Supercomputing Frontiers and Innovations

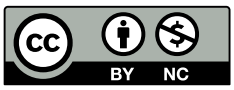
2019, Vol. 6, No. 4

Scope

- Enabling technologies for high performance computing
- Future generation supercomputer architectures
- Extreme-scale concepts beyond conventional practices including exascale
- Parallel programming models, interfaces, languages, libraries, and tools
- Supercomputer applications and algorithms
- Distributed operating systems, kernels, supervisors, and virtualization for highly scalable computing
- Scalable runtime systems software
- Methods and means of supercomputer system management, administration, and monitoring
- Mass storage systems, protocols, and allocation
- Energy and power minimization for very large deployed computers
- Resilience, reliability, and fault tolerance for future generation highly parallel computing systems
- Parallel performance and correctness debugging
- Scientific visualization for massive data and computing both external and in situ
- Education in high performance computing and computational science

Editorial Board

Contents



DOI: 10.14529/jsfi190401

An Energy-aware Dynamic Data Allocation Mechanism for Many-channel Memory Systems

Masayuki Sato[1], *Takuya Toyoshima*[1], *Hikaru Takayashiki*[1], *Ryusuke Egawa*[1], *Hiroaki Kobayashi*[1]

A modern memory system is equipped with many memory channels to obtain a high memory bandwidth. To take the advantage of this organization, applications' data are distributed among the channels and transferred in an interleaved fashion. Although memory-intensive applications benefit from a high bandwidth by many memory channels, applications such as compute-intensive ones do not need the high bandwidth. To reduce the energy consumption for such applications, the memory system has *low-power modes*. During no memory request, the main memory can enter these modes and reduce energy consumption. However, these applications often cause intermittent memory requests to the channels that handle their data, resulting in not entering the low-power modes. Hence, the memory system cannot enter the low-power modes even though the applications do not need the high bandwidth. To solve this problem, this paper proposes a dynamic data allocation mechanism for many-channel memory systems. This mechanism forces data of such applications to use the specified channels by dynamically changing the address-mapping schemes and migrating the data. As a result, the other channels to which the data are not allocated can have a chance to enter the low-power modes for a long time. Therefore, the proposed mechanism has the potential to reduce the energy consumption of many-channel memory systems. The evaluation results show that this mechanism can reduce the energy consumption by up to 11.8% and 1.7% on average.

Keywords: DRAM, main memory, low-power mode, address-mapping scheme, energy consumption.

Introduction

Modern microprocessors have improved their performances by increasing the numbers of cores significantly [3]. As a result, a high memory bandwidth is required by a microprocessor and applications executed on it [16]. On the other hand, the speed of Dynamic Random Access Memory (DRAM), which is used as a main memory, has modestly improved. Due to the performance gap between microprocessors and main memories, the main memories cannot supply the data at enough speed to the microprocessor. Therefore, a main memory has been a bottleneck in the performance of a modern computing system. This problem is well-known as the "Memory Wall" problem [18].

To cope with such a situation, the latest computing systems have a main memory system with many memory channels to obtain the higher memory bandwidth. Such a *many-channel memory system* is widely employed in industrial microprocessors from personal computers to supercomputer systems. For example, a memory module with 3D-integration technologies, High Bandwidth Memory (HBM), has 8 channels physically, and can act as a 16-channel module in the pseudo-channel mode [13]. Since six HBM modules can be integrated with a microprocessor on a silicon interposer [19], it is possible for a single memory system to have 48 to 96 channels.

On the other hand, not all applications require such a high memory bandwidth. As the diversity among applications increases, the applications could be categorized into memory-intensive and compute-intensive applications. The compute-intensive applications do not always need high bandwidths. For these applications, such a high bandwidth memory has a little performance im-

[1]Tohoku University, Sendai, Japan

pact and causes a high energy consumption. To reduce the energy consumption of the memory system in compute-intensive applications, DRAM-based memory systems have several low-power modes. These modes can reduce the power consumption by deactivating peripheral circuits of DRAM chips when the compute-intensive applications do not need the memory performance.

However, to take the advantages of using many channels simultaneously and get their full bandwidth, the data of an application are distributed among all the memory channels in an interleaved fashion, determined by *an address-mapping scheme*. Since the data placement for any applications is treated under a single address-mapping scheme, the data of the compute-intensive applications are also distributed. If these applications intermittently cause memory requests to all the channels, memory systems cannot enter the low-power modes, although they do not need to use the whole bandwidth. Therefore, address-mapping schemes should be appropriately selected so that the memory system can reduce the energy consumption by aggressively using the low-power modes.

To solve this problem, this paper proposes an energy-aware dynamic data allocation mechanism for many-channel memory systems. This mechanism dynamically switches two address-mapping schemes based on the access frequencies. One scheme is to distribute the data among channels, and the other scheme is to gather the data into specific memory channels. When the applications do not need to use all of the channels for the highest bandwidth, this mechanism selects the scheme that gathers the data accessed frequently into limited memory channels. By selecting this scheme, the access frequencies to the other channels can be reduced, and these channels can get more opportunities to enter the low-power modes. Therefore, this mechanism has the potential to reduce the energy consumption of many-channel memory systems. Note that, if the proposed mechanism switches the address-mapping schemes without caring the data, a memory system cannot correctly provide the data that are stored based on the address-mapping scheme used before the switching. To avoid such a situation, the proposed mechanism migrates data stored by the previous scheme so that they can be accessed by the current scheme. Therefore, the proposed mechanism can continue to correctly process the memory requests.

The rest of this paper is organized as follows. Section 1 explains the low-power modes and discusses the effectiveness of the low-power modes. Section 2 discusses address-mapping schemes to change the number of accessed channels, and proposes a dynamic mechanism that changes these schemes during an execution to reflect the demand for the memory bandwidth of applications. Section 3 shows the evaluation results that support the effectiveness of the proposed mechanism. The final section concludes this paper.

1. Power Management of Memory Systems

1.1. Basic Organization of a DRAM-based Memory System

Figure 1 shows a block diagram of a typical DRAM-based memory system. The system consists of channels, ranks, bank groups, and banks hierarchically. Note that the existence of these hierarchical layers and the number of components in each layer depend on the DRAM standards and the configuration of the memory system [7, 11]. A single channel contains some ranks, and each rank can be selected by chip-select lines and/or chip-ID lines. The rank selected by these lines accepts the commands on the shared buses. Each rank contains banks that are arrays of DRAM cells, or sometimes bank groups, which are groups of multiple banks. In each bank, DRAM cells are organized by rows and columns in DRAM arrays. The DRAM array is

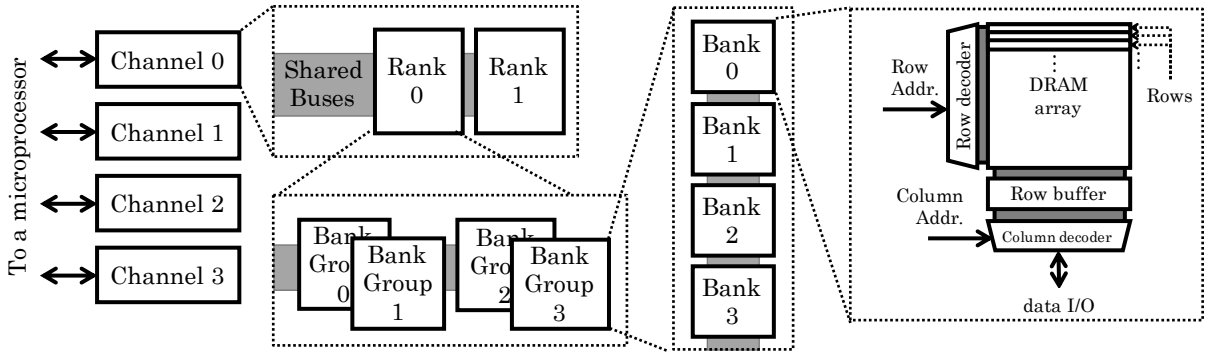


Figure 1. Overview of a memory system

accompanied by peripheral circuits to read/write data from/to a single row, and provide data in columns to the microprocessor.

The procedure to access the data is summarized in the following three steps. All the commands for these steps pass through the shared buses. The first step is *precharge*. This step sets up banks that include the accessed data. In this step, a row buffer adjoining each bank is cleared with writing its data back to the DRAM cells, and the banks and their peripheral circuits are prepared for the next step. The next step is *row activation*. In this step, one of the rows in the bank is specified by the row decoder, and then the data in the row are read out to the row buffer. The third step is *column read/write*. In this step, the column in the row buffer is specified. In the case of a read request, the column data are read out from the row buffer to the shared data bus. In the case of a write request, the data sent from the shared bus are stored in the target column in the row buffer.

To access the data based on these steps, it is requested to determine which bank, row, column, and rank should be accessed. To this end, the physical address included in a memory request is used. From the bit sequence of the physical address, the IDs of these hierarchical components are extracted as subsets of the bit sequence. A rule that determines how these IDs are taken from the physical address is called an *address-mapping scheme*.

1.2. Low-power Mode

Figure 2 shows the state transition diagram of a DRAM chip. Each state corresponds to each mode that the DRAM chip can transit. Idle mode is an initial state. If a bank receives the row-activation command (ACT), the bank transits to the state *active*. In this state, the bank can accept read/write operations issued by the column read and write commands. On the other hand, when the bank receives the precharge command (PRE), the state of the bank returns to idle. In the case where all the banks in the rank become the Idle state and receive the refresh command (REF), the rank starts to refresh data to avoid the DRAM cells from losing the data by leakage current.

For reducing the energy consumption of DRAM-based memory systems, the DRAM standards include the low-power modes. A rank can enter the low-power mode under the conditions that there is no waiting command based on the memory request to the banks, and the rank receives command *PDX* or *SRX*. As shown in Fig. 2, there are several low-power modes, e.g., the active power-down mode, the precharge power-down mode, and the self-refresh mode. The mode that the rank can enter is different depending on the original state. The deeper low-power mode is more effective to reduce the power consumption.

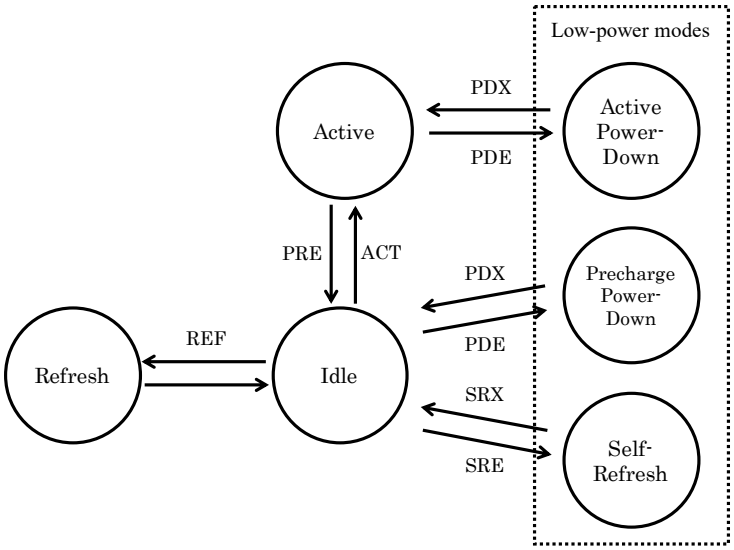


Figure 2. State transition diagram of power modes

1.3. Limitation of the Deep Low-power Mode

To reduce the power consumption of the memory system, the low-power mode should be appropriately used. This is because the energy consumption may increase due to the latency for the recovery from the low-power modes to the active mode. To obtain the target data in the rank in the low-power modes, the memory requests have to wait for the recovery of the ranks from the low-power modes. The recovery process needs to enable the power supply for the disabled circuits, resulting in a certain latency. If accesses invoke the recovery process frequently, the accumulated latency may cause a significant performance penalty for the executed application. As a result, the execution time of the application and the energy consumption may increase even if the power consumption decreases. Among the three low-power modes, the self-refresh mode is the most effective to reduce the power consumption, but the recovery latency is also the largest among them, which results in the large performance penalty [14].

To evaluate how the low-power modes, especially the self-refresh mode, can be exploited in a modern memory system, the preliminary evaluation is conducted by simulation. The target system has a 4GB HBM2 memory module for the main memory. The detailed parameters of the system are shown in Tab. 1. The latency parameters of HBM2 refer to [4], and the current parameters are based on those of a DDR4 module that has the same voltage and clock frequency [11]. In this simulation, the gem5 simulator system [1] is used. Applications executed on the simulator are selected from the SPEC CPU2006 benchmark suite. Each application is executed by three billion instructions after skipping the first one billion instructions.

The details of the self-refresh mode are as follows. Generally, to suppress the performance penalty, the rank can enter the self-refresh mode after a certain idle period has continued. In addition, one normal refresh operation should be done before moving to the self-refresh mode [11]. Therefore, in this simulation experiment, the rank enters the self-refresh mode after the conditions that a certain period without access is elapsed, and one refresh operation completes. The period is empirically determined as 78,000 cycles so that the performance degradation due to the self-refresh mode becomes less than 1%.

Figure 3 shows the evaluation results of the usage of the self-refresh mode. The left vertical axis and the bar graphs show a ratio of time in the self-refresh mode to the total execution time.

Table 1. Simulation parameters

Core	4GHz, 8 instruction width, out-of-order
L1 I-cache	8 ways, 32KB, 2-cycles latency
L1 D-cache	8 ways, 32KB, 2-cycles latency
L2 cache	8 ways, 1MB, 20-cycles latency
Main memory	HBM2, 4GB, 8 channels

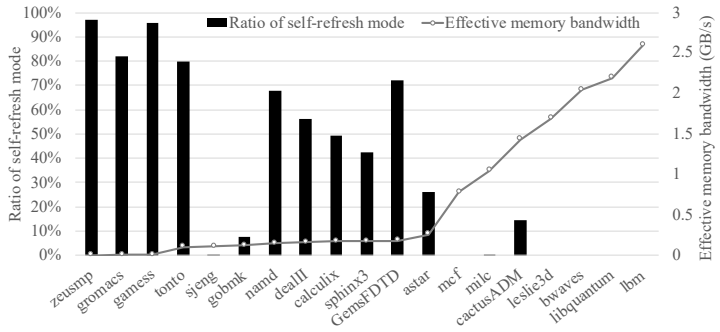


Figure 3. Ratio of self-refresh mode and effective memory bandwidth

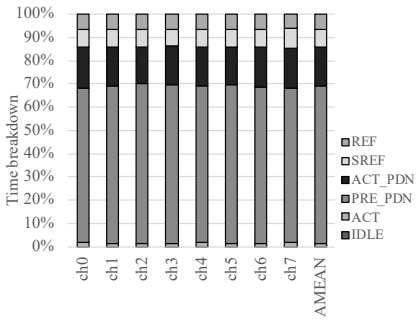


Figure 4. Ratio of times in various modes to the total time (*gobmk_13x13*)

The right vertical axis and the line graph indicate the effective memory bandwidth in the case without the self-refresh mode. All of these values are averaged by those of all the eight channels. Figure 3 shows the trend that the ratio of the self-refresh mode becomes smaller as the sustained memory bandwidth increases. This trend is very intuitive. As the number of memory requests increases, the idle time between a request and its subsequent one becomes shorter. Hence, the memory system cannot easily enter the self-refresh mode.

Focusing on each benchmark, the ratios of the self-refresh mode in some benchmarks are very low, 0.08% in *sjeng* and 7.8% in *gobmk*, respectively. Each of the benchmarks cannot enter the self-refresh mode at all while its effective bandwidth is low enough compared with the other applications. To more deeply discuss this fact, Fig. 4 shows the breakdown of the total execution time into times for individual modes in *gobmk*. This graph shows the results of all the channels. Figure 4 shows that the ratio of the self-refresh mode (*SREF*) to the total time is only 8%. In contrast, the precharge power-down mode (*PRE_PDN*) is dominant. This fact means that the time duration between one request and the subsequent request is long enough to enter the precharge power-down mode but too short to enter the self-refresh mode. This situation occurs for all the channels. Therefore, these results indicate that it is difficult to fully exploit the self-refresh mode for the applications that cause memory requests intermittently.

1.4. Related Work

This section reviews the related work to reduce the power and energy consumption of the DRAM-based memory systems.

Weis *et al.* [15] have proposed a physical interface that can reduce the energy consumption instead of decreasing the memory bandwidth. This proposal focuses on some applications that cannot exploit the bigger bus width. Therefore, they divide a bus into multiple sub-buses and enables a memory array to activate only a part of a row. This research focuses on the dynamic energy on fine-grain data access pattern, not on the effective usage of the low-power modes.

While this paper focuses on power management using channels, there are some studies focusing on power management using memory ranks. Lebeck *et al.* [9] have discussed how to effectively use DRAM chips including the low-power modes. They have investigated the parameters to enter the low-power modes, and show the first preliminary results of the frequency-based page allocation, which determines the page allocation based on the access frequency. Wu *et al.* have proposed *RAMZzz* [17]. Because applications generally have different access locality, RAMZzz dynamically migrates the memory pages, which are accessed frequently, into some of the ranks. Since the other ranks store only the pages that are less accessed, these ranks can obtain more room to enter the low-power mode. Jang *et al.* [8] have proposed a rank-aware power management method considering the virtual machines to reduce the memory power consumption of data centers. This work considers the scheduling of virtual machines and the placement of their data on the memory system together. The data of virtual machines that are simultaneously executed in multiple cores are gathered into the same memory ranks. Sato *et al.* [12] have proposed a rank-based power management method using multiple address-mapping schemes. Each of the address-mapping schemes determines how data with a physical address is mapped to a physical position of a DRAM-based memory system. By changing the address-mapping schemes, the proposal can gather the data into a single rank or distribute the data across multiple ranks regardless of applications and virtual machines. However, the address-mapping schemes have the limitations, and this method cannot exploit all the ranks to increase the performance.

One of the potential problems of these four studies mentioned above is that the power/energy management is done by controlling ranks. These proposals can work on a memory system with multiple ranks and becomes effective as the number of ranks increases. On the other hand, there are systems that have only one rank in reality. For example, there are typical cases that a personal computer has two DIMM modules, each of which has one rank. However, these modules are generally used for increasing the number of channels for performance improvement, not for increasing the number of ranks per channel. Moreover, the latest high-performance memory module, HBM2, has 8 channels but only one rank [13]. In such cases, the rank-based power management methods cannot be applied.

Hur *et al.* have proposed the *adaptive memory throttling* [6]. This method purposely blocks the incoming memory commands during a certain period. By delaying the commands, the memory system can increase the idle time. The number of cycles for blocking is determined by access history to suppress performance degradation. Bojnordi *et al.* have proposed the programmable memory controller, *PARDIS* [2]. This controller has a dedicated instruction set architecture for accessing DDR memories. They also have implemented application-specific address-mapping schemes on PARDIS and have succeeded in improving the performance and reducing the energy consumption. This fact supports that the memory controller should adopt the deeper optimization regarding the address-mapping schemes. Although these studies do not focus on adjusting the number of requests to ranks and channels, these approaches can be combined with those exploiting the low-power modes by reducing the number of requests.

