goal is to optimise the performance of message-passing over wide-area networks, especially for larger messages.

Because the MUSCLE 2 instances that make up a distributed simulation have to run concurrently and their in- and output files have to be managed, cross-cluster simulations are tedious to do manually. For this reason the MUSCLE 2 framework was integrated with the QosCosGrid (QCG) middleware stack [27]. The QCG middleware stack offers users advanced job management and resource management capabilities for e-Infrastructure. It will manage the execution of a distributed MUSCLE 2 simulation from a central location, reducing the input and management needed from the user.

3.3 Performance

The main benefit of MUSCLE 2 is the library and coupling environment that it provides. However, for many if not all multiscale simulations, performance is equally important. The performance of MUSCLE 2 has two aspects: the overhead it introduces and the messaging speed that it provides. These were measured on four resources: an iMac (a local desktop machine), Zeus (a community cluster accessible through EGI or PL-GRID), Huygens (a PRACE Tier-1 resource from SurfSARA, the Netherlands) and SuperMUC (a PRACE Tier-0 resource from Leibniz-Rechenzentrum, Germany). See Table 3.2 for their details.

3.3.1 Overhead

With MUSCLE's runtime overhead figures, a user can estimate what the impact of MUSCLE will be on the execution time and memory for a given multiscale model.

To test the overhead we will start a varied number of submodels and conduits, to evaluate their impact on CPU and memory usage. The overhead will be measured on an iMac and on SuperMUC.

To test the overhead in execution time, MUSCLE is started with 30 different submodel counts n, from 1 to 1000, and 36 different conduit counts m, from 0 to 50.000. The submodels are created in a configuration script, in which each submodel adds a conduit to each of the following m/n submodels, wrapping around to the first submodel if there are less than m/n succeeding submodels. Once the simulation has started, each submodel sends and receives one empty message through each conduit, and then exits. This way, all submodels must be active simultaneously for a small amount of time, like they would be in a normal simulation. Since a submodel with native (C/C++/Fortran) code needs to start an additional executable, it is measured separately. The amount of time spent on Java garbage collection is not separately measured. Software versions on iMac are Java 1.7.0_7 and Ruby 1.9.3; on SuperMUC they are Java 1.6.0_32 and Ruby 1.8.7. The minimal overhead a is determined by taking the minimum value encountered. The additional runtime overhead b per submodel and c per conduit is determined by fitting the data to the equation $a+bn+cm$, where n are the number of submodels and m are the number of conduits. The additional runtime per native submodel was fitted to a linear curve separately. The minimum memory overhead was taken as the memory of running with one submodel, all other values were separately fitted to a linear curve.

The results for this experiment are listed in Table 3.3. With the highest number of submodels and conduits (1000 and 50.000 respectively), execution took 7.1 seconds on the iMac and 6.6 seconds on SuperMUC; the lowest runtimes were 0.68 and 1.2 seconds respectively. For most if not all multiscale simulations, even 7.1 seconds overhead will not be significant compared to the overhead of running the simulation, and for multiscale simulations with less than 10 submodels, the overhead will be close to a second.

The memory consumption was measured in a similar way as runtime overhead, except here ten submodel counts from 1 to 1000 where used, and separately thirteen conduit counts from 0 to 50,000, each started four times. The Java Virtual Machine of the Local Manager was set up with an initial heap size of 1 GB and with a maximum heap size of 3 GB. Since MUSCLE uses Java and Ruby, exact memory consumption will differ per execution and it will include free space that their respective runtime en-

Table 3.3: Runtime and memory consumption[2] of MUSCLE, on a local iMac and the PRACE machine SuperMUC (see their details in Table 3.2). Entries marked '–' were not measured. We assume that the total memory consumption on both machines is similar, since they both use 64-bit Intel processors. The first row (Overhead) indicates the overhead of MUSCLE without starting any submodels, the other rows show additional overhead to this baseline.

	iMac runtime	**SuperMUC runtime**	**iMac memory**
Overhead	0.77 s	1.2 s	73 MB
Per submodel	1.6 ms	1.6 ms	168 kB
Per local conduit	0.11 ms	0.10 ms	3.4 kB
Per port with remote coupled port	–	–	1.1 MB
Per native submodel	24 ms	–	1.7 MB

gines have reserved. However, with enough memory allocation a trend does emerge. If multiple MUSCLE instances are started for a single multiscale model, additional buffers need to be reserved for communicating with other Local Managers. Therefore ports that are coupled to a port of a submodel with another Local Manager are measured separately, as are submodels with native code.

The results are listed in Table 3.3. With these figures, and taking into account the memory consumption of the individual submodels, a user can estimate how many submodels will fit in the memory of a single machine. As a result of the allocated buffers, ports coupled to a port on an other Local Manager take a large amount of memory, 1.1 MB. Similarly, a native executable linked to MUSCLE uses at least 650 kB of memory, and in Java an additional serialisation buffer is allocated. On a machine with 4 GB of memory per core, each core could accommodate up to 20,000 submodels with 10 local conduits each, up to 350 submodels with 10 remote conduits, or up to 300 native submodels with 10 remote conduits. In most scenarios this is more than sufficient, and the number of submodels will instead be limited by the computational cost of the submodel code.

[2]Stated memory sizes are multiples of 1024 (kilo)bytes.

3.3.2 Message speed

The performance of MUSCLE communication is compared with approaches that modellers would usually use for composite models. The two most prevalent methods of communication of our current users are file-based or MPI-based. The former is often used to couple different codes together, whereas the latter is used to form fast monolithic codes. For remote communication, GridFTP is a popular alternative and MPWide is a well-optimised one. We will compare these methods with the communication speed offered by MUSCLE 2. Both latency and throughput of the methods will be computed.

Single machine

For the local communications we will compare speeds of file copy, MPI, MUSCLE 1 and MUSCLE 2. Each of the tests is done with message size 0 kB and 2^i kB, with i ranging from 0 to 16, which is up to 64 MB. Since MUSCLE 1 will not send messages larger than 10 MB, its measurements are limited to i ranging from 0 to 13 (8 MB). Per message size, a message is sent back and forth 100 times, so it makes 100 round trips. The time to send one message of a certain size is calculated as the average over the round trip times, divided by two. The latency is calculated as the minimum time to send a message. The message times are then fitted to a linear curve $ax + b$ for message size x, where throughput is calculated as $\frac{1}{a}$ and b is taken as the latency.

For applications without a coupling library, a simple way to transfer data from one process to another is to write to a file which another process may read. The operating system might cache this file so that the read operation is fast. This scenario is simulated by creating files as messages. One round trip is taken as copying a file and copying it back using the systems file copy, which is equivalent to writing and reading a file twice.

For a monolithic model, possibly with multiple substructures or threads, MPI is a well-known and very fast option. This paradigm, however, gives none of the plug and play advantages that MUSCLE 2 has, nor does it keep time in sync between submodels, nor is it easy to combine resources of different providers. In our experiment, messages are sent by one MPI process, then received and sent back by another with the same executable.

To test MUSCLE 2, first we take the situation that all instances have a Java implementation and a single machine is sufficient to run them. As described in Section 3.2.4,

messages are then sent through shared memory. Next, we take two MUSCLE 2 processes that communicate with TCP/IP, for when a user wants to prioritise one process over the other, for instance. Finally, we take two instances that both have a C++ implementation.