From the above discussions, this paper focuses on reducing the energy consumption by effective usage of the built-in self-refresh modes and finding an alternative of the rank-based power management methods. There are many cases that the number of ranks is limited in a single memory system. Therefore, this paper focuses on *channels*, which are alternatives to ranks to gather or distribute the data. If the data are gathered into one channel, the number of memory requests to the other channels can be reduced, and the ranks in these channels can

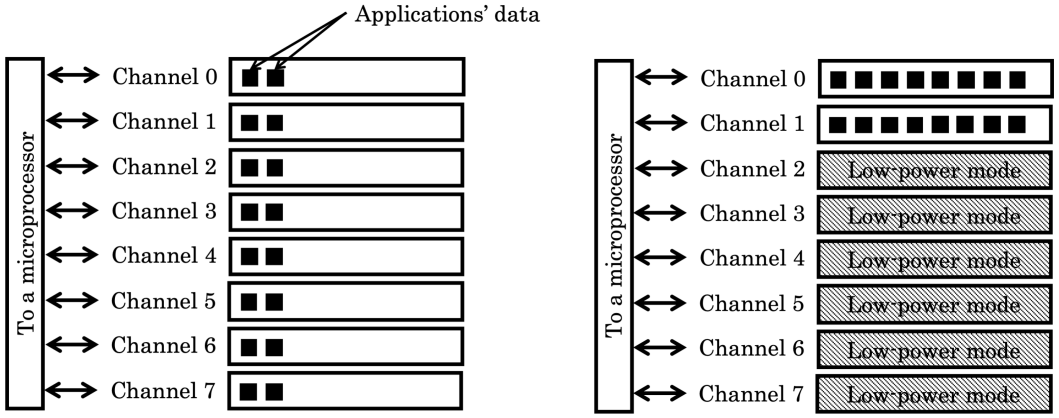
enter the self-refresh mode regardless of the number of ranks. Such an approach will become requisite in the era of many-channel memory systems.

2. Energy-aware Data Placement for Many-channel Memory Systems

2.1. Exploiting the Low-power Modes

Compute-intensive applications do not require the high bandwidth realized by many channels because their access requests are intermittent and not frequent. However, as shown in the results in Section 1.3, some of compute-intensive applications cannot exploit the low-power modes. These applications' data are distributed to all the channels, and these intermittent requests to the data also widely spread over the channels. The frequency of these requests is small but not enough to allow the channels and their components to enter the self-refresh modes. Therefore, the energy consumption of the memory system increases even in the case of compute-intensive applications.

Figure 5 shows the concept of our approach to solve the above problem. Figure 5a is a mode that can fully use all the channels, the *full-channel mode*. In this mode, the applications' data are distributed to all the eight channels. In this mode, the memory system can achieve the highest memory bandwidth. On the other hand, Fig. 5b is a mode that can use a limited number of channels. In this case, all the data are stored into Channels 0 and 1, and only these channels are accessed. To realize such a situation, the data in Channels 2 to 7 are migrated to Channels 0 and 1. Then, Channels 2 to 7 are not accessed. Therefore, these channels can move to the self-refresh mode. The compute-intensive applications can keep the high performance even though the memory system with limited channels reduces the bandwidth.



(a) Full-channel mode (b) 2-channel mode

Figure 5. Full-channel mode for high performance and 2-channel mode for low power

To control different channel modes, this paper considers using address-mapping schemes. As mentioned in Section 1.4, an address-mapping scheme determines where data with a physical address is mapped to a physical location in the DRAM-based memory system. Therefore, each channel mode can be realized by an address-mapping scheme. If these address-mapping schemes can be switched, it can change the channel modes. To switch the address-mapping schemes, the locations of the data of the old scheme can be easily known by those of the new schemes.

This is because, when switching the schemes, the data stored in the memory system have to be migrated to the locations mapped by the newly-switched scheme.

To satisfy the above conditions, this paper considers variations of address-mapping schemes inspired by matrix transposition. By the matrix transpositions, the elements standing in one row are rearranged as those in one column. Such rearranging can be applied to the data migrations to distribute the data over the channels or gather the data into one channel. Figure 6 shows two address-mapping schemes in the case of an 8-channel memory system, and how these schemes map the sequential addresses to channels and groups of rows, called *row groups*. Figure 6a is the scheme for the full-channel mode. Figure 6b is for the 2-channel mode. When the addresses in each scheme are regarded as items in a matrix, those of the 2-channel mode can be generated by transposing those of the full-channel mode by 2x2 sub-matrices. The orange and blue areas in the figures show where the 2x2 sub-matrices are exchanged.

These schemes can satisfy the requirement to realize the channel modes. If the memory requests are based on a sequential access pattern, the requests are distributed, and the accessed channel is changed for each request in Fig. 6a. On the other hand, in the 2-channel mode in Fig. 6b, the sequential access requests firstly go to Channels 0 and 1. After all the data in these channels are accessed, the subsequent requests go to Channels 2 and 3 until all the data in these channels are accessed. Such an access pattern continues to Channels 6 to 7. Focusing on a certain period during execution, two channels are accessed simultaneously. Moreover, to change the channel modes, data are migrated by the pattern of matrix transposition. There is no need to memorize the mapping between physical addresses and locations on the memory system.

	Ch. 0	Ch. 1	Ch. 2	Ch. 3	Ch. 4	Ch. 5	Ch. 6	Ch. 7
Row group 0	0	1	2	3	4	5	6	7
Row group 1	8	9	10	11	12	13	14	15
Row group 2	16	17	18	19	20	21	22	23
Row group 3	24	25	26	27	28	29	30	31
Row group 4	32	33	34	35	36	37	38	39
Row group 5	40	41	42	43	44	45	46	47
Row group 6	48	49	50	51	52	53	54	55
Row group 7	56	57	58	59	60	61	62	63

(a) Full-channel mode

	Ch. 0	Ch. 1	Ch. 2	Ch. 3	Ch. 4	Ch. 5	Ch. 6	Ch. 7
Row group 0	0	1	16	17	32	33	48	49
Row group 1	8	9	24	25	40	41	56	57
Row group 2	2	3	18	19	34	35	50	51
Row group 3	10	11	26	27	42	43	58	59
Row group 4	4	5	20	21	36	37	52	53
Row group 5	12	13	28	29	44	45	60	61
Row group 6	6	7	22	23	38	39	54	55
Row group 7	14	15	30	31	46	47	62	63

(b) 2-channel mode

Figure 6. Maps of the sequential addresses to the channels and the row groups

Figure 7 shows how a physical address is mapped to the physical location of DRAM. These examples are the cases of switching the full-channel mode and the two-channel mode. In the full-channel mode, the bits of row address, bank address, channel address, and column address are extracted as shown in Fig. 7. The channel address is extracted from the least significant bits except for the column address. In the 2-channel mode, the most significant two bits in the row address are exchanged. Hence, the channel address is generated by using the most significant two bits and the least significant one bit except for the column address. Generally, due to the locality of memory accesses, the lower bits in the physical address are changed easier than the upper bits. Therefore, in the 2-channel mode, the lower one bit of the channel address easily changes, but the upper two bits hardly change. It limits the number of accessed channels to two in a certain period. These address-mapping schemes can be generalized. Here, the number of the lower bits in the channel address is defined as n. The matrix transposition shown in Fig. 6 is done by n x n sub-matrices, and the scheme can limit the number of accessed channels to 2^n.

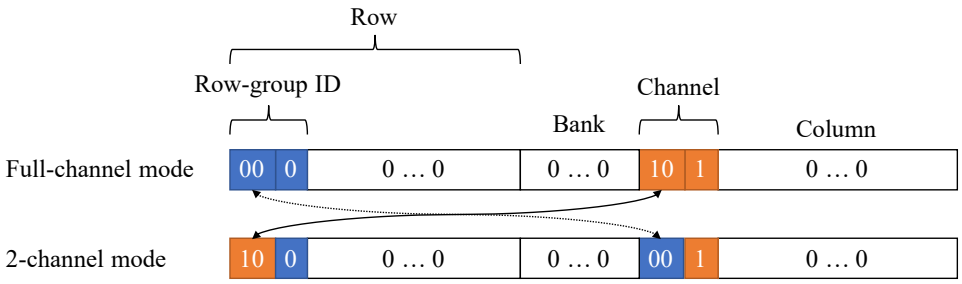


Figure 7. Address-mapping schemes limiting the number of channels

2.2. Dynamic Data Allocation Mechanism

2.2.1. Hardware organization

To realize switching the channel modes, this paper proposes the dynamic data allocation mechanism. The proposed mechanism is implemented on the memory controller, which is the front-end of accessing the memory system. In the case where the number of channels is large, a lot of channel modes are available, e.g., the 1-channel mode, the 2-channel mode, the 4-channel mode, and the full-channel mode in the case where the number of channels is eight. To suppress the hardware cost to manage these modes, the full-channel mode and another channel mode are considered in this paper.

Figure 8 shows the memory controller with the proposed dynamic data allocation mechanism. The memory controller is accompanied with additional components, the *mode register*, the *mode table*, and the *access counter*. The mode register indicates which channel mode should be used currently. The preferred channel mode is determined by the number of accesses at a certain period. The controller counts memory requests by using the access counter. The mode table manages which channel mode is used for the previous access to the physical address by memorizing pairs of the physical address and its channel mode. Note that, the proposed mechanism migrates the data in a unit of row only when the included data are firstly accessed after switching the schemes. This is because it takes a long time to complete migrating all the data to every switching. Therefore, the mode table has to store the pairs for every row.

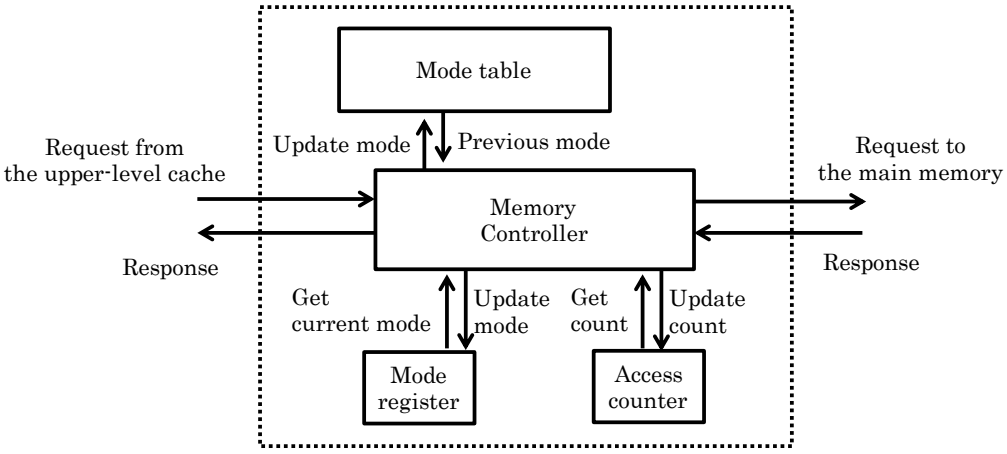


Figure 8. Dynamic data allocation mechanism: a memory controller with facilities switching channel modes

2.2.2. Memory access procedure

The memory controller with the proposed mechanism is placed between the processor and the main memory. The address conversion is done only in the lower layers from the memory controller. Therefore, the processor-side hardware does not need to know whether the data are migrated or not. The memory controller determines the physical location of the address of a memory request by the address-mapping scheme currently selected, and processes the access request.

The detailed procedure to process the memory requests is as follows. If a memory request comes from the processor, the controller knows which is the current mode from the mode register. Next, the controller accesses the mode table based on the physical address to know the mode used in the previous access to that address. If the current mode is different from the previous mode, the data are migrated so that the data can be accessed by the current mode. The mode table is also updated to store the current mode for this physical address. Finally, the memory request is processed by the current mode as the normal memory controller does.

After receiving the data from the main memory, the memory controller knows the channel mode from the mode table when the memory request comes. If the mode is changed from the default mode, here the full-channel mode, the data are sent directly to the processor. If the mode is different from the default mode, the address is reverted to the original one, and the data are sent to the processor.

2.2.3. Hardware overhead of the mechanism

To realize such a migration mechanism, the mode table needs to store the previous address-mapping scheme for each row. In the case where the controller switches two schemes, the mode table should be indexed by a row address, and each entry should store one bit that indicates the mode. In addition, the rows to be swapped should be kept temporarily for the migrations. These costs depend on the configuration of the memory system. The details of the overhead will be discussed in Section 3.

3. Evaluations

3.1. Evaluation Setup

3.1.1. Applications

To evaluate the proposed mechanism, both the applications examined in Section 1.3 and multi-programmed workloads, which are generated by randomly selecting four applications as shown in Tab. 2, are used. To understand the effectiveness of the proposal more clearly, the applications and the multi-programmed workloads combining them are categorized based on the effective bandwidth on average. If the effective bandwidth of an application is lower than 0.5GB/s, the application belongs to Category A. If the bandwidth is higher than 0.5GB/s, the application is put into Category B. Note that the effectiveness of the proposal for the above applications and workloads can also be applied to multi-threaded applications. This is because the proposed mechanism can be affected only by the number of accesses per fixed interval but not by any other factors of applications and workloads. As a result, this paper examines the performance by using single-threaded applications and their multi-programmed workloads only.

Table 2. Multi-programmed workloads

	Applications	Effective Bandwidth (GB/s)
M1	zeusmp, gromacs, sjeng, gobmk	0.163
M2	zeusmp, gamess, sjeng, namd	0.175
M3	tonto, sjeng, gobmk, namd	0.334
M4	dealII, calculix, sphinx3, GemsFDTD	0.526
M5	gamess, gobmk, calculix, astar	0.777
M6	zeusmp, sjeng, calculix, mcf	1.01
M7	gromacs, gobmk, sphinx3, milc	1.54
M8	gamess, namd, GemsFDTD, cactusADM	1.71
M9	dealII, sphinx3, astar, milc	2.11
M10	calculix, GemsFDTD, mcf, cactusADM	2.55
M11	calculix, sphinx3, bwaves, libquantum	2.65
M12	tonto, calculix, milc, lbm	3.78
M13	astar, mcf, milc, cactusADM	3.82
M14	astar, milc, leslie3d, libquantum	4.13
M15	dealII, GemsFDTD, leslie3d, lbm	4.36
M16	mcf, cactusADM, bwaves, lbm	5.60
M17	leslie3d, bwaves, libquantum, lbm	5.91

3.1.2. Parameters of the proposed mechanism

To evaluate the proposed mechanism, the simulation experiment is conducted. This evaluation supposes that the target system has a 4GB HBM2 module with eight channels as the main memory. The simulation parameters are similar to those of the preliminary experiment in Section 1.3, but the system has four cores with L1 private I/D caches, and the L2 cache is shared among the cores. In this experiment, the proposed mechanism switches two schemes, the full-channel mode that uses eight channels and the 2-channel modes. For the proposed scheme, two parameters, *an interval to switch the modes* and *the threshold to trigger the switch,* should be determined. The relationships between these parameters and the effectiveness of the proposal are preliminarily evaluated. Based on the results, the threshold that triggers to switch the mode is set to 0.64GB/s, and the interval to change the scheme is 50ms.

The simulation starts by using the full-channel mode, and the proposed mechanism does not change the initial scheme until the first one billion instructions are skipped. After this skipping, the switching function of the proposed scheme is enabled, and the data correction for the evaluation is validated.

As mentioned in Section 2.2.3, the proposed mechanism needs additional hardware units. The mode table holding which rows are swapped is the most dominant unit from the viewpoint of the energy consumption. In the case of a 4GB memory, the row size of which is 2KB, the mode table has to memorize which channel mode is applied to each row. From this fact, the table should be indexed by 20 bits, and each entry in the table has a one-bit flag that indicates the mode. On the other hand, the table does not need to hold the status of one row for each group of swapped rows and the regions that are not needed to be migrated. By eliminating these status

bits, the number of entries can be reduced by three-eighths. Therefore, the table size required for this experimental condition is 93KB.

The energy overhead of this mode table is estimated by using CACTI [10] and statistics from the simulator. CACTI can estimate the dynamic read and write energy consumptions and the leakage power of memory arrays. Since the number of accesses to the mode table and the execution time can be estimated by the simulator, the total energy consumption of the mode table can be calculated for each workload. Note that the capacity of the memory array must be 2^n bytes in CACTI where n is a natural number. Therefore, the estimated results of the 128KB SRAM array are used in this evaluation. The technology node and the transistor model used in this evaluation is 22nm and the Low Standby Power model (LSTP), respectively. Since the number of accesses to the mode table is not so large compared with those to the upper-level caches, the LSTP transistor can contribute to the reduction in the total energy consumption of the mode table.

3.2. Evaluation Results

3.2.1. Performance

In this paper, the performances of the multi-programmed workloads are evaluated by the *weighted speedup* [5]. This metric is defined as Equation (1).

$$WeightedSpeedup = \sum_{i}^{N} \frac{IPC_i^{shared}}{IPC_i^{alone}} \tag{1}$$

Here, IPC_i^{alone} means IPC of the i-th application in the case where the i-th application is executed solely. IPC_i^{shared} stands for IPC of that application in the case where the application runs with all the N applications in the workload together. The higher weighted speedup means the better performance. Moreover, this metric can fairly assess the workload performance even in the case where one application has a large IPC but the others mark lower IPCs.

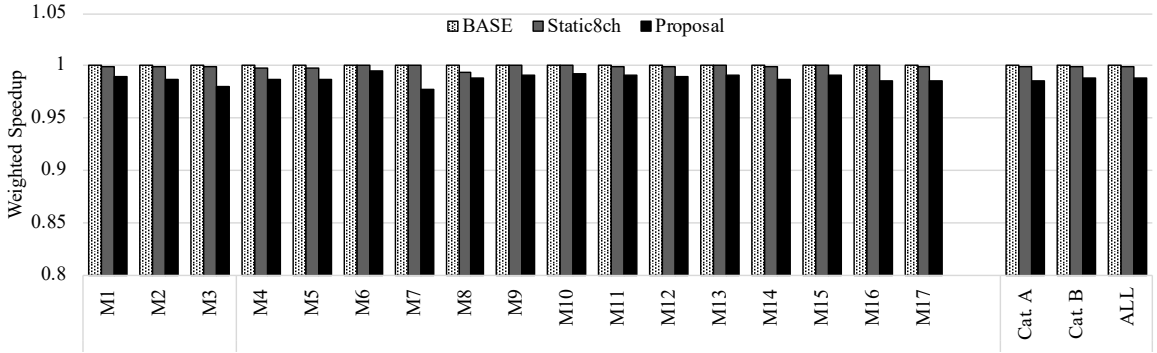


Figure 9. Evaluation results of performance

Figure 9 shows the evaluation results of the performance. The horizontal axis shows the workloads used in this evaluation, and the vertical axis shows the weighted speedup when the self-refresh mode is not used, called *BASE*. The four graphs in one application show the performances in the case of *BASE*, *Static8ch* that always uses the full-channel mode, and *Proposal* showing the results of the proposed mechanism.

Figure 9 shows that, on average, the proposed mechanism slightly decreases the performance by up to 2.3% compared with BASE and Static8ch. Note that these results include the

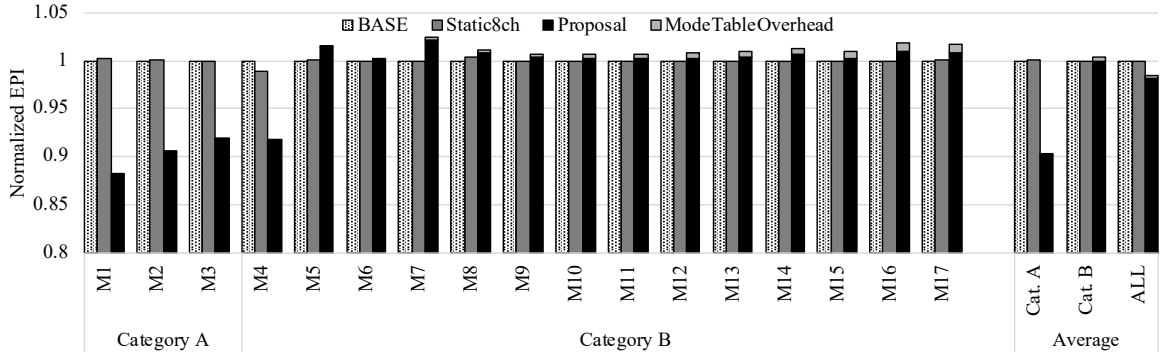


Figure 10. Evaluation results of energy per instruction

performance overheads by the additional latency and the migration costs of the proposed mechanism. Therefore, the performance of the proposed mechanism does not reach those of BASE and Static8ch.

Comparing the workloads in Category A and those in Category B, there are no remarkable differences. The workloads in the Category A easily switch to the two-channel mode because of the low effective memory bandwidth. Although switching the schemes causes the migration overhead, its performance penalty is not so large compared with those of Category B. Therefore, the impact of the migration overhead of the proposed mechanism on the performance is not significant.

3.2.2. Energy consumption

Figure 10 shows the evaluation results of the energy consumptions. The horizontal axis shows the workloads, and the vertical axis shows the energy per instruction (EPI) normalized by the results of *BASE*, which does not use the self-refresh mode. The legend of the graph is almost the same as that of Fig. 9, but an energy overhead of the mode table is added as *ModeTableOverhead*. Note that the energy consumption by the additional memory requests for the migration is already included in that of the proposal.

From Fig. 10, it is observed that the EPIs of the proposed scheme are smaller than those of BASE and Static8ch on average. The proposed mechanism reduces the EPIs by up to 11.8% and 12.0%, and 1.7% and 1.6% on average compared with BASE and Static8ch, respectively. Therefore, the proposed mechanism can contribute to the reduction in the energy consumption by reducing the number of channels.

Focusing on each category, the EPIs of the workloads in Category A are reduced by 9.9% on average. It indicates that the proposed mechanism is effective for these applications, each of which marks the lower bandwidth. On the other hand, for the workloads in Category B, the proposed mechanism cannot reduce the EPIs. This is because the proposed mechanism cannot switch to the 2-channel mode to avoid performance degradation for these workloads.

The proposed mechanism causes the energy overheads due to the migration of the data for switching the modes. Their effects on average do not overwhelm the advantage of the proposed mechanism. However, for some workloads, especially M5 and M7, their EPIs slightly increase. During the execution of these workloads, there are some periods where their effective bandwidth becomes lower than 0.5GB/s. The proposed mechanism switches to the 2-channel modes in that period, and it causes additional memory requests for the migration. Moreover, the overheads of the mode table are relatively large in Category B. This is because the mode table has been

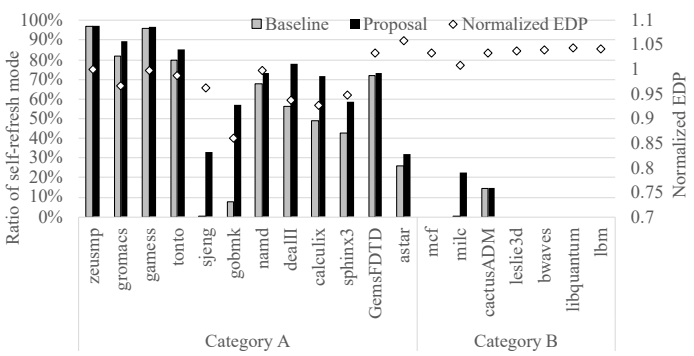


Figure 11. The ratio of entering the self-refresh mode

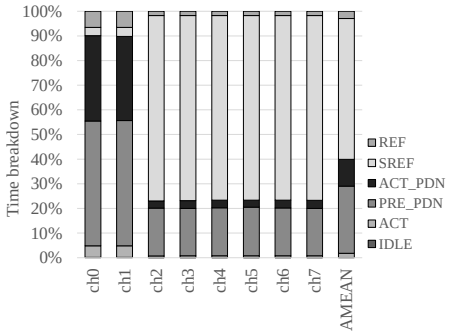


Figure 12. Breakdown of the channels in each mode

checked every memory access. Since the benchmarks in Category B need many memory accesses, the number of accesses to the mode table also becomes large. Therefore, the energy consumption of the mode table increases.

3.2.3. Mode breakdown

To analyze the effectiveness of the proposed mechanism, this section evaluates the breakdown of the execution time in the single-thread application. Figure 11 shows the total length of the period in the self-refresh mode and their effects. The horizontal axis shows the applications. The left vertical axis and two bar graphs show ratios of the self-refresh mode to the total time. The two graphs show the ratios of *Baseline* and *Proposal*, which are the original address-mapping scheme and the proposed mechanism, respectively. The right vertical axis and dot graphs show the energy-delay products when using the proposed mechanism, which is normalized by that of Baseline.

From Fig. 11, it is observed that the ratio of the self-refresh mode increases by the proposed mechanism compared with Baseline. This fact indicates that the proposed mechanism actually increases the time in the self-refresh mode, resulting in the energy reduction. The applications, the times of which in the self-refresh mode increase, especially in Category A, can reduce the energy-delay products. Otherwise, in Category B, the energy-delay products do not decrease. As a result, the energy-delay products slightly increase due to the performance degradations by the performance overheads of the proposed mechanism.

Among these applications, the proposed mechanism is most beneficial to the benchmark gobmk. To analyze the details of this application, Fig. 12 shows the breakdown of the total execution time into times for individual modes in gobmk. The difference from Fig. 4 is that the proposed mechanism is applied. From Fig. 12, it is clear that the time in the self-refresh mode increases compared with the case in Fig. 4 in the six channels that are *ch2* to *ch7* in these figures. This is the main reason for reducing the energy consumption. On the other hand, the fact that the ratios of the active mode (*ACT*) increase in *ch0* and *ch1* indicates that the proposed mechanism successfully gathers the frequently accessed data into *ch0* and *ch1*. Although the time durations of the self-refresh mode of these channels decrease, its effect is not dominant for the total energy consumption.

Conclusions

Modern computing systems employ memory systems with many memory channels to obtain the higher performance. However, the access request comes into the system intermittently, the

ranks in the channels cannot exploit the lower-power modes, especially the self-refresh modes. To solve this problem, this paper proposes a dynamic data allocation mechanism for many-channel memory systems. The proposed mechanism switches multiple address-mapping schemes. One scheme is for limiting the number of accessed channels to reduce the energy consumption. The other scheme is to fully exploit all the channels to obtain the higher performance. The evaluation results show that the proposed mechanism can reduce the energy consumption by up to 11.8% and 1.7% on average.

As future work, the variations of channel modes by different address-mapping schemes should be explored. The proposed mechanism should use more effective channel modes. Furthermore, the proposed mechanism switches only the two modes, however, may become more effective by switching three or more channel modes.

Acknowledgement

This work is partially supported by MEXT Next Generation High-Performance Computing Infrastructures and Applications R&D Program, entitled "R&D of A Quantum-Annealing-Assisted Next Generation HPC Infrastructure and its Applications" and Grants-in-Aid for Early-Career Scientists No. 19K20232.

References

1. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. ACM SIGARCH Computer Architecture News 39, 1–7 (Aug 2011), DOI: 10.1145/2024716.2024718

2. Bojnordi, M.N., Ipek, E.: Pardis: A programmable memory controller for the DDRx interfacing standards. In: 39th Annual International Symposium on Computer Architecture. pp. 13–24. IEEE, Portland, OR, USA (Jun 2012), DOI: 10.1109/ISCA.2012.6237002

3. Borkar, S.: Thousand core chips: A technology perspective. In: The 44th Annual Design Automation Conference. pp. 746–749. DAC '07, ACM, New York, NY, USA (2007), DOI: 10.1145/1278480.1278667

4. Chatterjee, N., O'Connor, M., Lee, D., Johnson, D.R., Keckler, S.W., Rhu, M., Dally, W.J.: Architecting an energy-efficient DRAM system for GPUs. In: IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 73–84. Austin, TX, USA (2017), DOI: 10.1109/HPCA.2017.58

5. Eyerman, S., Eeckhout, L.: System-level performance metrics for multiprogram workloads. IEEE Micro 28(3), 42–53 (2008), DOI: 10.1109/MM.2008.44

6. Hur, I., Lin, C.: A comprehensive approach to DRAM power management. In: IEEE 14th International Symposium on High Performance Computer Architecture. pp. 305–316. Salt Lake City, UT, USA (Feb 2008), DOI: 10.1109/HPCA.2008.4658648

7. Jacob, B., Ng, S.W., Wang, D.T.: Memory Systems: Cache, DRAM, Disk. Morgan Kaufmann (2008)

8. Jang, J.W., Jeon, M., Kim, H.S., Jo, H., Kim, J.S., Maeng, S.: Energy reduction in consolidated servers through memory-aware virtual machine scheduling. Computers, IEEE Transactions on 60(4), 552–564 (Apr 2011), DOI: 10.1109/TC.2010.82

9. Lebeck, A.R., Fan, X., Zeng, H., Ellis, C.: Power aware page allocation. ACM SIGOPS Operating Systems Review 34(5), 105–116 (Dec 2000), DOI: 10.1145/384264.379007

10. Li, S., Chen, K., Ahn, J.H., Brockman, J.B., Jouppi, N.P.: Cacti-p: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 694–701. IEEE (Nov 2011), DOI: 10.1109/ICCAD.2011.6105405

11. Micron Technology Inc.: Micron MT40A2G4 data sheet (Oct 2015)

12. Sato, M., Chengguang, H., Komatsu, K., Egawa, R., Takizawa, H., Kobayashi, H.: An energy-efficient dynamic memory address mapping mechanism. In: IEEE Symposium on Low-Power and High-Speed Chips (COOL Chips XVIII). pp. 1–3. Yokohama, Japan (Apr 2015), DOI: 10.1109/CoolChips.2015.7158660

13. Tran, K., Ahn, J.: HBM: Memory solution for high performance processors. In: MemCon. Santa Clara, CA, USA (Oct 2014)

14. Udipi, A.N., Muralimanohar, N., Chatterjee, N., Balasubramonian, R., Davis, A., Jouppi, N.P.: Rethinking DRAM design and organization for energy-constrained multi-cores. SIGARCH Comput. Archit. News 38(3), 175–186 (Jun 2010), DOI: 10.1145/1816038.1815983

15. Weis, C., Loi, I., Benini, L., Wehn, N.: Exploration and optimization of 3-D integrated DRAM subsystems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 32(4), 597–610 (Apr 2013), DOI: 10.1109/TCAD.2012.2235125

16. Wilkes, M.V.: The memory gap and the future of high performance memories. SIGARCH Comput. Archit. News 29(1), 2–7 (Mar 2001), DOI: 10.1145/373574.373576

17. Wu, D., He, B., Tang, X., Xu, J., Guo, M.: RAMZzz: Rank-aware DRAM power management with dynamic migrations and demotions. In: The International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 32:1–32:11. Los Alamitos, CA, USA (Nov 2012), DOI: 10.5555/2388996.2389040

18. Wulf, W.A., McKee, S.A.: Hitting the memory wall: Implications of the obvious. ACM SIGARCH Computer Architecture News 23, 20–24 (Mar 1995), DOI: 10.1145/216585.216588

19. Yamada, Y.: Vector engine processor of NECs brand-new supercomputer SX-Aurora TSUBASA. In: HotChips: A Symposium on High Performance Chips. Cupertino, CA, USA (Aug 2018), https://www.hotchips.org/hc30/2conf/2.14_NEC_vector_NEC_SXAurora_TSUBASA_HotChips30_finalb.pdf

DOI: 10.14529/jsfi190402

Towards Heterogeneous Multi-scale Computing on Large Scale Parallel Supercomputers

Saad Alowayyed[1,2]***, Maxime Vassaux***[3]***, Ben Czaja***[2]***, Peter V. Coveney***[2,3]***, Alfons G. Hoekstra***[2]

New applications that can exploit emerging exascale computing resources efficiently, while providing meaningful scientific results, are eagerly anticipated. Multi-scale models, especially multi-scale applications, will assuredly run at the exascale. We have established that a class of multi-scale applications implementing the heterogeneous multi-scale model follows, a heterogeneous multi-scale computing (HMC) pattern, which typically features a macroscopic model synchronising numerous independent microscopic model simulations. Consequently, communication between microscopic simulations is limited. Furthermore, a surrogate model can often be introduced between macro-scale and micro-scale models to interpolate required data from previously computed micro-scale simulations, thereby substantially reducing the number of micro-scale simulations. Nonetheless, HMC applications, though versatile, remain constrained by load balancing issues. We discuss two main issues: the *a priori* unknown and variable execution time of microscopic simulations, and the dynamic number of micro-scale simulations required. We tackle execution time variability using a pilot job mechanism to handle internal queuing and multiple sub-model execution on large-scale supercomputers, together with a data-informed execution time prediction model. To dynamically select the number of micro-scale simulations, the HMC pattern automatically detects and identifies three surrogate model phases that help control the available and used core amount. After relevant phase detection and micro-scale simulation scheduling, any idle cores can be used for surrogate model update or for processor release back to the system. We demonstrate HMC performance by testing it on two representative multi-scale applications. We conclude that, considering the subtle interplay between the macroscale model, surrogate models and micro-scale simulations, HMC provides a promising path towards exascale for many multi-scale applications.

Keywords: multi-scale modelling, surrogate model, computational science, heterogeneous multi-scale computing, high performance computing, exascale.

Introduction

Science has the ability to describe phenomena and often to predict their occurrence and outcome in an accurate and rapid manner. These phenomena naturally, and computationally, are multi-scale both in time and space [4, 13, 15, 16, 18, 28, 32]. Multi-scale computing [3, 5, 10, 11, 14, 17–19] depends on scale separation and invokes set of single-scale models, each representing a process in time and space, coupled together to describe a phenomenon ranging over temporal and spatial scales. From a computational point of view, the single-scale models, which by themselves could be massively parallel simulations, could run independently of each other, which provide new opportunities in terms of scalability and performance tuning for the emerging exascale [5, 17]. By investigating a broad range of multi-scale applications from many different domains, we have identified a set of generic patterns that are common to these applications. These patterns can be seen as a "high-level call sequences that exploit the functional decomposition of multi-scale models in terms of single scale models" [5].We believe

[1]King Abdulaziz City for Science and Technology (KACST), Riyadh, Saudi Arabia
[2]Computational Science Lab, Institute for Informatics, Faculty of Science, University of Amsterdam, The Netherlands
[3]Centre for Computational Science, University College London, United Kingdom

there are three generic multi-scale computing patterns to be most relevant for high performance multi-scale computing, namely extreme scaling (ES), replica computing (RC) and heterogeneous multi-scale computing (HMC) [5].

The extreme scaling pattern is where one sub-model (the primary model) dominates the computation, while the others are auxiliary models. The main target is to reduce communication between them and prevent serialisation due to auxiliary models. The replica computing pattern represents a set of multi-scale applications where a large number of single-scale simulations are combined to obtain statically robust outcomes. Distributing these replicas on different supercomputers could lead to an increase in overall performance. For further discussions on these two patterns we refer to [3–5, 25].

The heterogeneous multi-scale computing pattern is based on, and inspired by, the heterogeneous multi-scale method [33]. In a heterogeneous multi-scale method, a complex phenomenon is modelled by employing a numerical solver for the macro-scale equations and obtaining missing properties (e.g., constitutive equations) from suitable micro-scale simulations. Hence, the macro-scale model is coupled with a large and typically dynamic number of micro-scale models [4, 30] (Fig. 1).

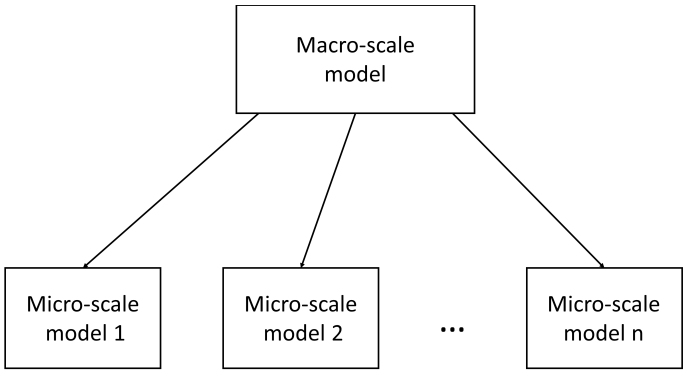


Figure 1. Computational structure of hierarchical multi-scale applications

The HMC pattern implements a family of multi-scale models which, using the multi-scale modelling and simulation framework (MMSF) terminology (see [5, 9, 10]), are single domain with multiple and dynamic instantiations of the micro-scale dynamics [9]. The heterogeneous multi-scale method [1, 33] represents the most obvious multi-scale model that fits the HMC pattern. Other examples could include uncertainty quantification on extreme scaling applications using the so-called semi-intrusive algorithms [26].

The primary potential and advantage of heterogeneous multi-scale modelling is capturing the dynamics at the macro-scale level by explicitly considering some microscopic details of the problem. Thus, HMM is a modelling approach used to numerically solve single-domain multi-scale problems by coupling multiple micro-scale submodels together with a macro-scale model, and each of the micro-scale simulations solves a macroscopic property concurrently. The overall macro-scale behaviour then emerges when the separate submodels are combined. Micro-scale models thus are employed to resolve each unknown component of the problem separately and return the result to the macro-scale model. Frameworks schema taking into account the computational aspects of HMM, certainly in relation to HPC, are rare [5, 20]. For this reason we propose heterogeneous multi-scale computing as a way to efficiently execute HMM models on state of the art HPC infrastructure.

For the purpose of illustration, consider the case of a flowing suspension where the macro-scale constitutive equations may not be known [23]. A lattice Boltzmann model of local fluid velocity u and an advection–diffusion model of local particle density H, representing the macro-scale models, are coupled with a set of fully resolved 3-D lattice Boltzmann suspension model(s). These micro-scale models compute the local viscosity ν and the diffusivity tensor D from a complete run on each, or many lattice point(s) in the macro-scale model to be used in the next time step [23]. This example will be used as proof of concept and is shown in Fig. 2.

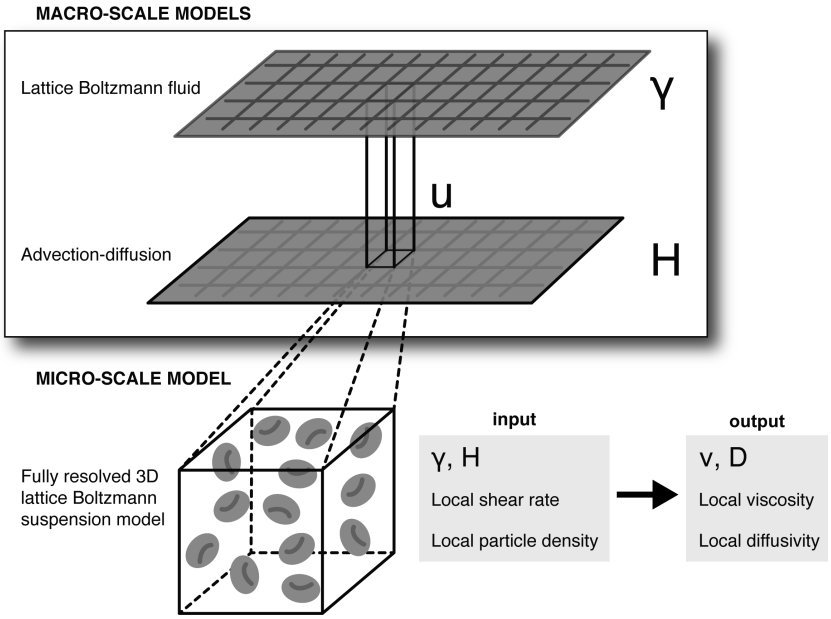


Figure 2. Lattice boltzmann fluid model of local velocity u and advection–diffusion model of local particle density H, representing the macro-scale model, are coupled with a set of fully resolved 3-D lattice boltzmann suspension model(s). These micro-scale models compute the local viscosity ν and the diffusivity tensor D

Load Balancing in HMC

Micro-scale models can be and usually are computationally intensive, such as in the 3-D example considered above. The number of micro-scale models also depends on the spatial properties of the macro-scale model [23]. From a computational point of view, the potentially very large number of micro-scale simulations, the dynamic nature of the amount of required micro-scale models, and their execution time, can become a bottleneck in production runs [5]. This leads to challenging large scale dynamic simulations that are far from trivial to efficiently execute in HPC environments.