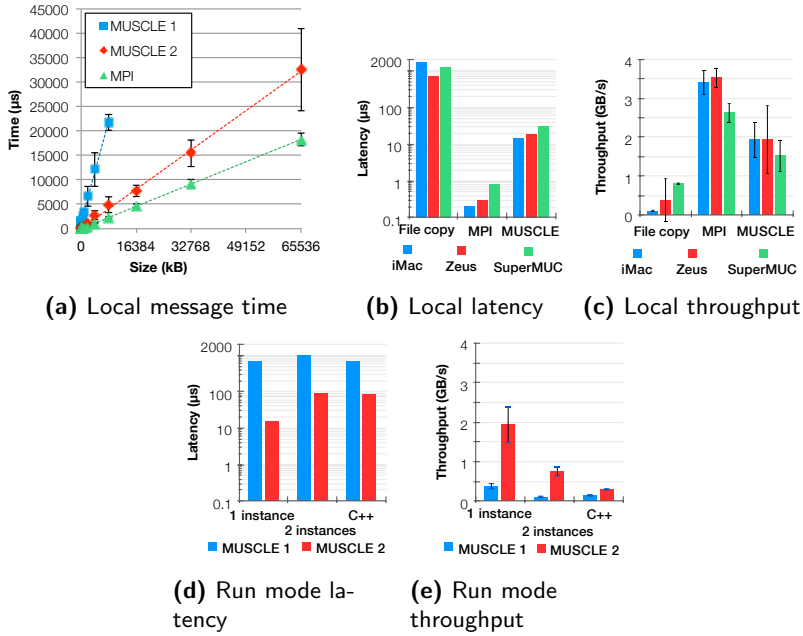
The file copy, MPI and MUSCLE 2 scenarios are tested on the iMac (local desktop), Zeus (cluster), and SuperMUC (supercomputer). MUSCLE 1 is only tested on the iMac due to portability issues.

The results are plotted in Fig. 3.4. The standard deviation for the latency is very low and is not shown. Obviously, copying data has a higher latency and lower throughput than the alternatives. The latency of MPI is clearly the lowest and MPI has the highest throughput as well, which would be expected because it uses highly optimised native code. MUSCLE falls in the mid-category, and is thus a serious contender if neither a monolithic nor a file-based simulation is desired. These results do signal that for optimal performance of a very tightly integrated code, MPI could be preferred over MUSCLE 2. Of course, this MPI code can then be used in MUSCLE 2 as a single submodel, so that MUSCLE can take care of starting the submodel and coupling it with other codes.

Comparing the run-modes of MUSCLE, and MUSCLE 1 and MUSCLE 2, a few remarks can be made. First, the latency of C++ is lower than having two Local Managers, which is surprising: with C++ a message is first serialised with XDR, sent, passed through shared memory in Java and then serialised again to be sent to another C++ program. With two Local Managers, a message is serialised once with MessagePack and directly used. So although the throughput of MessagePack is higher, its latency is worse. Second, MUSCLE 1 falls far behind MUSCLE 2 in all cases, since it uses the JADE system to send messages and overall has a less optimised code. That the performance of MUSCLE 1 is most similar to MUSCLE 2 in the C++ scenario, is because the Java Native Interface (JNI) transfers data between Java and C++ faster than TCP/IP sockets can. JNI was removed in MUSCLE 2 due to the portability issues that it caused, see Appendix B.3 for the details.

Distributed computing

Besides local message speed, distributed message speed is also important for computing on large infrastructures. Although the main bottleneck will usually be the available network bandwidth, software does have an influence on message speed. In this section