To deal with the issue of variable number of executing micro-scale models and their variable execution time in a high performance computing environment, we rely on the pilot job mechanism in combination with a layer of dedicated optimisation and scheduling functionality. The pilot job, for example RADICAL-Pilot [29], is a normal job submitted to supercomputers to reserve resources and use them as an integrated whole. The QCG pilot job manager system [21] has a similar mechanism which allows management of the resources on the application level. A pilot job is akin to a traditional job array, but it allows the scheduling and execution of small and dynamic jobs with different resource requirements.

A possible approach to reduce the number of micro-scale simulations is to utilise a surrogate model as an intermediate layer between the scales to prevent re-computing micro-scale simula-

tions for parameters that are already known, and to use the already computed quantities at the micro-scale, stored in a database, to build a surrogate model. The database and surrogate model, in conjunction with an HMC manager, restores previously computed data, interpolates them where necessary using the surrogate model, and provides input to the macro-scale model(s). Depending on the state of the surrogate model and the history of the data required, this mechanism can reduce the number of micro-scale simulations significantly, potentially by orders of magnitude.

Generally, this approach is practical and has already been demonstrated in multiple fields [22, 31]. The exchange of data should not be a bottleneck in this approach, because the exchange between scales usually involves a few floating-point numbers that represent specific properties. However, the number of micro-scale models should be mapped efficiently to available hardware resources to avoid potential load-balancing required at runtime issues [5].

The role of the HMC manager is to build the surrogate model and evaluate it when needed. In this process, at every time-step the macro-scale model sends a request to the HMC manager for required quantities. The manager then consults the surrogate model for the required information. For this purpose, a user-defined script is implemented to decide whether the cached or interpolated data is sufficient. If not, the manager will start new micro-scale models to obtain more accurate results. To prevent the manager itself from becoming the bottleneck, so as we will assume asynchronous I/O and separate computing resources for the HMC manager. A schematic of this process is shown in Fig. 3.

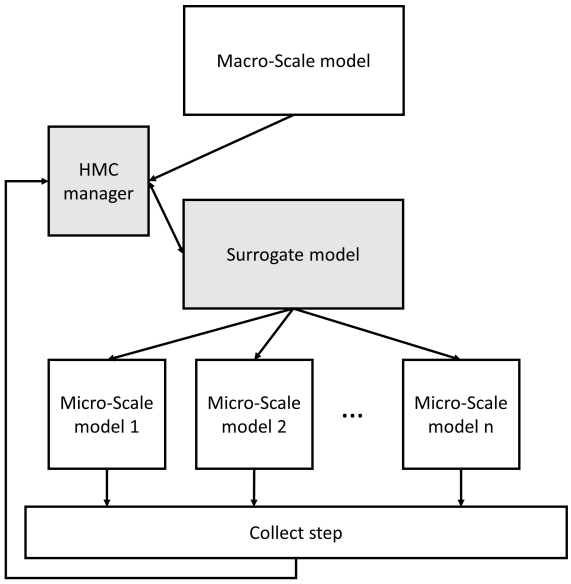


Figure 3. Main workflow of the heterogeneous multi-scale computing and different components (macro, micro, surrogate and HMC manager). The main components are highlighted

Although the main benefit of the dynamic nature of the surrogate model is to decrease the computational cost, it still may cause load imbalance because the number of single-scale models, at each macro-scale iteration, is dynamic [22]. The purpose of the HMC pattern is to mitigate these issues and provide a level of control over the execution that aims to minimise load imbalance and maximise resource utilisation over the complete HMC run.

Our approach is to use the resources dynamically supported by the architecture, with the assistance of tailored scheduling which is controlled by the pattern. In the next section, we will focus on load balancing in micro-scale models in HMC.

1. Methods

A typical iteration of an HMC application consists of simulating one time-step of a macro-model, followed by a large number of simulations of micro-models. The outputs of the micro-model simulations are synchronised before moving to the next iteration of the macro-model. While the single macro-model simulation usually is straightforward to perform, efficient use of resources during simulation of the micro-model requires potentially complex scheduling. A first issue arises from the submission of the micro-model simulations, as their large number makes it impractical to submit them individually to the supercomputer job manager. The relatively small size of each of these jobs would generally grant them a low execution priority in turn, the variable queuing time may cause significant bottlenecks when synchronising the micro-model simulations results.

A simple solution is to batch the micro-model simulations and perform them in a single, large resource allocation. This large allocation can either be requested at the beginning of the multi-scale simulation, or at every iteration of the workflow. The former avoids repeated queuing periods of time, while the later releases the large allocation during the execution of the macro-model simulation. Independently of the chosen method, the micro-model simulations have to be internally scheduled within the requested resource allocation. The Internal scheduling algorithm, set by the user in QCG, is First Come First Serve, while the large allocation can be subdivided into a separated number of sub-allocations of fixed size. An identical number of cores is allocated to each sub-allocation. Naively, before starting the first micro-model simulation, each of these are arbitrarily assigned to a sub-allocation, such that each will perform a similar number of simulations (see Fig. 4a and b).

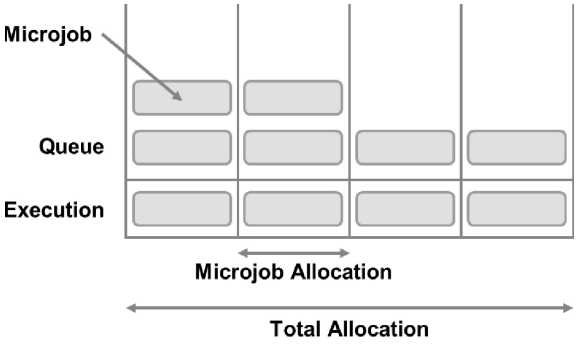
However, this simplistic sub-allocation assignment method can rapidly become inefficient. Even though each sub-allocation is similar in terms of resource size and number of simulations to perform, nothing guarantees that each of the simulations have comparable execution time. Due to the lack of *a priori* knowledge about this execution time, arbitrary assignment can cause some sub-allocations to complete their simulations far quicker than others by, for example performing only short ones. Consequently, a significant load imbalance can result in leaving sub-allocations idle for long periods of time (see Fig. 4c).

1.1. Optimisation of Resources

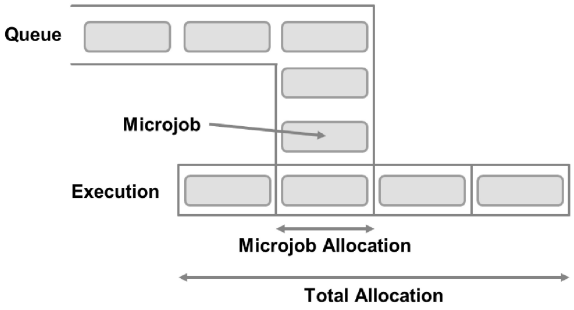
We propose to address the load balancing issue induced by arbitrary *a priori* resource assignment and variable execution time of the micro-model simulations using two mechanisms: (i) pilot job manager (PJM) internal scheduler, and (ii) optimisation of sub-allocation size.

The PJM mechanism essentially consists of an internal job scheduler for the large allocation provisioned for the whole set of micro-model simulations of a given iteration. The execution order of the jobs is specified in a First In First Out (FIFO) manner. The main advantage of the PJM is that all the jobs are gathered in a single queue, allowing execution on any subset of cores from the large allocation, as soon as the required resource size is available. Moreover, the size of this sub-allocation can be easily specified independently for each job, which helps when size adjustment is needed to reduce execution time disparities.

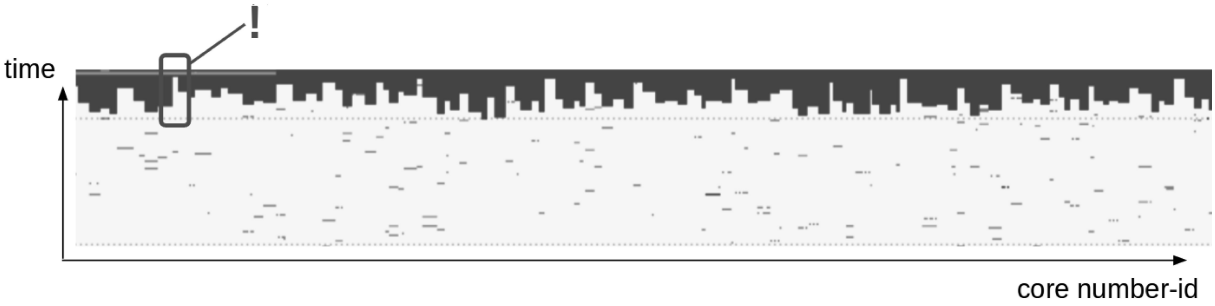
In order to reduce the variability of the micro-model simulations execution time, the sub-allocation size can be adjusted individually. Acknowledging that micro-model simulations ought to scale strongly up to certain number of cores, this can be achieved without any loss of effi-



(a) Naïve, equitable partition of the total allocation with arbitrary and *a priori* assignment of the simulations



(b) Pilot job manager based scheduling, with single queuing mechanism allowing on-the-fly assignment of free resources



(c) Consequences of the variable execution time of the micro-model simulations on idle time of resources. In light the portion of actual computation, and in dark, idle time. Few sub-allocations (e.g., Exclamation mark) can keep all other sub-allocation idle for a significant amount of time causing load imbalance

Figure 4. Internal scheduling of micro-model simulations inside a large allocation

ciency, as long as the sub-allocation size remains in the strong scaling domain. The upper limit of the strong scaling domain can easily be obtained from benchmarks scaling (see Fig. 8b). Subsequently, the simulations allocation size can be re-scaled proportionally to their estimated execution time. The longest simulations are performed with the number of cores associated with upper boundary of the strong scaling domain, while the shortest are performed with the lower boundary, that is one core. However, in less variable micro-model simulations execution time, in job farming it is always better to run with the lower bound as shown next.

Predicting execution time can be done in various ways which either rely on benchmarking data or a reference model function of input parameters. The former can be queried directly to obtain an estimation of an execution time (see Fig. 8a).

1.2. Incorporating Surrogate Model

Now we want to consolidate the multi-scale model with a surrogate model, let us assume that we will utilise a pilot job running on P_{total} processors for a time T, where P_{total} and T are decided by the user at launch time. Let the macro-scale model (M) execute on P_M processors, the surrogate model (S) execute on P_S processors, the HMC manager (HMC) on P_H processor and η micro-scale models execute on P_μ processors. For the macro-scale model, the HMC manager and the surrogate model, the processor allocation is static while for the micro-scale models it is dynamic.

Ultimately, the number of micro-scale models η to be executed is dynamic and changes at every macro-scale time step, depending on how effective the surrogate model is in predicting

the missing quantities. Thus, $\eta(t)$ is the number of micro-scale models at time t. Its range is $(0 \leq \eta \leq DoF)$, where DoF is the total number of degrees of freedom of the macro-scale model that are coupled to the micro-scale simulations. This could, for example, be the number of lattice points in a flow simulation where, in each lattice point, the viscosity is unknown and needs to be obtained from a micro-scale simulation.

The number of successful calls to the surrogate model per time step, which replaces the need to generate micro-scale jobs, is $(\eta_D(t))$. This will reduce the number of micro-scale models to

$$\eta(t) = DoF - \eta_D(t). \tag{1}$$

We define $g(t)$ as the performance of the surrogate model,

$$g(t) = \frac{\eta_D(t)}{DoF}, \tag{2}$$

and then write the actual number of micro-scale models that need to be executed at a macro-scale time step as

$$\eta(t) = DoF(1 - g(t)). \tag{3}$$

To complete the process, the number of processors allocated to run *one* micro-scale model i, $P_{\mu i}$, is determined at *runtime* by the HMC manager. These numbers are stored in an array $(P_\mu[])$ to represent the number of processors to run the $\eta(t)$ micro-scale model(s) for one macro-scale iteration t. Also, the computing resources for the HMC manager P_H are determined separately.

At launch time, the HMC pattern software obtains the number of processors for each main component (i.e. P_M, P_S, P_μ and P_H) and the macro-model degrees of freedom (DoF) from the user. Also, the initial performance measurements of the micro-scale model are benchmarked to retrieve the minimum $P_{\min}$ and maximum $P_{\max}$ number of processors to run *one* micro-scale model. $P_{\min}$ is determined by memory requirements and $P_{\max}$ by strong scaling behaviour where $P_{\max}$ denotes the number of processors required, where the execution time is minimised.

Depending on the budget of the user, the requested time to completion and the expected number of micro-scale models, the user would estimate the resources to be used and the time required for execution. Next, the variables used during runtime, namely the number of processors per micro-scale model $P_{\mu i}$, must be determined as discussed previously.

Generally, the total number of processors for micro-scale models P_μ can be used either in *stateful* or *stateless* mode. If the micro-scale models are stateful, then the micro-scale models reside in memory and the HMC manager feeds the micro-scale models with appropriate input and initial conditions. Although this method is easy to implement, it assumes that all micro-scale models will be the same, requiring the same time and computational power, which is a main source of a load-imbalance situation.

In the *stateless* mode, the number of processors per micro-scale model $P_{\mu i}$ is totally dynamic. In this case, the pattern software decides how to utilise P_μ for each macro-scale iteration. The main advantage of this method is that when the computation of the micro-scale models is complete, their nodes/cores can be used for pre-calculating some of the database properties of the surrogate model, or simply release them to the system.

The distribution of the number of nodes/cores per micro-scale model depends on the number of micro-scale models requested and the number of available processors for these micro-scale models. In the following subsections, we will discuss a mathematical model for the dynamic num-

ber of micro-scale models, and we will provide an example of the performance of the surrogate model.

1.2.1. Execution time model

In this section, we discuss optimal manners in which to run the micro-scale jobs. First, the computing time for one and for multiple micro-scale models is analysed, then the performance of the surrogate model $g(t)$ per time step during the simulation is evaluated.

The computing time to run *one* micro-scale model using $P_{\mu i}$ processors can be calculated utilising the concept of fractional overhead [5, 7], with

$$T^{\mu i}(P_{\mu i}) = \frac{T^{\mu i}(1)}{P_{\mu i}} + To(P_{\mu i}) = \frac{T^{\mu i}(1)}{P_{\mu i}}(1 + fo(P_{\mu i})), \tag{4}$$

where $T^{\mu i}(1)$ is the time to run *one* micro-scale model using a single processor and $To(P_{\mu i})$ is the overhead time for running *one* micro-scale model using $P_{\mu i}$ processors. The fractional overhead fo for running *one* micro-scale model using $P_{\mu i}$ processors can be expressed as

$$fo(P_{\mu i}) = \begin{cases} 0, & \text{if } P_{\mu i} = 1 \\ P_{\mu i} To(P_{\mu i})/T^{\mu i}(1) > 0, & \text{otherwise.} \end{cases} . \tag{5}$$

The total time T to run *each* micro-scale model on the same number of $P_{\mu i}$ processors (so, independent of i) using a total of P_μ processors for all micro-scale models will be

$$T = \frac{\eta(t)}{\lceil P_\mu / P_{\mu i} \rceil} \left(\frac{T^{\mu i}(1)}{P_{\mu i}} \big(1 + fo(P_{\mu i})\big) \right) \sim \frac{\eta(t)}{P_\mu} \big(T^{\mu i}(1)(1 + fo(P_{\mu i})) \big), \tag{6}$$

where $\eta(t)$ is the number of micro-scale models in time step t. The target is to minimise T, then

$$P_{\mu i} = 1; fo(1) = 0, \tag{7}$$

$$T = \frac{\eta(t) T^{\mu i}(1)}{P_\mu}. \tag{8}$$

Otherwise,

$$T = \frac{\eta(t) T^{\mu i}(1)}{P_\mu}(1 + fo(P_{\mu i})), \tag{9}$$

which means that the fewer processors we use per micro-scale model, the lower will be the overhead. One should note the difference with Section 1.1, where the surrogate model is not used. In the second step we assume that $T^m u_i(1)$ is equal for all i, so very little or no variability in the runtime for the microscale simulations comparing to the surrogate model.

1.2.2. Surrogate model performance

The number of micro-scale models $\eta(t)$ changes with every macro-scale model iteration and is highly dependent on the state of the surrogate model. For this, we need to analyse the performance of the surrogate model per time step ($g(t)$). The value of g can vary for cases where the user is building the surrogate from scratch ($g \longrightarrow 0$), to cases where the surrogate model replaces the micro-scale models efficiently ($g \longrightarrow 1$), or for in-between situations. To deal with

the distribution of the number of nodes/cores in these cases, we defined three different phases, and the corresponding processor distribution mechanism is shown in Alg. 1.

The important elements in Alg. 1 are the number of processors reserved for all micro-scale models P_μ, and $\eta(t)$, the number of micro-scale models in time step t. If $P_\mu < P_{min}\eta(t)$, then the most appropriate action to take is to perform farming by running each micro-scale model with a minimal number of processors P_{min}. On the other hand, if $P_\mu < P_{max}\eta(t)$, then running with P_{max} is the most suitable choice. Otherwise, for $P_{min}\eta(t) < P_\mu < P_{max}\eta(t)$, all micro-scale models must be run on $\lceil P_\mu/\eta(t) \rceil$, limiting it to no more than P_{max}, which is the maximum number of processes that the model can benefit from before performance decreases. Also, if the performance values of running the micro-scale models using different parameters are available or can be estimated, the number of processors per micro-scale model can be changed accordingly in the second phase. We will apply this concept to two different cases for the surrogate model, starting from scratch and from a developed surrogate model.

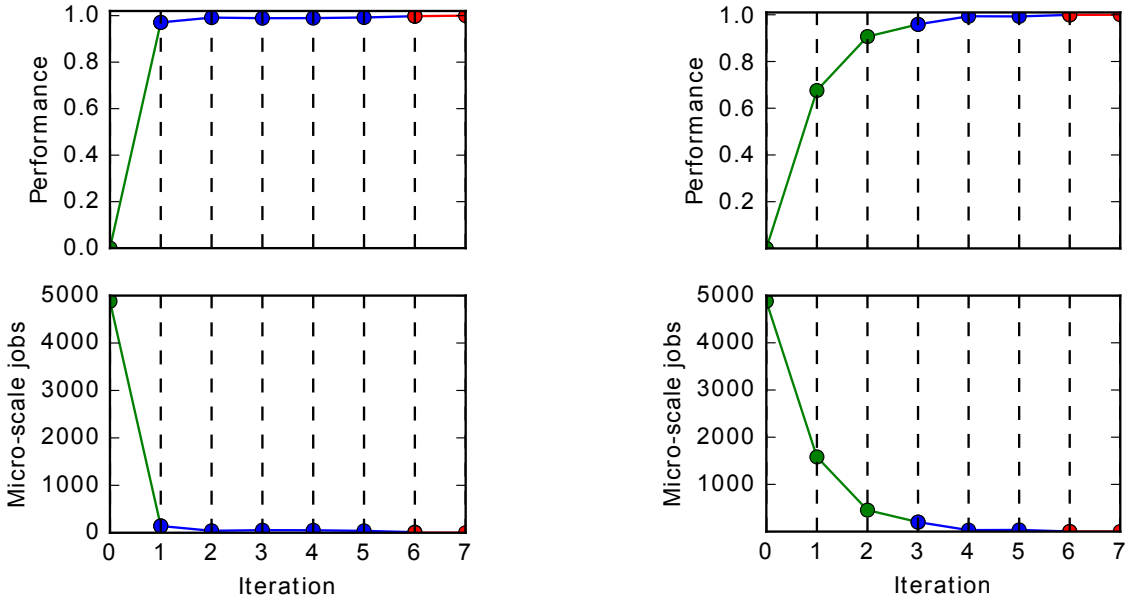
Algorithm 1 HMC phases

```
1: procedure HMC(g(t), DoF, P_min, P_max, P_μ)
2:     η(t) = DoF(1 − g(t))
3:     if η(t)P_min > P_μ then                          ▷ Phase 1
4:         run(η(t), P_min)
5:     else if η(t)P_min < P_μ < η(t)P_max then         ▷ Phase 2
6:         run(η(t), ⌈P_μ/η(t)⌉)
7:     else                                             ▷ Phase 3
8:         run(η(t), P_max)
```

Case (a): Surrogate model from scratch

If the user starts building the surrogate from scratch, i.e. for $g \longrightarrow 0$, then it is meaningless to run DoF micro-scale models to fill the database at the beginning. This will be inefficient, since the degree of freedom can easily reach 10^6 or more. What can be done is cluster input parameters into an input subsample. In this case, the initial batch of micro-scale models (η_{init}), from which a surrogate model is built, is first executed, then run the next batch (either look it up in the database or run the micro-scale model), and so on.

Figure 5 shows the performance of a surrogate model (top graphs), expressed by $g(t)$, and the corresponding number of micro-scale jobs (lower graphs) for two different performance levels of the surrogate model. Both examples had a ($DoF = 48818$). The colours show the three phases, as introduced above. Phase one, where the farming of jobs is done using P_{min} processors per micro-scale model, is represented in green. Phase two is shown in blue, and the third and final phase is shown in red. The dashed lines in the figures are the macro-scale model iterations. The first example, Fig. 5a, shows a surrogate model with good performance (model = scratch, good performance), while the second example; Fig. 5b, demonstrates poor performance (model = scratch, poor performance). These performance figures are based on the results from a simulation in which this surrogate model was actually implemented [22] with modification at the first few macro-scale iterations in order to mimic the case of a new surrogate model.



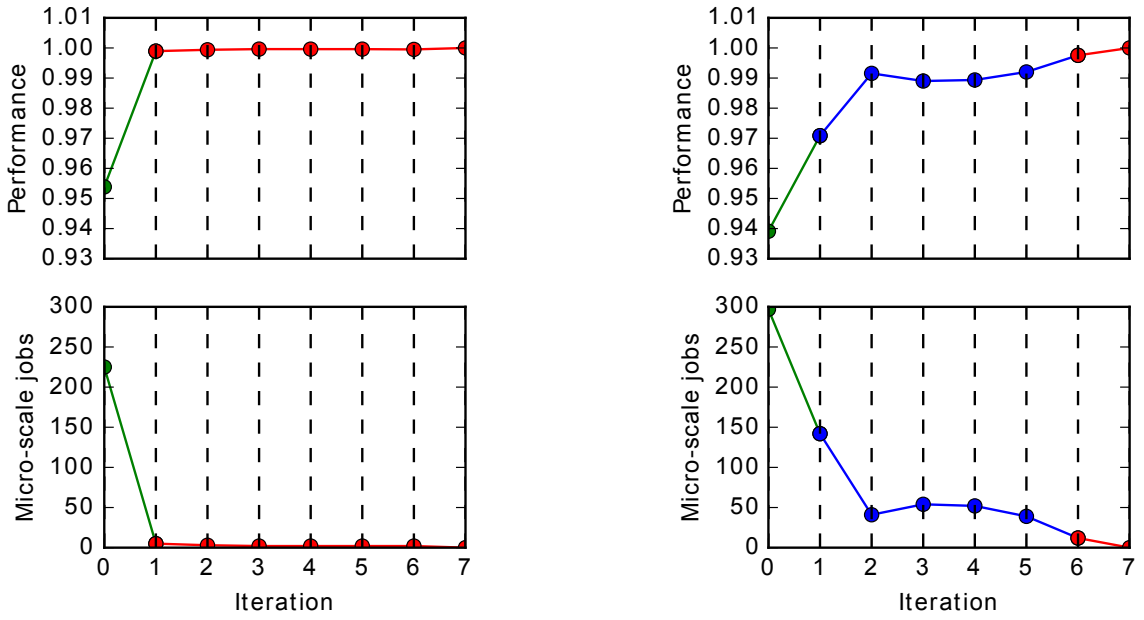
(a) (b)

Figure 5. Behaviour of a surrogate model generated from scratch for two different performance levels. A good performance level is on the left (model = scratch, good performance), a poor performance level is on the right (model = scratch, poor performance). The upper two panels show the performance values of the surrogate model and lower two panels show the corresponding numbers of micro-scale jobs. In this example, $P_{min} = 1$, $P_{max} = 16$ and $P_{\mu} = 206$. The colours refer to the phases, where green is the first phase, blue the second, and red the third. The dashed lines represent the macro-scale model iterations

Case (b): Developed surrogate model

A HMM simulation can also be executed using a previously constructed surrogate model. Figure 6 (top) shows the performance of a well-established surrogate model and the corresponding micro-scale models (Fig. 6 (lower)) for two different performance levels of the surrogate model. As for case (a), both simulations had a $DoF = 48818$. The performance figures are based on the results from a simulation in which this surrogate model was actually implemented [22].

As shown in Fig. 5 and 6, the first phase of the new surrogate model, case (a), requires more jobs at the beginning to build the surrogate model. This phase also takes more macro-scale iterations to complete. Note that the number of micro-scale jobs per iteration is calculated as $\eta(t) = DoF(1 - g(t))$. However, in the first phase we do not run the total number of degree of freedom (48818) jobs, but we run batches from which we can then train the surrogate model. In the second phase, the number of micro-scale jobs is less than the number of micro-scale jobs in the first phase. Knowing that it is not beneficial to run a micro-scale job utilising more than P_{max} processors, and the time to run a micro-scale job varies with the number of processors and the input parameters, we might have a number of idle cores/nodes. For the two study cases, we can use the free cores/nodes in the second and third phases to further explore the parameter space of the micro-scale model for better performance of the surrogate model, or even change the number of processors per micro-scale model based on different input parameters for each micro-scale model. In the third phase, it is preferable to release a number of unused processors back to the system to save on the cycle budget. Generally, switching between phases will be totally dynamic and the runtime part of the HMC pattern software should handle this process, as will be illustrated in the next sections.



(a) (b)

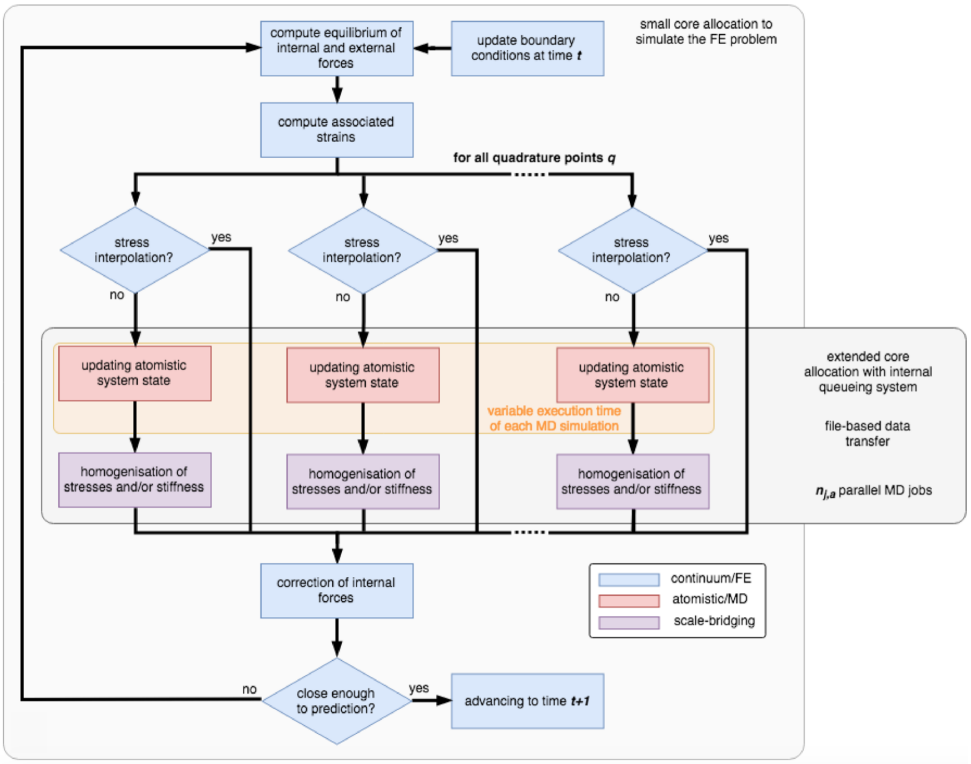
Figure 6. Behaviour of a well-developed surrogate model for two different performance levels. The good performance level is shown on the left (model = developed, good performance), the poor performance level on the right (model = developed, poor performance). Upper panels show the performance values, and lower panels show the corresponding numbers of micro-scale jobs. In this example, $P_{min} = 1$, $P_{max} = 16$ and $P_{\mu} = 206$. The colours refer to the phases, where green is the first phase, blue the second, and red the third. The dashed lines represent the macro-scale model iterations

2. Results

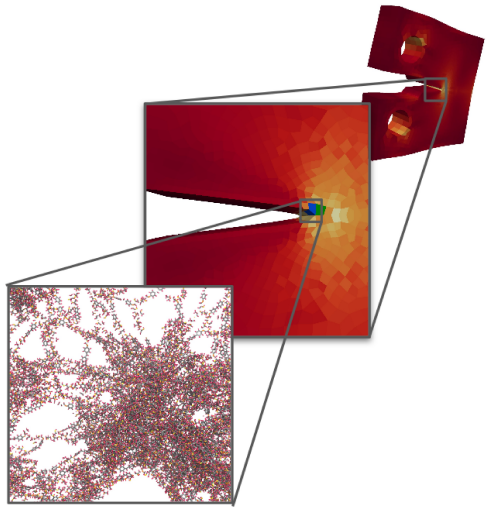
In this section, we will discuss our results for addressing the issues of load balancing both due to variable execution time and to a dynamic number of micro-model using two representative applications, which are a nano-materials system and a red blood cell suspension system, respectively.

2.1. Nano-materials Application

Ab initio physical models are unfeasible beyond the smallest scales (i.e. the nano-scale), although density functional theory and molecular dynamics are the models of choice of material scientists, these models overlook the geometrical and structural complexity of the engineering scale, which induces a heterogeneous conglomeration of local mechanical states. Correctly capturing these is essential to observe the emergence of macroscopic materials properties. Current attempts to explore the properties of a system of atoms when bridging from the atomistic to the continuum scale most often involve ad hoc assumptions [12], in the form of constitutive modeling, most certainly missing out substantial peculiarities of the newly formulated material. Our HMC application [30] computes the dynamic equilibrium of mechanical forces in a continuum structure using the finite element method (FEM). In a classical FEM approach the local constitutive relation between stresses and strain comprises a series of phenomenological mathematical equations, but in the present case it is replaced by a molecular dynamics (MD) simulation (see Fig. 7b). An MD simulation, with nanoscale detail of the material structure, is



(a) Computational workflow coupling the FEM and MD models



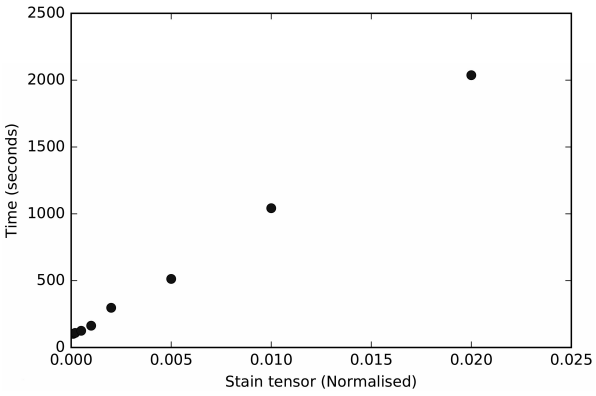
(b) Zoom across the multiple scales of the workflow, from which the continuum description reproduces realistic boundary conditions of a standard compact-tension test for fracture toughness estimation, embedding at atomistic description of the polymer, capturing the chemical specificity of the crosslinked chain network

Figure 7. Prediction of mechanical properties using an HMM workflow

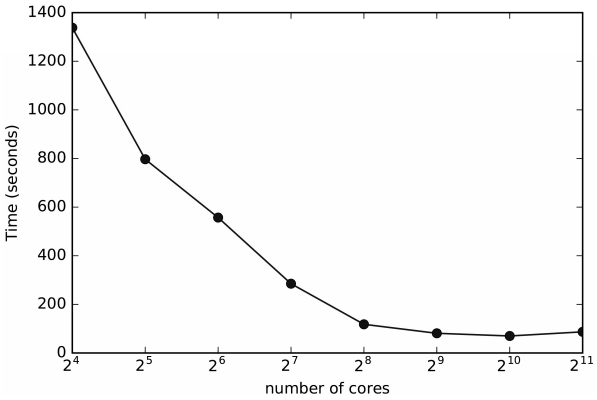
performed whenever the stress resulting from an applied strain history is required. In a nutshell, the application consists of a macroscopic FEM model which synchronises the simulation of a large number of microscopic MD models iteratively as time advances (see Fig. 7a).

In more detail, the macroscopic model feeds a microscopic model with a given strain tensor; the microscopic model evolves following Newton's equation of motion to reach the given strain.

The resulting evolution is dependent on the strain rate applied on the microscopic system. In turn, the execution time of the microscopic simulation is highly correlated with the amplitude of the applied strain (see Fig. 8a). Any heterogeneous macroscopic field at a certain iteration in the macroscopic model will inevitably result in varying execution time of the microscopic model simulations. However, as mentioned in Section 1.1, this variability can be toned down by exploiting the known strong scaling curves of the microscopic model (see Fig. 8b). Note that since the microscopic system evolves out of equilibrium with the applied strain, each finite element cell is necessarily more associated with its own atomistic structure. This makes the use of surrogate modelling difficult with the nano-materials application, but this will be addressed with the red blood cell suspension application in Section 2.2.



(a) Linear dependence of the strain amplitude on the execution time



(b) Strong scaling of the simulation of the model of the 40000 atoms structure strained at 0.1% of epoxy resin

Figure 8. Benchmarking of the molecular dynamics model simulation

The microscopic MD systems describe epoxy resin and graphene nanocomposites, involving approximately 40000 atoms. Each microscale system represents a 10 nm wide cubic box. The microscopic simulations are performed with a constant time step of 2 fs and a constant strain rate of 10^{-4} s^{-1}. The MD simulations are performed using LAMMPS [27] and the ReaxFF force field [2] that captures on-the-fly bonding and debonding of atoms. The macroscopic system models the structure of a compact-tension test following ASTM International Standards [6] (ASTM/E1820). The geometry of the test is specifically designed to trigger the appearance of a single crack in the structure. The dimensions of the compact-tension test specimen are typically of the order of a few centimetres, while the load is applied over a few seconds (see Fig. 7b). Assuming time scale separation, the macroscopic simulations are performed with a much larger time step, that is 0.5 μs, as compared to the microscale simulations. Similarly, assuming space scale separation, the FEM cells are 1 mm wide. The FEM simulations are performed using the Deal.II library [8].

2.1.1. Results

We perform a benchmark of the nano-materials application on the five first time steps of the simulation described in Section 2.1. Internal scheduling of the micro-scale model simulations is executed with three different configurations: (i) a *naïve*, (ii) a *PJM*, and (iii) a *PJM+opt* configuration. The first (*naïve*) corresponds to the configuration described in Fig. 4a, with

arbitrary and *a priori* assignment of the simulations. The second (*PJM*) refers to the pilot job manager based scheduling (see Fig. 4b), with single queuing mechanism allowing on-the-fly assignment to free resources. The third (*PJM+opt*) is equivalent to the *PJM* configuration, with the additional optimisation layer compensating for the variability of the micro-scale model simulation execution time. This benchmark is performed for two sizes of allocation: (a) 20 nodes and (b) 100 nodes each computing a total of 28 cores on the Eagle supercomputer[4] in Poznan, Poland. In this test case, the strong scaling limit of the micro-scale model is 4 nodes due to the limited number of particles. In the *naïve* and *PJM*, we assume that the micro-scale simulations are all performed on the same sub-allocation size, hence 4 nodes. From practice and the in-depth knowledge discussed in 1.2.1, in case of large number of replicas, running on one node will reduce the communication and would provide a better performance. In the *PJM+opt* configuration, the sub-allocation is varied from 1 to 4 nodes following the predicted execution time of the micro-scale simulation. The reason for this choice is because one replica takes the whole node, i.e. all the 28 cores.

The total runtime for each of the three internal scheduling configurations with the two sizes of allocation is shown in Fig. 9a, b. Independently of the resource allocation size, we observe an important speedup, up to a 70% runtime reduction, in configurations featuring the flexible PJM mechanism. Nonetheless, the relative speedup in presence of the PJM is reduced on larger allocations to approximately 35%. As the allocation size grows, with a constant amount of micro-scale simulations, internal queuing is reduced as is the predominance of load balance induced by rigid *a priori* plan. Conversely, flexible internal queuing dominates when the total allocation size becomes comparable to the micro-scale simulation allocation, respectively 20 and 4.

A more detailed analysis using the performance reports provided by the PJM, helps us to determine why the optimisation layer actually has an effect on the total runtime with a 100 nodes application. In Fig. 9c, d, the PJM analytics indicate the evolution over the course of the simulation of the utilisation percentage of the resources, the number of concurrent micro-scale simulations, and the instantaneous average of cores per micro-scale simulation. In the first iteration, we can then observe that for the simulation with the *PJM+opt* configuration with 100 nodes (see Fig. 9d), 62 out of 100 nodes are not used due to misleading scheduling decisions and currently working on improving this. In turn, the first iteration is slightly longer with than without the optimisation layer. Nonetheless, when we focus on the last three iterations the *PJM+opt* configuration is actually faster than the *PJM* configuration by 20% due to the size of single-scale model. These three iterations demonstrate the benefit of reducing the execution time variability between micro-scale simulations. Clearly, this gain of performance would be way beneficial when we run the simulation for a large number of iterations. The small peaks in Fig. 9d (middle) represent the small and fast micro-scale models being executed by the end of each macro-scale iteration.

The detailed analytics reported by the PJM also illustrate generally the benefits of having a flexible internal queuing system. Consistently for both configurations shown in Fig. 9c, d, we observe optimal usage of the total allocation up to a certain, late, point in time. This corresponds to the point in time when the queue of simulations is finally emptied. From then onward, resource usage slowly decreases until all simulations are completed.