(a) Local message time **(b)** Local latency **(c)** Local throughput

(d) Run mode latency **(e)** Run mode throughput

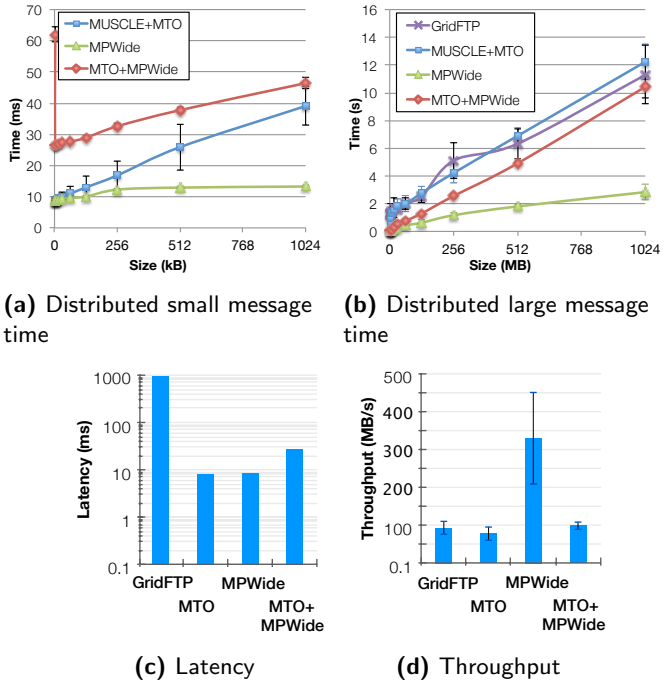
Figure 3.4: The performance of the communication methods described in Section 3.3.2. (a) shows a plot of the time to send a single message, along with linear fit. (b) and (c) show the performance of sending a message within a machine, for three machines (the latency is averaged as it showed little variation). (d) and (e) show the performance of sending messages on a local machine with MUSCLE 1 and MUSCLE 2, by starting a model: in a single MUSCLE instance; with two coupled MUSCLE instances; or, with C++ submodels. The standard error of the latency measurements was negligible so it is only shown for the throughput.

we will compare the speed of three possible technologies to do wide area network communication: MPWide 1.8, GridFTP 0.8.9, and MUSCLE 2 with the MTO. MPWide is designed specifically for optimally making use of the available bandwidth by using packet pacing, multiple streams per connection and adapted buffer sizes. GridFTP [57] is a dedicated file transfer service run by EGI and PRACE sites. MUSCLE uses the MTO, which by default uses a single plain TCP/IP socket per connected MTO but can also be used in conjunction with MPWide.

Each test was performed between a PRACE Tier-1 site Cartesius in Amsterdam and the PRACE Tier-0 site SuperMUC in Garching, Munich (more details in Table 3.2). They send a message from Amsterdam and back again, using message sizes

0 kB and 2^i kB, with i ranging from 0 to 20, which is up to 1 GB. For each message size up to 1 MB, hundred messages were sent, for messages ranging from 2 MB to 1 GB ten messages were sent. The TCP/IP route from Cartesius to SuperMUC uses the high-speed PRACE network. The average ping time over 50 consecutive pings on this route was 15.2 ms.

In all applications the standard settings were used. For MPWide the number of streams must be specified and was set to 128 streams. Although GridFTP can open multiple TCP streams for a transfer, firewall settings prevented it to do so from SuperMUC to Cartesius, so in this experiment it used only one.



(a) Distributed small message time

(b) Distributed large message time

(c) Latency

(d) Throughput

Figure 3.5: The performance of sending a message between Huygens and SuperMUC. (a) shows the time to send a message on a kilobyte scale, and excludes GridFTP which fluctuates around 890 ms. (b) shows the time to send a message on a megabyte scale. The other two plots show the fitted values of the data.

The results of the test are shown in Fig. 3.5. For very small messages both MUSCLE 2 with the MTO and MPWide come very close to the ping time, adding up to 2

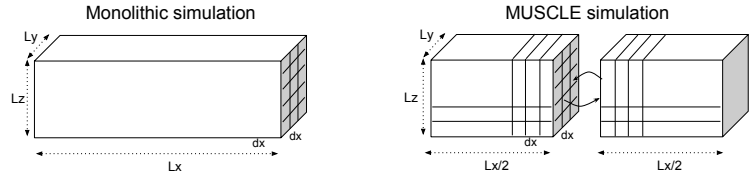


Figure 3.6: How a 3D cavity can be split into equal parts for use with MUSCLE, where L_x is its length, L_y its width and L_z its depth, and dx the resolution at which it is resolved.