[4]https://wiki.man.poznan.pl/hpc/index.php/Eagle

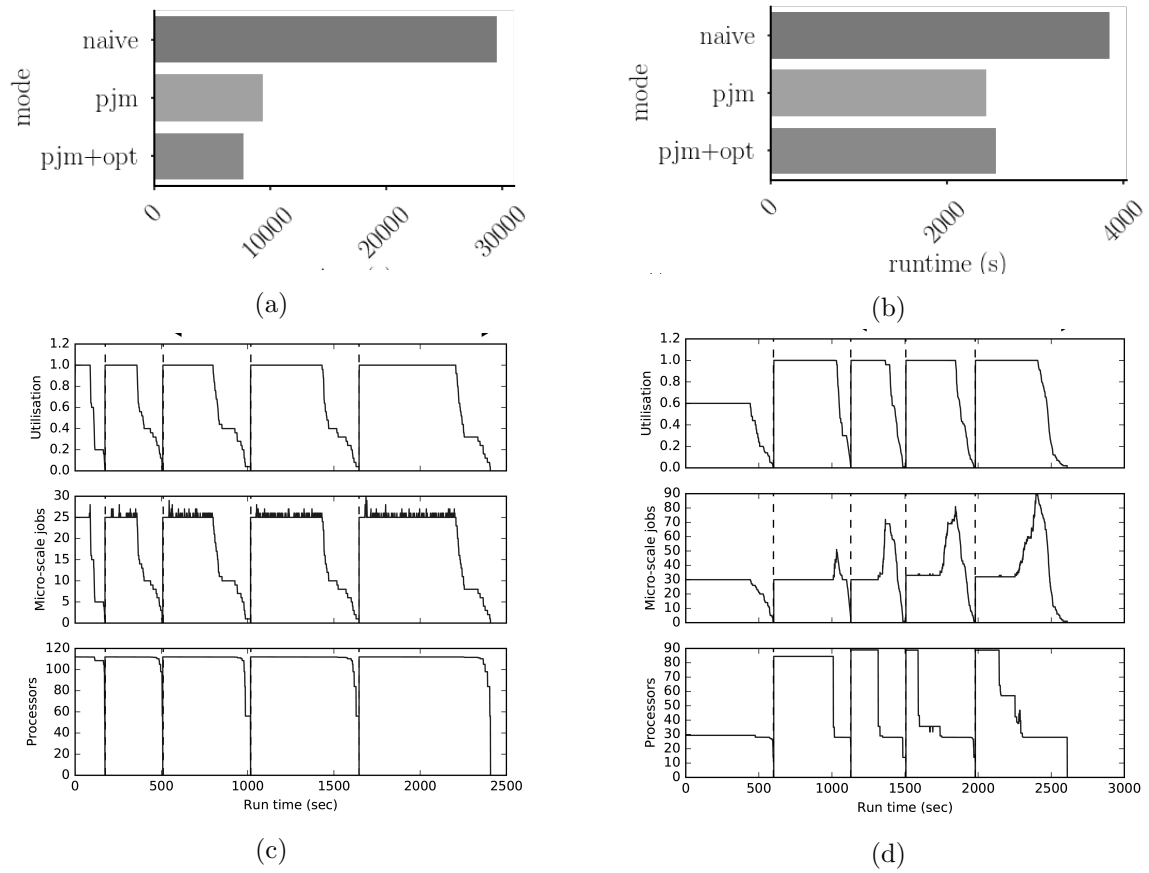


Figure 9. Benchmarking of the full nano-materials with various internal scheduling methods. Comparison between *naïve* (*a priori* scheduling assignment with multiple internal queues), *PJM* (single queue internal scheduler) and *pjm+opt* (single queue and variable resource allocation) configurations. The application is run for 5 iterations of the HMC workflow. (a, b) Influence of the internal scheduling method on the total runtime of the Nano-materials application. Two allocation sizes are tested: 20 nodes (left) and 100 nodes (right). (c, d) Detailed analytics from the PJM for the 100 node allocations with *pjm* (left) and *pjm+opt* (right) configurations only. That is time evolution of total allocation usage (top), number of running microscopic model simulations (middle) and average number of processes per running micro-scale simulation (bottom)

2.2. Red Blood Cell Suspension Application

In this example, we present the benefit of our scheduling method for HMM applications with a surrogate model. Resolving the properties of whole blood on the multiple spatial scales present in the human body is a significant challenge for a single blood-flow model. Whole blood is a suspension of deformable red blood cells (RBCs) and as a result has many non-Newtonian flow properties, for example shear thinning. On spatial scales $\leq 300\mu$m phenomena like the Fåhraeus-Lindqvist effect, the RBC free layer and platelet margination require cell resolved blood flow models which take into account the material properties of the RBCs that influence the emergent rheology of blood. On scales $> 300\mu$m cell resolved models quickly become too expensive and are not viable options, so continuous models are employed. The ultimate goal of developing an HMM blood flow model is to simulate blood on scales where only continuous models have been

applied, informed by the cell resolved nature of whole blood. In this section we present a place holder 3D HMM blood flow application to highlight the benefit of our scheduling method.

In this application, blood-flow is modelled on the macro-scale as a continuous fluid using the lattice Boltzmann method (LBM). Also on the macro scale is an advection-diffusion solver which models the evolution of red blood cell volume fraction (i.e. the haematocrit) profiles. The micro scale is modelled with the cell resolved blood flow model HemoCell, in which plasma is modelled by the LBM, the mechanical model of the RBCs are described by a discrete element method and are couple to the plasma via the immersed boundary method [34, 35]. From each local haematocrit and shear rate combination on the macro scale, cell resolved micro scale will simulate perfect sheared environments using Lees-Edwards boundary conditions [24]. Local viscosity and diffusion coefficients will be measured on the micro scale and passed up to the macro scale. The local viscosity will be passed to the LBM fluid solver, and the diffusion coefficient will be passed to the advection-diffusion solver. On the macro scale the LBM fluid solver will take the local viscosities, and the advection-diffusion solver the difusion coefficients, and both will step the macro simulation through the next time step calculating new shear rates and haematocrit profiles for the next iteration. The micro-scale models will resolve blood flow on spatial scales of $\sim 100\mu m$ and temporal scales of $\sim 10ms$, while the macro-scale models will resolve spatial scales of centimeters and temporal scales of seconds. A benefit of such an HMM model is that we will have on the largest scales a continuous blood flow solver which is informed by a micro scale cell resolved blood flow solver. This should lead to a better resolution of the transport of blood cells on the largest vessels found in the body. To avoid duplicating shear rate and local haematocrit microscale simulations, we aim to build a surrogate model (e.g., based on a Gaussian process) to conduct interpolations for similar parameters. This process decreases the required number of micro-scale models requested, which in turn decreases computation time. A schematic of this HMM blood flow model is shown in Fig. 10.

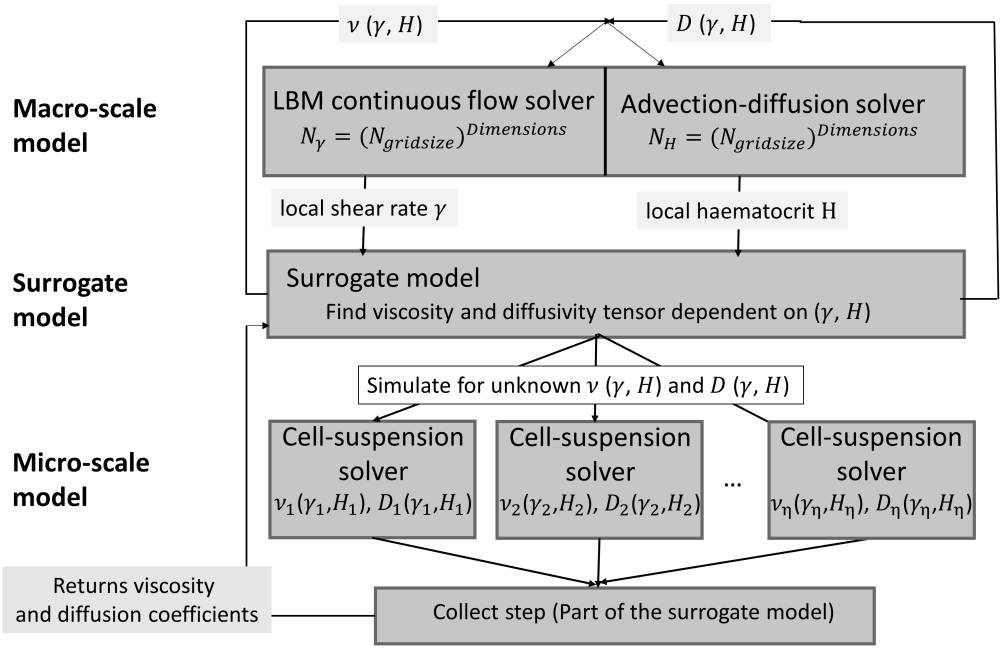


Figure 10. Main structure of the red blood cells HMM example used in this section exhibiting the different scales (macro and micro)

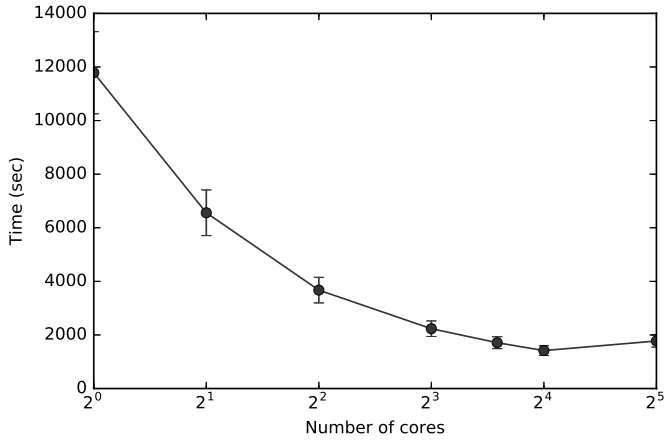


Figure 11. Average performance of the cell-suspension LBM solver, representing micro-scale models. In this example, the system size was 256^3 lattice units, and the total number of simulated cells was 16277

In this study, we replaced the actual surrogate model with the performance graphs in Fig. 5 and 6. The performance of the surrogate model will vary the number of micro-scale models needed for each macro-scale iteration significantly, which can lead to load imbalance and low utilisation of the available resources. Thus, the main target of the runtime HMC pattern software is to schedule this dynamically varying load of micro-scale models in an optimal way on the available resources.

To understand and improve the scheduling of these micro-scale jobs, we need to profile the average performance of the micro-scale models first. Figure 11 shows the execution time of the micro-scale model as a function of the number of processors. In the micro-scale, we simulated the red blood cells and platelets suspensions in plasma using HemoCell [35]. Total system was 256^3 lattice units, and the simulation contained 16277 red blood cells. In this specific benchmark, it is clear that the minimal execution time is obtained using 16 processors. Increasing the number of processors to more than 16 actually increases the execution time. This is a well-known phenomenon in strong scaling of parallel applications, and is due to increasing the overhead time as the number of processors increases. Thus, in this example, the boundary limits for the number of processors for the micro-scale models are $P_{min} = 1$ and $P_{max} = 16$.

To confirm the choice of farming a large number of micro-scale models using the least number of processors, in addition to the discussion presented in Section 1.2.1, consider the following example. Assume that $P_\mu = 1024$ processors are reserved for micro-scale models and that in one macro-scale iteration it is required to run $\eta(t) = 4096$ jobs. This is the first phase of the surrogate model performance (i.e. $\eta(t)P_{\min} > P_\mu$). By running the simulation using one core/node per micro-scale model, four batches are needed to finish, where each batch will take ~ 3 core hours. This means that a total of ~ 12 core hours of computing is required. On the other hand, if 16 processors per micro-scale model were to be used, then 64 batches would be needed at 30 min each, resulting in a total of ~ 32 core hours of computational time.

2.2.1. Results

In the following computer simulations the runtime and utilisation for the four cases of performance shown in Fig. 5 and 6 are measured. Resource utilisation U is defined as $U = R/C$,

where R is the actual used number of processors, and C is the capacity, which is the total number of processors available for the job. The utilisation was measured as a function of wall clock time during the execution.

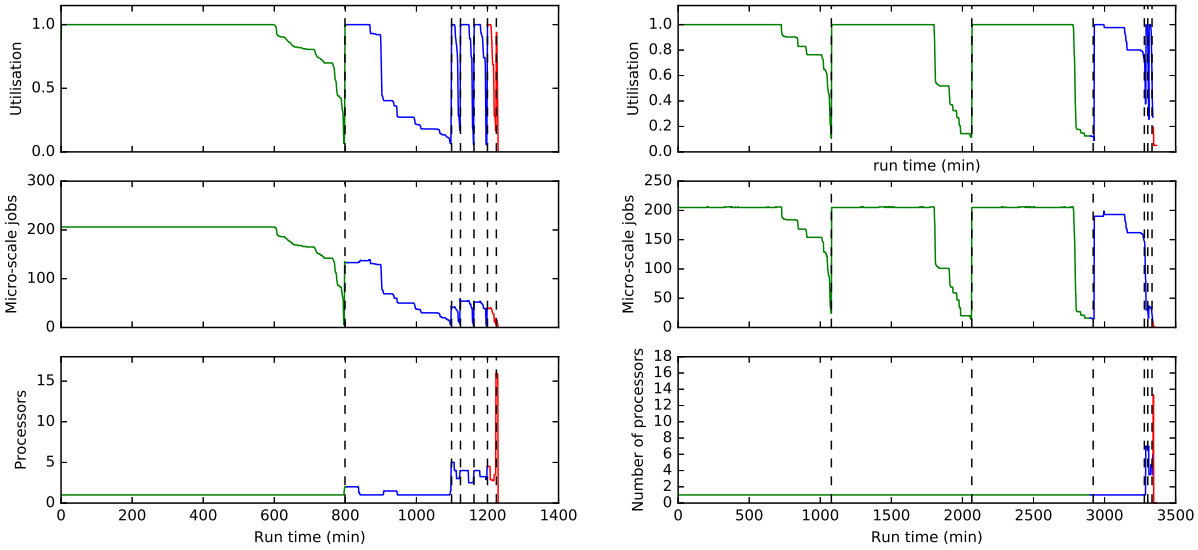
In all these simulations, the number of cores allocated for the macro-scale model was $P_M = 8$, for the surrogate model $P_S = 1$ and for the HMC manager $P_H = 1$. The total number of cores available for the micro-scale models was $P_\mu = 206$. The degree of freedom selected was $DoF = 48818$.

The QCG pilot job used in the experiments was a normal job submitted to Eagle to reserve a set of resources. After reservation, these resources were then managed using a python script, the pilot job manager. In this script the user can launch, request and kill jobs dynamically. In our benchmarks, the pilot job reserves a number of processors first. Then, in the pilot job manager, the macro-scale model and the HMC manager are submitted. The macro-scale, after an iteration requests a number of parameters from the surrogate model. The surrogate model looks in the database, interpolates the missing quantity and requests to run a number of micro-scale models for the missing quantities. In the benchmarks, we mimic this operation by using the performance of the surrogate model. The number of micro-scale models and the available resources are then sent to the HMC manager to suggest the right distribution of the resources to the pilot job manager. Also, it acts to different phases of the HMM application accordingly. The pilot job manager then executes the submodels utilising an internal queue on the required resources, gathers the values from the micro-scale jobs, and sends them back to the surrogate model.

Figure 12 (top panels) presents the utilisation of the system, with assistance from the HMC manager and the surrogate model that starts from scratch. The utilisation of the performance of the surrogate model presented in Fig. 5a (model = scratch, good performance) is shown here in Fig. 12a (top panel), while the utilisation of the performance of the surrogate model presented in Fig. 5b (model = scratch, poor performance) is shown in Fig. 12b (top panel).

In Fig. 12a (top panel), in the first macro-scale iteration, a large number of micro-scale jobs are executed, each executing with $P_{\min} = 1$ in a first-in, first-out queue (in our method, we use sub-queues in the pilot job rather than batches). The utilisation in this phase is high because we simply exploit all the available resources. At the outset the utilisation is one, which means that all 206 processors are used as shown in Fig. 12a (middle panel). After a while, between 18–32 jobs are completed, shown as the first few small decreases of the green line.

The blue line in Fig. 12a (top panel), for the second to fifth macro-scale iterations, shows $P_{\min}\eta(t) < P_\mu < P_{\max}\eta(t)$ phase. The utilisation at the outset of these iterations is one, because all the processors reserved for micro-scale models are used by running the micro-scale models with $\eta(t), \lceil P_\mu/\eta(t) \rceil$. For example, in the second macro-scale iteration (which lasted from 800 to 1100 minutes of the runtime), 142 jobs were run with $\sim 1-2$ cores, each shown in Fig. 12a the middle and lower panels, respectively. As a result of different micro-scale model execution times with different numbers of processors per micro-scale model invocation, we notice a gradual decrease in the utilisation. In this situation, an internal mechanism could be implemented to use the available cores to refine the surrogate model by proactively (i.e. not informed by the macro-scale model) executing micro-scale models in yet unexplored regions of the micro-scale input parameter space. The gradual decrease in the second macro-scale iteration does not occur in the following iterations (macro-scale iterations 3 to 5), because the number of micro-scale



(a) (b)

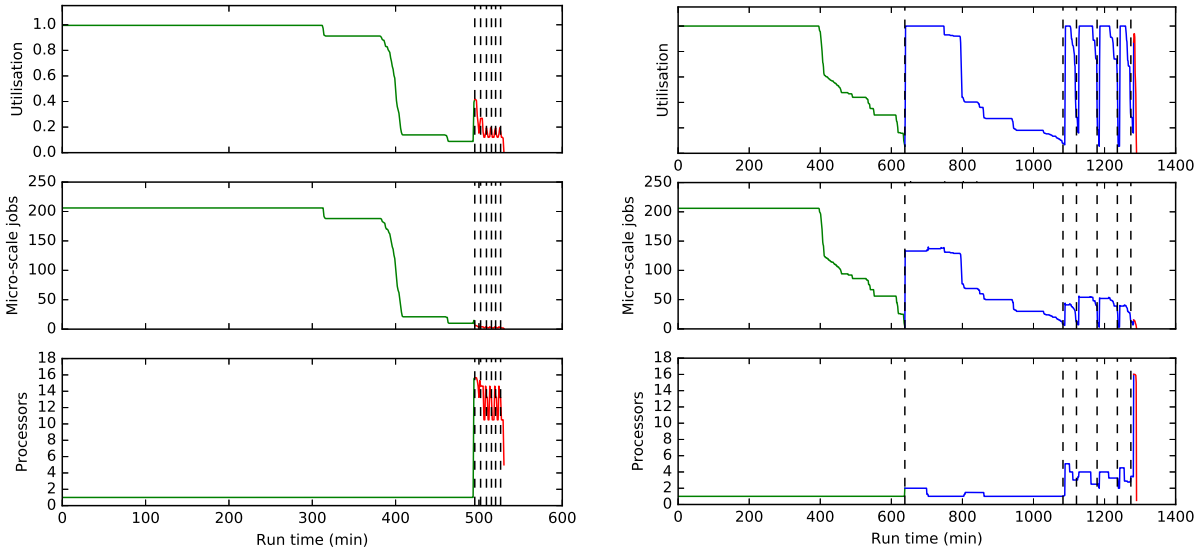
Figure 12. Utilisation of the red blood cell HMM application using PJM (top panels) using the surrogate model from scratch, as shown in case (a) in Section 1.2.2. Good performance level manifest on the left (model = scratch, good performance), while poor performance level is displayed on the right (model = scratch, poor performance). The corresponding number of micro-scale jobs (middle panels) and number of cores per micro-scale job (lower panels) were decided by the HMC manager. The colours represent the phases, where green is the first phase, blue – the second, and red is the third. The dashed lines represent the macro-scale iterations

model jobs in iteration 4, for example, is smaller (54 running with ~ 4 processors shown in blue in Fig. 12a (middle and lower panels)).

The last macro-scale iteration in Fig. 12a (top panel) falls in the third phase, where the number of micro-scale jobs is only 12, and the number of cores per micro-scale job run is $P_{\max} = 16$. As is shown by the red line in the graph and also the red lines in Fig. 12a (middle panel), we run all the micro-scale models in one fast run. This phase is fast, but the utilisation remains low. In this state, as discussed, we could release the unused processors as the surrogate is mature enough to replace the need to generate new micro-scale jobs.

While the second case, Fig. 12b (top panel), shows similar behaviour, the surrogate model is not as effective as for the first case. Figure 12b (middle panel) illustrates the corresponding number of micro-scale jobs and the average number of processors per micro-scale jobs. Here, we need to generate more micro-scale jobs to sample parameter space at the outset. This example also shows that after running a large number of micro-scale jobs for the first three macro-scale iterations, we reached a steady level where we ran 35 micro-scale jobs in the second phase and ~ 2 micro-scale jobs in the third phase utilising a dynamic number of cores/nodes.