ms. When the MTO uses MPWide internally the latency goes up considerably because it uses an additional management layer. GridFTP has to do a certificate hand-shake before when connecting, which takes significantly longer at about 890 ms. With large messages its performance is much better, at 90 MB/s, although it does show an occasional bump when the hand-shake can not be processed immediately. MUSCLE 2 with the MTO did a bit worse and MUSCLE 2 with the MTO using MPWide did a bit better. Plain MPWide performance was much better than the other methods for messages larger than 128 kB. This indicates that further efforts to integrate the MTO and MPWide may be beneficial.

3.4 Use cases

To show the real-time usage of MUSCLE 2 as well as its practical performance we will show how it is applied to a multiscale model of a canal system and specifically to the submodel of one canal section. In the next chapter, a multiscale model of in-stent restenosis shows the heterogeneity of submodels that can be coupled.

3.4.1 Hydrology application

An optimal management of rivers and waterways is a necessity in modern society to ensure an adequate supply of water, in particular for agriculture, electricity production or transportation [102]. An important requirement is to control the water level and sediment transport in populated areas [97]. These problems can be addressed through computer simulation in combination with optimisation methods.

Many of such hydrology problems can be implemented using a "Lego based philosophy" [16, 17], where river or water sections are modelled by submodels and con-

nected with mappers, based on the topology of existing canal systems. A submodel can for instance implement a 3D Free Surface (3DFS) model and be connected to a 1D shallow water submodel. Because of their different resolution and time step this gives a multiscale system. The decomposition into submodels allows a distributed execution, which may be necessary for larger canal systems.

Our use case consists of a 3D cavity flow problem solved with the Lattice Boltzmann (LB) [130] numerical method. The submodels are implemented with the Palabos toolkit[2], which uses MPI for parallelisation. The aim is to evaluate the time overhead induced by the use of the MUSCLE API when performing distributed computations of hydrodynamical problems. As illustrated in Fig. 3.6, the computational domain (here a 3D cavity) is divided across several parallel clusters and information should be exchanged between them at each iteration. This use case itself has full scale overlap but will be coupled to different time scales when a canal system is simulated.

Listing 3.2: Pseudo-code of the cavtiy3d example.

```
1 f_init()
2 for (iterations < maxIteration) {
3    collideAndStream()
4    gatherBoundaryData()
5    sendReceiveBoundaryData()
6    updateBoundaryData()
7 }
```

Listing 3.2 gives the pseudo-code of the algorithm used by the numerical method. During each loop iteration (line 2), the submodel computes the flow on line 3 using the parallel Lattice Boltzmann method. On Line 4, each submodel retrieves boundary data from all MPI processes in the same job and submodel. Line 5 establishes the coupling between submodels. In this case, the submodel sends and receives boundary data using the MUSCLE API hidden in the `sendReceiveBoundaryData()` function. On line 6, each section updates its boundary according to the data received from the other submodels.

To show the performance of MUSCLE 2 when it is used in an actual problem, we will consider the performance of the 3D cavity submodel described above. Our benchmark will consist of running a monolithic code first and comparing its run-

[2]Palabos: `http://www.palabos.org/`

time with using two MUSCLE submodels. A detailed treatment has been made by Ben Belgacem et al. [17]; here we show some results obtained by using more CPUs.

The computational domain of the canal we will use, as depicted in Fig. 3.6, has a length L_x of 13000 metres, a width L_y of 40 metres and a depth L_z of 10 metres. The spatial resolution Δx may vary and will determine the problem size: decreasing Δx implies increasing the domain size.