When a previously developed surrogate model is used, the resulting utilisation of the system is as shown in Fig. 13 (top panel), and the corresponding number of micro-scale jobs and the average number of processors per micro-scale jobs are shown in Fig. 13 (middle and lower panels, respectively). This case shows the situation where the surrogate model must initially run a large number of micro-scale models at the outset to fill the database rapidly and then becomes sufficiently effective to replace the need for micro-scale models. The utilisation of the performance of the surrogate model presented in Fig. 6a (model = developed, good performance)



(a) (b)

Figure 13. Utilisation of the red blood cell HMM application using PJM (top panels) by using a previously developed model, as shown in study case (b) in Section 1.2.2. Good performance level manifest on the left (model = developed, good performance), while poor performance level is displayed on the right (model = developed, poor performance). The corresponding number of micro-scale jobs is shown in the middle panel. The lower panel shows the corresponding number of cores per micro-scale job decided by the HMC manager. The colours represent the phases, where green is the first phase, blue – the second, and red is the third. The dashed lines represent the macro-scale iterations

is shown here in Fig. 13a (top panel), while the utilisation of the performance of the surrogate model presented in Fig. 6b (model = developed, poor performance) is shown in Fig. 13b (top panel).

Figure 13 (top panels) appears similar to the previous case, but we notice that because the difference in the number of micro-scale jobs requested for each macro-scale step is high, there are more idle cores in the first phase than for the previous case. Here, we might consider returning the cores back to the system as expected, reaching the steady state in this case is faster than in the previous case, and in the subsequent runs the surrogate model replaces the need for generating new micro-scale jobs, up to where the first few macro-scale iterations are completed as shown in Fig. 13 (middle panels).

Finally, note that in production runs, the number of iterations on the macro-scale will be much larger, and if the surrogate model is performing very well, the overall utilisation can be very small (as, e.g., reported in ref [22]). Once the HMC run is running steadily in phase 3, the HMC manager should return resources to the system.

Conclusion

We have illustrated the mechanism of running HMM applications using pattern software. In this paper, two sources of load imbalance in HMC, namely variable execution time and dynamic number of micro-models, are investigated and a number of solutions are proposed.

First, the large number of micro-scale model simulations at each iteration of the HMC workflow requires them to be performed inside a single large resource allocation to avoid sig-

nificant and asynchronous queuing time. In turn, the variable execution time of the micro-scale simulation induced significant idle periods on part of the large allocation. A pilot job manager mechanism was introduced to handle internal queuing and execution of these simulations. The flexibility of how the PJM mechanism in comparison to a pre-assigned rigid batching system enables drastic improvements of core utilisation lead to reducing of overall ideal time by up to 70%. The improvement was particularly significant on smaller allocations. In addition, facilitated by the use of a PJM, an optimisation layer has also been used to tackle directly the variability of execution time. Scaling the individual resource sub-allocation of each micro-scale simulation, based on predicted execution time, load balancing was even further reduced with 20% shorter runtime. The adaptive resource allocation was maintained within the range of strong scaling of the model in order to preserve efficiency.

Second, we propose that the execution of the micro-scale models in the HMM application should be viewed as three distinct phases. The first phase arises when a very large number of micro-scale models ($P_\mu < \eta(t)P_{\max}$) needs to be executed. In this phase, the appropriate action is to conduct farming with $P_{\min}$ processors per micro-scale model. This reduces the overhead, as demonstrated mathematically and by means of computational simulations. The second phase occurs when ($P_{\max}\eta(t) < P_\mu < P_{\max}\eta(t)$). In this phase, the required micro-scale jobs are executed with $\lceil P_\mu/\eta(t) \rceil$ cores/nodes. During the last phase, when the number of micro-scale jobs requested is less than the total number of the available processors for micro-scale models, the most appropriate action is to run all the remaining jobs using $P_{\max}$ processors. The surrogate model is nearly or fully developed at this stage and can replace the need to generate micro-scale jobs efficiently. At this stage, certainly when many iterations of the macro-scale need to be simulated, releasing unused processors will result in an increase in utilisation and a reduction of the computational cost. An HMC assuming four different scenarios of surrogate models was executed. In each scenario, the number of micro-scale jobs generated at each macro-scale iteration is shown, the utilisation of the system using a pilot job and the HMC manager, with the focus on running the micro-scale jobs on the right resources.

Although farming independent jobs is a well-known method, the act of dynamically changing from one farming phase to another under the control of a dynamically evolving surrogate model, with its corresponding actions and decisions, is new. In our simulations, we substituted the actual implementation of Gaussian process regression of the surrogate model with the performance of a surrogate model, as observed in an HMM for predicting material properties [22]. This performance figure provides the required number of micro-scale jobs that are needed at each macro-scale iteration. Finally we note that combining the dynamically changing runtime of the micro-scale models and the dynamically changing number of micro-scale models to be executed has not yet been addressed in the paper. However, the combination of the scheduling mechanisms discussed in Section 1, and the overall HMC phases in Section 1.2.2, should be able to handle this situation as well.

Acknowledgments

We acknowledge partial funding from the European Union Horizon 2020 research and innovation programme under grant agreement No 671564 for the ComPat project (http://www.compat-project.eu/). SA acknowledges funding from King Abdulaziz City for Science and Technology (KACST), Saudi Arabia. P.V.C. thanks the MRC Medical Bioinformatics project (MR/L016311/1), the EU H2020 CompBioMed grant (http://www.compbiomed.eu, Grant

No. 675451) and funding from the UCL Provost. This research was also supported in part by the PLGrid Infrastructure including dedicated HPC resources at the Poznan Supercomputing and Networking Center.

References

1. Abdulle, A., Weinan, E., Engquist, B., Vanden-Eijnden, E.: The heterogeneous multiscale method. Acta Numerica 21, 1–87 (2012), DOI: 10.1017/S0962492912000025

2. Aktulga, H.M., Fogarty, J.C., Pandit, S.A., Grama, A.Y.: Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques. Parallel Computing 38(4), 245–259 (Apr 2012), DOI: 10.1016/j.parco.2011.08.005

3. Alowayyed, S., Piontek, T., Suter, J.L., Hoenen, O., Groen, D., Luk, O., Bosak, B., Kopta, P., Kurowski, K., Perks, O., Brabazon, K., Jancauskas, V., Coster, D., Coveney, P.V., Hoekstra, A.G.: Patterns for high performance multiscale computing. Future generation computer systems 91, 335–346 (2019), DOI: 10.1016/j.future.2018.08.045

4. Alowayyed, S.: Patterns for multiscale computing. Ph.D. thesis, University of Amsterdam (2018), `http://hdl.handle.net/11245.1/9cf904f8-fcc7-4b8a-a105-622a865359d8`

5. Alowayyed, S., Groen, D., Coveney, P.V., Hoekstra, A.G.: Multiscale computing in the exascale era. Journal of Computational Science 22, 15–25 (2017), DOI: 10.1016/j.jocs.2017.07.004

6. ASTM: Test method for short-beam strength of polymer matrix composite materials and their laminates, DOI: 10.1520/d2344_d2344m-16, accessed: 2019-11-29

7. Axner, L., Bernsdorf, J., Zeiser, T., Lammers, P., Linxweiler, J., Hoekstra, A.G.: Performance evaluation of a parallel sparse lattice Boltzmann solver. Journal of Computational Physics 227(10), 4895–4911 (2008), DOI: 10.1016/j.jcp.2008.01.013

8. Bangerth, W., Hartmann, R., Kanschat, G.: Deal.II–A general-purpose object-oriented finite element library. ACM Trans. Math. Softw. 33(4) (Aug 2007), DOI: 10.1145/1268776.1268779

9. Borgdorff, J., Ben Belgacem, M., Bona-Casas, C., Fazendeiro, L., Groen, D., Hoenen, O., Mizeranschi, A., Suter, J.L., Coster, D., Coveney, P.V., Dubitzky, W., Hoekstra, A.G., Strand, P., Chopard, B.: Performance of distributed multiscale simulations. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 372 (2014), DOI: 10.1098/rsta.2013.0407

10. Borgdorff, J., Lorenz, E., Bona-Casas, C., Hoekstra, A.G., Falcone, J.L., Chopard, B.: Foundations of distributed multiscale computing: Formalization, specification, and analysis. J. Parallel Distrib. Comput. Journal of Parallel and Distributed Computing 73(4), 465–483 (2013), DOI: 10.1016/j.jpdc.2012.12.011

11. Borgdorff, J., Mamonski, M., Bosak, B., Kurowski, K., Belgacem, M.B., Chopard, B., Groen, D., Coveney, P., Hoekstra, A.: Distributed multiscale computing with MUSCLE 2, the Multiscale Coupling Library and Environment. Journal of Computational Science 5(5), 719–731 (2014), DOI: 10.1016/j.jocs.2014.04.004

12. Bouvard, J.L., Ward, D.K., Hossain, D., Nouranian, S., Marin, E.B., Horstemeyer, M.F.: Review of hierarchical multiscale modeling to describe the mechanical behavior of amorphous polymers. Journal of Engineering Materials and Technology 131(4) (Sep 2009), DOI: 10.1115/1.3183779

13. Cheng, L.T., Weinan, E.: The heterogeneous multi-scale method for interface dynamics. Contemporary mathematics. 330, 43–54 (2003), DOI: 10.1007/978-94-007-0412-1_18

14. Chopard, B., Falcone, J.L., Kunzli, P., Veen, L., Hoekstra, A.: Multiscale modeling: recent progress and open questions. Multiscale and Multidiscip. Model. Exp. and Des. Multiscale and Multidisciplinary Modeling, Experiments and Design 1(1), 57–68 (2018), DOI: 10.1007/s41939-017-0006-4

15. Coveney, P.V., Boon, J.P., Succi, S.: Bridging the gaps at the physics–chemistry–biology interface. Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences 374(2080), 335–339 (2016), DOI: 10.1098/rsta.2016.0335

16. Engquist, B., Lötstedt, P., Runborg, O.: Multiscale modeling and simulation in science. Springer Science & Business Media, Heidelberg (2009), DOI: 10.1007/978-3-540-88857-4

17. Hoekstra, A.G., Chopard, B., Coster, D., Portegies Zwart, S., Coveney, P.V.: Multiscale computing for science and engineering in the era of exascale performance. Phil. Trans. R. Soc. A Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 377(2142), 20180144 (2019), DOI: 10.1098/rsta.2018.0144

18. Hoekstra, A.G., Chopard, B., Coveney, P.V.: Multiscale modelling and simulation: a position paper. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 372, 30377–30385 (2014), DOI: 10.1098/rsta.2013.0377

19. Hoekstra, A.G., Portegies Zwart, S., Coveney, P.V.: Multiscale modelling, simulation and computing: from the desktop to the exascale. Phil. Trans. R. Soc. A Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 377(2142), 20180355 (2019), DOI: 10.1098/rsta.2018.0355

20. Knap, J., Spear, C.E., Borodin, O., Leiter, K.W.: Advancing a distributed multi-scale computing framework for large-scale high-throughput discovery in materials science. Nanotechnology 26(43), 434004–434016 (2015), DOI: 10.1088/0957-4484/26/43/434004

21. Kopta, P., Bosak, B.: QCG-PilotJob. `https://github.com/compat-project/QCG-PilotJob` (2018)

22. Leiter, K.W., Barnes, B.C., Becker, R., Knap, J.: Accelerated scale-bridging through adaptive surrogate model evaluation. Journal of Computational Science 27, 91–106 (2018), DOI: 10.1016/j.jocs.2018.04.010

23. Lorenz, E., Hoekstra, A.G.: Heterogeneous multiscale simulations of suspension flow. Multiscale Modeling and Simulation 9(4), 1301–1326 (2011), DOI: 10.1137/100818522

24. Lorenz, E., Hoekstra, A.G., Caiazzo, A.: Lees-edwards boundary conditions for lattice Boltzmann suspension simulations. Phys. Rev. E Physical Review E 79(3) (2009), DOI: 10.1103/PhysRevE.79.036706

25. Luk, O., Hoenen, O., Perks, O., Brabazon, K., Piontek, T., Kopta, P., Bosak, B., Bottino, A., Scott, B.D., Coster, D.P.: Application of the extreme scaling computing pattern on multiscale fusion plasma modelling. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 377(2142), 20180152 (2019), DOI: 10.1098/rsta.2018.0152

26. Nikishova, A., Veen, L., Zun, P., Hoekstra, A.G.: Uncertainty quantification of a multiscale model for in-stent restenosis. Cardiovascular Engineering and Technology (2018), DOI: 10.1007/s13239-018-00372-4

27. Plimpton, S.: Fast parallel algorithms for short-range molecular dynamics. Journal of Computational Physics 117(1), 1–19 (Mar 1995), DOI: 10.1006/jcph.1995.1039

28. Sloot, P.M.A., Hoekstra, A.G.: Multi-scale modelling in computational biomedicine. Briefings in bioinformatics 11(1), 142–152 (2009), DOI: 10.1093/bib/bbp038

29. Turilli, M., Santcroos, M., Jha, S.: A comprehensive perspective on Pilot-Job systems. ACM Computing Surveys 51(2), 1–32 (2018), DOI: 10.1145/3177851

30. Vassaux, M., Richardson, R., Coveney, P.V.: The heterogeneous multiscale method applied to inelastic polymer mechanics. Philos. Trans. R. Soc. A Math. Phys. Eng. Sci. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 377(2142) (2019), DOI: 10.1098/rsta.2018.0150

31. Wang, C., Duan, Q., Gong, W., Ye, A., Di, Z., Miao, C.: An evaluation of adaptive surrogate modeling based optimization with two benchmark problems. Environmental Modelling and Software 60, 167–179 (2014), DOI: 10.1016/j.envsoft.2014.05.026

32. Weinan, E.: Principles of multiscale modeling. Cambridge University Press, Cambridge, New York (2011), DOI: 10.1063/PT.3.1609

33. Weinan, E., Engquist, B., Huang, Z.: Heterogeneous multiscale method: a general methodology for multiscale modeling. Physical Review B 67(9), 092101 (2003), DOI: 10.1103/PhysRevB.67.092101

34. Závodszky, G., van Rooij, B., Azizi, V., Hoekstra, A.G.: Cellular level in-silico modeling of blood rheology with an improved material model for red blood cells. Frontiers in physiology 8 (2017), DOI: 10.3389/fphys.2017.00563

35. Závodszky, G., van Rooij, B., Azizi, V., Alowayyed, S., Hoekstra, A.G.: Hemocell: a high-performance microscopic cellular library. Procedia Computer Science 108, 159–165 (2017), DOI: 10.1016/j.procs.2017.05.084

DOI: 10.14529/jsfi190403

Improving Reliability of Supercomputer CFD Codes on Unstructured Meshes

Andrey V. Gorobets[1], *Pavel A. Bakhvalov*[1]

The paper describes a particular technical solution targeted at improving reliability and quality of a highly-parallel computational fluid dynamics code written in C++. The code considered is based on rather complex high-accuracy numerical methods and models for simulation of turbulent flows on unstructured hybrid meshes. The cost of software errors is very high in large-scale supercomputer simulations. Reproducing and localizing errors, especially "magic" unstable bugs related with wrong memory access, are extremely problematic due to the large amount of computing resources involved. In order to prevent, or at least notably filter out memory bugs, an approach of increased reliability is proposed for representing mesh data and organizing memory access. A set of containers is proposed, which causes no overhead in the release configuration compared to plain arrays. At the same time, it provides throughout access control in the safe mode configuration and additional compile-time protection from programming errors. Furthermore, it is fully compatible with heterogeneous computing within the OpenCL standard. The proposed approach provides internal debugging capabilities that allow us to localize problems directly in a supercomputer simulation.

Keywords: CFD, supercomputer, unstructured mesh, data structure, MPI, OpenMP, OpenCL.

Introduction

The importance of software reliability for supercomputer simulations can hardly be overestimated. Software errors that appear in large-scale simulations involving thousands of processor cores cause the loss of many CPU hours and large labor costs. Those readers who deal with such simulations using in-house codes most likely have noticed that the work on debugging parallel applications (often in conditions of strict time limits and burning deadlines) is especially stressing and hard. If an error shows up in a large-scale simulation, it means that it has infiltrated through the quality assurance procedure based on numerous test cases and preliminary simulations on coarse meshes. Therefore, most likely, this is a rather insidious bug that will be difficult to catch. Such errors typically exhibit unstable behavior that makes reproduction and localization difficult. The immense amount of suffering from debugging parallel applications that the authors experienced motivated the present work, which is aimed at increasing reliability of supercomputer simulation software.

A way to represent mesh data in a computational fluid dynamics (CFD) code is proposed. It is designed primarily for simulations using static unstructured meshes, dynamic meshes with constant topology and moving meshes with sliding interfaces. It is assumed that the code has release and safe mode build configurations, both using compilation with full optimization. The latter enables low-level checks that can affect performance. The following requirements were imposed on a set of containers for mesh data:

- no observable overhead in the release configuration compared to plain arrays;
- tolerable overhead in the safe mode configuration (say 20–30%, not several times);
- full access control in the safe mode configuration;
- informative diagnostics that reports where exactly the error occurred in the code and in which container;

[1]Keldysh Institute of Applied Mathematics (RAS), Moscow, Russian Federation

- only plain arrays at the backend of containers that provide direct compatibility with linear algebra libraries and computing on massively-parallel accelerators, such as GPUs.

One way or another, containers and structures for the mesh data are represented in every CFD code. These components play an important role in large-scale supercomputing. However, implementation details are in most cases hidden and not available in the literature. Typically, articles describing CFD software provide only a brief description of the general approach (see [11], for instance). In order to look at implementation details, we can anatomize third-party open source codes and learn how the data structure is organized. But this usually takes some effort. For instance, we have studied the Nectar++ open source CFD code [2]. It operates with containers, which are template classes based on plain pointers accessed via plain integers. There is no extra error protection.

There are multiple scientific groups working on general-purpose mesh data frameworks for scientific computing. Among them, the SIGMA team from the Argonne National Lab with the MOAB library [10] and an array-based mesh data structure [5]. There are relevant works in the Los Alamos National Laboratory [6, 7] for complex multi-material and geophysical mesh-based applications, respectively.

Another representative example of open source C++ software for scientific computing using mesh methods can be found in [4]. A brief overview of the relevant software is also presented there. This platform, called INMOST, has rather complicated containers and data structure due to its high generality. This may lead to notable overhead compared to narrower application-specific implementations. Also the low-level representation of data storage appears to be too complex for using in GPU computing.

Further examples of data structures for scientific computing can be found in [1, 12]. Implementation details are described in these works, but the proposed approaches do not provide additional error protection.

In the present work, we propose a simple approach for organizing mesh data in C++ that gives us additional correctness checks while causing no overhead in the release configuration. The containers have only plain pointers (1D vectors) at the backend. This ensures direct compatibility with heterogeneous computing within the OpenCL standard.

The following sections provide implementation details and representative code fragments that can help readers implement the proposed approach or some particular ideas in their CFD codes. In the next section, we define what the basic data for a mesh method consist of. Section 2 is devoted to containers that store these data. In Section 3, we address some complications, such as combining cell-centered and vertex-centered approaches in one code. Section 4 covers parallel computing aspects. Details on implementation of runtime diagnostics and memory allocation are given in Section 5. Performance evaluation is given in Section 6. Finally, the results of the study are summarized and the conclusions are given in the last section.

1. Basic Data for a Mesh Method

For the sake of simplicity, let us consider an unstructured hybrid mesh with a constant topology (only coordinates of nodes may change). There are numerous kinds of mesh objects in the computational domain (we say a kind, not a type, to avoid confusion with data types stored in containers). For instance, in our codes we have more than 10 types of objects, including: volume mesh nodes, edges, elements, faces, cells; boundary nodes, edges, faces; sliding interface nodes, edges, faces.