For the benchmark, we will evaluate three scenarios:

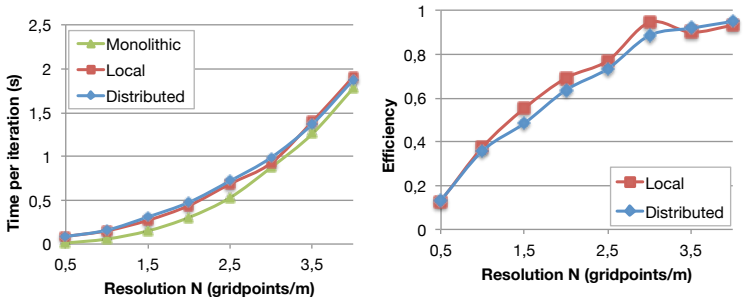
1. a monolithic simulation of the canal on a single cluster;
2. a simulation with two canal submodels on the same cluster, coupled using MUSCLE; and
3. a simulation with two coupled canal submodels on different clusters, coupled using MUSCLE 2.

The first case shows what the performance of a usual monolithic model with MPI is, the second what the cost is in splitting that into multiple parts using MUSCLE, and the third what the cost is of distributing it with MUSCLE.

The execution time of these scenarios is indicated T_{mono}, T_{local}, and T_{distr}, respectively. In scenarios 2 and 3, the canal section computed in scenario 1 is split equally amongst the submodels called left and right. Each simulation carries out 100 iterations, and this is repeated three times. We varied the number of grid points per metre $N = \frac{1}{\Delta x}$ from 0.5 to 4, with a step size of 0.5. For the total domain this means varying the problem size from under 820 thousand grid points to over 340 million points, scaling with N^3. The MUSCLE communication volume, however, only scales with N^2, so computation will dominate computation for increasing N.

The simulations are run on the Gordias and Scylla clusters (for their details see Table 3.2). The monolithic execution is done with 100 cores of the Gordias cluster. Likewise, the MUSCLE execution is done with 100 cores, but here the left and right section run on 50 cores each. The local MUSCLE execution is run on Gordias whereas the in distributed one, both clusters are used. In the local execution we first ran the MUSCLE Simulation Manager in a separate node so that it had a fixed address before the job started. In the distributed scenario, QCG-Broker takes care of queuing the jobs and starting the Simulation Manager.

Fig. 3.7a shows the results of the benchmark of T_{mono}, T_{local} and T_{distr} on Gordias cluster. We measured the average time per iteration. If we compare T_{mono} and T_{local},



(a) Runtime

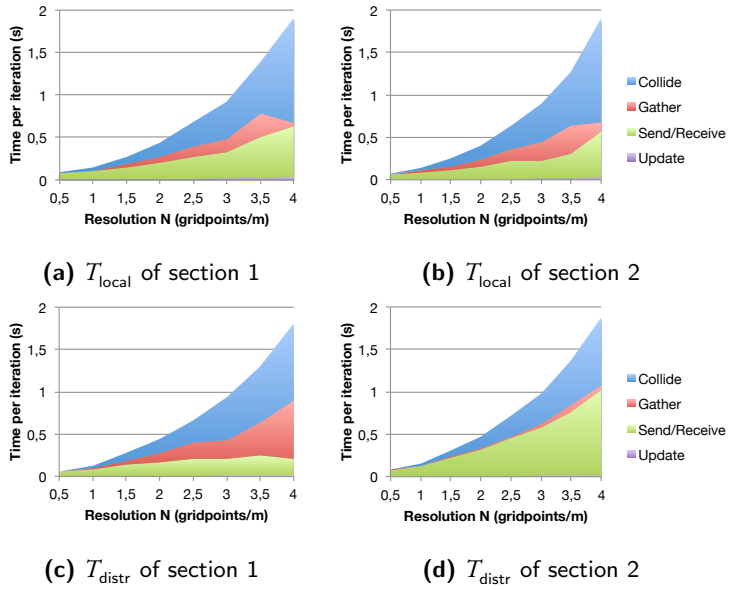
(b) Efficiency compared to monolithic runtime

Figure 3.7: The performance of the three execution scenarios of the cavity 3D model, for the number of grid points per metre N, as described in Section 3.4.1.

we see that the difference between execution times varies very little over all values of Δx. The main bottleneck seems to be a fixed synchronisation overhead due to waiting for messages between submodels.

Regarding the distributed execution (Fig. 3.7b), the efficiency values $\varepsilon_{\text{distr}} = T_{\text{mono}}/T_{\text{distr}}$ and $\varepsilon_{\text{local}} = T_{\text{mono}}/T_{\text{local}}$ show the same behaviour; i.e, we observe a large communication ratio with small values of N, and vice versa. This goes to the extent that for $N = 4$, using only MPI is just 5% more efficient than using MUSCLE. $\varepsilon_{\text{distr}}$ is smaller than $\varepsilon_{\text{local}}$ for smaller problem but for large values of N $\varepsilon_{\text{distr}}$ is slightly higher, which can be explained if we look at the detailed plots.

The runtimes of the two sections on the Gordias cluster, per operation of the pseudo-code 3.2, are very similar, as shown in Fig. 3.8a and Fig. 3.8b. For large N, the fraction of time spent actually calculating increases steadily. For smaller N, however, most of the time is spent in waiting for messages from the other submodel, so if one submodel was slower then the other would have to wait until it was finished and vice-versa. In the distributed experiment however, the submodel on Scylla (Fig. 3.8d) was computed consistently faster, which means the submodel on Gordias (Fig. 3.8c) needs to wait far less. This gives a lower average time per iteration for situations that depend more on computational time than on communication time.



(a) T_{local} of section 1 **(b)** T_{local} of section 2

(c) T_{distr} of section 1 **(d)** T_{distr} of section 2

Figure 3.8: The runtime of different operations in the local and distributed cases, where (a)–(c) run on the Gordias cluster and (d) runs on the Scylla cluster, and (a)–(b) are run concurrently as are (c)–(d). The operations match the pseudo-code in Listing 3.2: `collideAndStream()` [Collide]; `getBoundaryData()` [Gather]; `sendReceiveBoundaryData()` [Send/Receive]; and `updateBoundaryData()` [Update].

3.5 Conclusions

In this chapter, we have introduced and discussed the component-based and flexible design of MUSCLE 2, and its distributed computing capabilities. It is based on a general approach to multiscale modelling and simulation [22, 67, 70] combined with the multiscale modelling language [22, 47]. Because of its modular setup, clearly separating API, coupling, and runtime environment, users can modify parts of a multiscale model without affecting the rest. A multiscale model implemented with MUSCLE 2 can be executed on distributed computing resources at any stage. Moreover, submodel code written in Java, C, C++, Python, or Fortran, and using serial code, MPI, OpenMP, or threads can freely communicate with other submodels using different technologies.

The overhead of starting MUSCLE 2 for multiscale models with a reasonable amount of submodels is shown to be low, both time- and memory-wise. For local computing MUSCLE 2 is shown to be more efficient than file based message passing, but it has a factor two lower throughput than MPI and up to 30 μs higher latency. For parts of a multiscale model where MPI is better suited, such as performing a lattice method or doing agent based simulations, MUSCLE 2 can simply run that part as a submodel with MPI, and the multiscale model will still have the advantages of flexible coupling and execution.

For distributed computing, the MUSCLE Transport Overlay transfers data from one high-performance computing centres to another. Its efficient transfers easily surpass GridFTPs speed for smaller messages and give performance similar to GridFTP for large messages. Using MTO with MPWide gives slightly better performance on the high-speed PRACE network, but plain MPWide still much faster, so the integration between the MTO and MPWide will be further examined.

For a canal system model, MUSCLE 2 makes it easier to generate canal topologies by flexible coupling and being able to distribute different parts of the canal system. Moreover, for canal sections with sufficiently large problem sizes, the performance of MUSCLE 2 is competitive with using a single monolithic code. It will need distributed computing for larger problems when a local cluster does not provide enough resources; this turns out not to be very detrimental to performance.