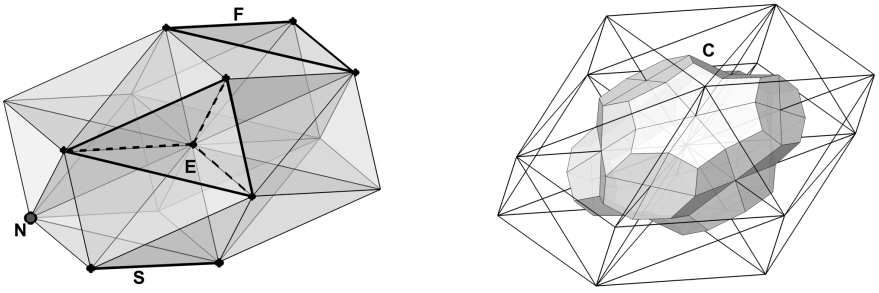
Note that there is a special kind, the set of cells (or control volumes, in other words), that introduces even more chaos and disorder. It switches between other kinds depending on where the mesh functions are defined. For vertex-centered schemes the set of cells is equal to the set of nodes, for element-centered schemes to the set of elements, respectively.

Furthermore, in computations we may need to know the list of adjacent objects of some kind for each object of the same kind or some other kind. It results in tens of so called topology containers that store this adjacency data. Since there are so many different kinds of mesh objects and relations between them, it is easy to make a mistake (and we often did it) by confusing types of objects when accessing containers.

To simplify the notation and explanation, let's limit ourselves just to the following few kinds of objects (denoted by one letter):

- **E** – mesh *E*lements;
- **N** – mesh *N*odes (vertices);
- **S** – mesh edges (in other words, line *S*egments);
- **F** – *F*aces of mesh elements.

Example of adjacent mesh objects is shown in Fig. 1.



(a) Tetrahedrons adjacent to one node (b) A vertex-centered control volume

Figure 1. Fragment of a tetrahedral mesh illustrating adjacent mesh objects

Then, let us denote different kinds of topology with two-letter notation, since for each object of the first kind the list of adjacent objects of the second kind is stored. The basic adjacency data comes from the main mesh topology (stored on disk):

- **EN** – for each mesh element stores the indices of its nodes.

The other topologies are derived from it:

- **EF** – for each element stores the indices of its faces;
- **FN** – for each face stores the indices of its nodes;
- **SN** – for each segment stores the indices of its nodes;

and, similarly, **ES**, **FS**.

Then, the following derivative adjacency data we can denote as the inverse topology:

- **NE** – for each node stores the indices of the elements it belongs to;
- **NS** – for each node stores the indices of the segments it belongs to;
- **FE** – for each face stores the indices of the elements it belongs to;

and, similarly, **NF**, **SF**, **SE**.

Then, there are adjacency data for elements of the same kind:

- **EE** – stores for each element the list of indices of the elements that share faces with this element;

- **NN** – for each node stores the list of indices of the nodes connected by an edge to this node.

This list is not complete. More topologies can be added for other combinations, if needed. Note that we have not considered the boundary surface so far, it will be addressed later.

Finally, data sets (mesh functions, structures of object properties, etc.) can be associated with the corresponding kinds of mesh objects or adjacency relations between them (such as non-zero entries of a sparse matrix that arise from the spatial discretization).

2. Basic Containers for Mesh Data

Since there is a chaos of numerous kinds of mesh objects and relations between them, some extra error protection mechanism would be very welcome. Basic index range check when accessing an array is insufficient in this case. In order to distinguish data associated with sets of objects of different kinds, we need a list of all possible kinds, an index that knows its kind, and an array that can be accessed only with an index of the proper kind.

In our simplified representation (Fig. 2), the list of possible kinds of mesh objects is given by the enumeration *tIdxKind*. The index class that knows its kind is just a wrapper around an integer value (can be 32 or 64 bits). The template class *tIdx* supplements our index with a kind using the *tIdxKind* enumeration. In doing so, an index of some kind can be used to access only arrays of the same kind. Violation of the kind will result in error at compilation time.

```
typedef int tInt; // Integer index value (to switch 32/64 bit if needed)

// Kinds of mesh elements
enum tIdxKind{ IDX_N=0/*nodes*/,IDX_E=1/*elements*/,IDX_S=2/*segments*/,IDX_F=3/*faces*/};

template<tIdxKind I> class tIdx { // Index of a given kind - wrapper around integer value
    tInt i; // Index value
    inline tIdx(const tInt j){ i=j; } // private direct cast from int
public:
    inline tInt idx() const { return i;}
    inline tIdx(){ i=-1; } // initialization with an incorrect value by default
    inline tIdx<I>& operator=(const tInt j){     i=j;   return *this; }
    inline tIdx<I>& operator=(const tIdx<I>& j){ i=j.i; return *this; }
    inline tIdx<I> operator+(tInt j) {  return (i+j); }
    inline tIdx<I>& operator++() { ++i; return *this; }
//  ... whatever other necessary functions and operators
};

// Error processing
int Crash(const char *fmt,...); // (MPI)Abort with message to log file, stdout and stderr
#ifdef SAFE_MODE // Safe assertion - to be placed in performance-critical places
  #define SAFE_ASSERT(X,...) if(!(X)) exit(Crash(__VA_ARGS__)); //enabled in safe-mode only
#else                          // Via exit – informs compiler/code analyzer about exit point
  #define SAFE_ASSERT(X,...) // disabled in release configuration
#endif
// Basic assertion - to be placed at upper level (where checks don't affect performance)
#define ASSERT(X,...) if(!(X)) exit(Crash(__VA_ARGS__));
#define ULL(X) ((unsigned long long)X) // Use %llu to print indices

// Memory allocations
template <typename T> T* GimmeMem(size_t N, const char* label=NULL); // Allocation wrapper
template <typename T> void FreeMem(T* &ptr); // Deallocation wrapper
```

Figure 2. Code fragments that illustrate index types, error handling and memory allocation definitions

There are explicit checks that ensure that the actual index value fits the array range. The runtime error handling is implemented in two configurations: safe mode and release. The safe mode configuration enables all the low-level checks in performance-critical areas (*SAFE_ASSERT*

definition in Fig. 2). The release configuration only performs upper-level checks (*ASSERT* definition) that don't affect performance. If some assertion fails, the *Crash* function is called. It prints complete diagnostics information and aborts the execution. Further details will be given in Section 5.

An array of values for a set of mesh objects of a certain kind is represented as a template class wrapped around a plain pointer. This class, denoted as *tArray* (see Fig. 3), knows the size of the array and its name and provides explicit access check at runtime. Access to elements of a *tArray* object via *operator[]* is only allowed for an index of the same kind defined by the first template parameter. This prevents confusing kinds of sets at compilation time. Source code fragments with selected relevant methods are shown in Fig. 3.

```
template <tIdxKind I, typename T> class tArray { // Basic 1D array
protected:
    T *VV;       // Pointer to data vector
    size_t NN;   // Size
    string name; // Text label (for memory reports and error messages)
    inline void init_vec(string lbl=""){ NN=0; VV=NULL; name=lbl; } // Defaults
public:
    inline void Alloc(tIdx<I> n, string lbl=""){
        ASSERT(NN==0,"tArray::Alloc %s: already allocated", lbl.c_str());
        ASSERT(n>0 , "tArray::Alloc %s: wrong size %llu",   lbl.c_str(), ULL(n));
        VV=GimmeMem<T>(n.idx(), lbl.c_str()); NN=(size_t)n; name = lbl;
    }
    inline void Dealloc(){ if(NN>0 && VV) FreeMem(VV); init_vec(""); }

    tArray(string lbl=""){ init_vec(lbl); }
    tArray(tIdx<I> n, string lbl=""){ init_vec(); Alloc(n, lbl); }
   ~tArray(){ Dealloc(); }

    inline T &operator[](tIdx<I> i){
        SAFE_ASSERT(i.idx()>=0 && i.idx()<(tInt)NN, "tArray %s[%llu] out of size %llu",
                    name.c_str(), ULL(i.idx()), ULL(NN));
        return VV[i.idx()];
    }
//  ... whatever other necessary functions and operators
};
```

Figure 3. Code fragments of the 1D array template class that stores data for a set of mesh objects of a certain kind

2D arrays (in other words, block arrays) are represented by the *tBlockArray* class (see Fig. 4). It is assumed that the first index of the 2D array is the number of the block that corresponds to the mesh object in the set, and the second index is the position inside the block. Blocks store multiple values per mesh object, for instance, values of mesh functions defined in objects of a certain kind. This template class has two parameters, the kind of objects and the type of values. The underlying data storage is also a plain array. In the release configuration *operator[]* just returns the pointer to the requested block. In the safe mode configuration it returns an object of the *tBlock* class. This class is a wrapper around the pointer to the block that performs access correctness check inside the block.

Finally, we need block arrays with variable block size for adjacency data (mesh topology, portraits of sparse matrices, etc.) from Section 1. Such a container is represented by the *tVBlockArray* class (see Fig. 5). Its implementation is similar to block arrays, *operator[]* also returns a *tBlock* object. The difference is that there is a plain CSR-based (or whatever else suitable format) structure behind this wrapper.

In summary, the simplified set of containers consists of the *tArray* class for 1D arrays, the *tBlockArray* and *tVBlockArray* classes for 2D arrays with fixed and variable block sizes,

```
template <typename T> class tBlock { // A block of a block array (return type for operator[])
private:
  T *V; // Data pointer
  const char *name; // Pointer to parent object's name
  int M; // Block size (its not so big, 32-bit is ok)
  inline tBlock(){ M=0; V=NULL; } // Forbidden
public:
  inline tBlock(const tBlock<T> &b){ M=b.M; V=b.V; name=b.name; }
  inline tBlock(int m, T *v, const char *lbl=NULL){ M=m; V=v; name=lbl; }
  inline T& operator[](int i){
    SAFE_ASSERT(i>=0 && i<M, "tBlock %s[][%d] out of size %d", name, i, M);
    return V[i];
  } // ... other operators and functions ...
};

#ifdef SAFE_MODE
#define BLOCK(T) tBlock<T> // A block with safe mode access check
#define PTR2BLOCK(T,M,P,L)/*Pointer-to-block*/ tBlock<T>(M/*block size*/,P/*pointer*/,L/*label*/)
#else
#define BLOCK(X) X* // A plain pointer in release
#define PTR2BLOCK(T,M,P,L) P
#endif

template <tIdxKind I, typename T> class tBlockArray : public tArray<I,T>{
protected:
  tInt N; // Number of blocks
  int M;  // Block size
public:   // ...
  inline BLOCK(T) operator[](tIdx<I> i){
    SAFE_ASSERT((i>=0 && i<N),"tBlockArray %s[%llu] out of size %llu",
                tArray<I,T>::name.c_str(), ULL(i.idx()), ULL(N));
    return PTR2BLOCK(T, M, (tArray<I,T>::VV+(size_t)(i.idx()*M)), (tArray<I,T>::name.c_str()));
  } // ... other operators and functions ...
};
```

Figure 4. Code fragments that illustrate how block arrays work

```
template <tIdxKind I, typename T> class tVBlockArray : public tArray<I,T>{
protected:
   tInt N;   // Size - number of blocks (rows)
   tInt *IA; // CSR-like block (row) pointer offsets (of N+1 size)
public:      // ...
   inline int BlockSize(const tIdx<I> i)const{
      SAFE_ASSERT(i>=0&&i<N, "tVBlockArray::BlockSize %s[%llu] out of size %llu",
                  tArray<I,T>::name.c_str(), ULL(i.idx()), ULL(N));
      SAFE_ASSERT(IA[i.idx()+1]-IA[i.idx()]>=0 && IA[i.idx()]>=0,
                  "tVBlockArray::BlockSize %s: error in IA", tArray<I,T>::name.c_str());
      return (int)(IA[i.idx()+1]-IA[i.idx()]);
   }
   inline BLOCK(T) operator[](tIdx<I> i){
      SAFE_ASSERT(i>=0 && i<N, "tVBlockArray %s[%llu] out of size %llu",
                  tArray<I,T>::name.c_str(), ULL(i.idx()), ULL(N));
      return PTR2BLOCK(T, BlockSize(i), (tArray<I,T>::VV+IA[i.idx()]), (tArray<I,T>::name.c_str()));
   }
   // ... other operators and functions ...
 };
```

Figure 5. Code fragments that illustrate how block arrays with variable block size work

respectively. These template wrappers around plain pointers provide complete access correctness check that includes range check for all indices (both indices of 2D arrays) at runtime and compile-time check of correctness of the kind. Illustrative code fragments are shown in Fig. 6 (note the node ranges at the top of the figure, which will be explained in Section 4).

3. Some Complications of the Data Structure

In the previous sections, we described a simplified model with just a few kinds of mesh objects. There are still some problems that need to be addressed, in particular, the representation of the boundary surface of the mesh and the data sets associated with the cells.

```
  tIdx<IDX_N> nn; // Number of nodes in MPI extended subdomain
  tIdx<IDX_N> n_beg, n_end; // Range for extended subdomain (owned+halo nodes)
  // Subsets (vertex-centered case, mesh decomposition for the nodal graph)
  tIdx<IDX_N> n_owned_beg, n_owned_end; // Range for owned nodes (inner+interface)
  tIdx<IDX_N> n_inner_beg, n_inner_end; // Range for inner nodes
  tIdx<IDX_N> n_iface_beg, n_iface_end; // Range for interface nodes
  tIdx<IDX_N> n_halo_beg,  n_halo_end;  // Range for halo nodes
  // ...
  tIdx<IDX_E> en; // Number of elements in MPI subdomain
  tIdx<IDX_E> e_beg, e_end; // Range for elements
  // ...
  tBlockArray<IDX_N, double> BAN; // Some block array over nodes
  tVBlockArray<IDX_E, tIdx<IDX_N> > EN_topo; // Elements-Nodes mesh topology
  // ...
  void DoSomething1(tIdx<IDX_N> in, BLOCK(double) v);
  void DoSomething2(tIdx<IDX_N> in, tIdx<IDX_E> ie, BLOCK(double) v);
  // ...
//for(tIdx<IDX_N> in=e_beg; in<n_end; ++in) // Compile-time error
//for(tIdx<IDX_E> ie=n_beg; ie<n_end; ++ie) // Compile-time error
  for(tIdx<IDX_N> in=n_beg; in<n_end; ++in) // Loop over nodes
      DoSomething1(in, BAN[in]);
  // ...
  for(tIdx<IDX_E> ie=e_beg; ie<e_end; ++ie){  // Loop over elements
      tIdx<IDX_N> in;
      BLOCK(tIdx<IDX_N>) nodes = EN_topo[ie]; // The nodes of the element
//    BLOCK(tIdx<IDX_N>) nodes = EN_topo[in]; // Compile-time error
//    BLOCK(tIdx<IDX_E>) nodes = EN_topo[ie]; // Compile-time error
      for(int j=0; j<EN_topo.BlockSize(ie); ++j){
//    for(int j=0; j<=EN_topo.BlockSize(ie); ++j){ // Runtime error at block access check
          in = nodes[j];
//        in = ie; // Compile-time error
//        DoSomething2(ie, in, BAN[in]);   // Compile-time error
          DoSomething2(in, ie, BAN[in]);
      }
  }
```

Figure 6. Code fragments that illustrate how basic containers for mesh data work

The objects of the mesh surface are represented with additional kinds, such as boundary nodes, boundary faces and elements. In this case, the data containers associated with the boundary cannot be accessed using indices of the volume mesh. On the other hand, a surface mesh is in fact a submesh of a volume mesh. Therefore, functions that work with the boundary must have access to the volume mesh data. For this purpose, we use index arrays that map the boundary numeration on the volume mesh numeration. Such an array of the *tArray* class stores for each boundary node its index in the volume mesh. Thus, access from the boundary submesh to the mesh data is organized with compile-time checking of correctness of the kind. Inverse mapping from the mesh to the boundary is implemented in a similar way. Inner nodes have negative values in this index array, which will result in an error if used in access to boundary data. The same approach is used for other submeshes, such as sliding interface zones. Note that we do not use associative containers, such as *std::map* and *std::unordered_map*, in order to preserve plain vectors (that we need for computing on GPU) and to avoid logarithmic cost of access (*std::map*).

Alternatively, the boundary surface could be represented with the same basic set of kinds. Then a boundary index cannot be distinguished from a volume mesh index. The correct access in this case would be ensured by using a specific structure of objects in which the boundary surface is separated from the volume mesh.

Finally, if the code deals with both vertex-centered and element-centered formulations, this makes a lot of problems. It is convenient to have a special kind for cells that is switching between nodes and elements, respectively. Since the choice of the formulation of the numerical scheme occurs at run time, we cannot control correctness of the kind at compilation time when

accessing data in cells. This fact complicates the implementation, making it more nasty, however, correctness of the kind is still ensured. To access data associated with nodes and elements using an index associated with cells (and vice versa), we use the corresponding template cast operator (see Fig. 7). Converting a cell index is only allowed into a node index or an element index. Incorrect conversion into one of these kinds will be prevented at run time by the assertion that the numerical scheme is of the proper formulation. Conversion to any other kind will be prevented at linking time (since such a function is not defined).

```
template<tIdxKind I> class tIdx {
// ...
public:
    template<tIdxKind TT> tIdx(const tIdx<TT> &I); // Conversion of kinds
};

bool IsCC(); // Checks if the numerical scheme is cell-centered.
// Allowed index kind conversions
template<>template<> inline tIdx<IDX_N>::tIdx(const tIdx<IDX_C>& j){SAFE_ASSERT(!IsCC(),"Wrong C2N"); i=j.idx();}
template<>template<> inline tIdx<IDX_C>::tIdx(const tIdx<IDX_N>& j){SAFE_ASSERT(!IsCC(),"Wrong N2C"); i=j.idx();}
template<>template<> inline tIdx<IDX_E>::tIdx(const tIdx<IDX_C>& j){SAFE_ASSERT( IsCC(),"Wrong C2E"); i=j.idx();}
template<>template<> inline tIdx<IDX_C>::tIdx(const tIdx<IDX_E>& j){SAFE_ASSERT( IsCC(),"Wrong E2C"); i=j.idx();}
// ...
tArray<IDX_C, double> A; // Array over cells
tIdx<IDX_N> in;
tIdx<IDX_E> ie;
tIdx<IDX_S> is;
A[ie] = 0.0;   // Correct access in cell-centered case
//A[in] = 0.0; // Runtime error - conversion exists but the assertion fails
//A[is] = 0.0; // Link-time error - conversion is declared but the function doesn't exist
```

Figure 7. Code fragments that illustrate how index conversion works

4. Parallel Computing

We use multilevel parallelization, which combines different parallel programming models: a message-passing model for cluster systems with distributed memory, a shared-memory model for multicore processors, and a stream processing paradigm for computing on accelerators.

The Message Passing Interface (MPI) is used on the first level. The mesh is decomposed into subdomains in order to distribute workload among supercomputer nodes. The cells (nodes or elements) are assigned to MPI processes. The part of the mesh that each MPI process works with consists of its own (local) cells and halo cells, which are cells from neighboring subdomains coupled by the scheme stencil with owned cells. The set of owned cells is into interface cells (that are coupled with halo cells) and inner cells. This separation of inner and interface cells is used for hiding communications behind computations by overlapping data exchanges needed for the interface with computations over the inner set.

Since MPI processes work with their local parts of the mesh, we need containers that provide mapping between local numbering of the MPI process and global numbering:

- the local-to-global (**L2G**) mapping array that stores for each cell its index in the whole global mesh;
- the global-to-local (**G2L**) array that stores the inverse mapping.

The former is needed to write simulation results, and the latter is needed to derive the local mesh data from the global mesh data stored on disk. Both L2G and G2L arrays are represented with *tArray* containers, and, of course, the global index has a different kind. The G2L array stores a negative index value for objects that do not belong to the MPI subdomain. This certainly leads to some memory overhead proportional to the global number of cells, but the access cost is constant. Again, as in the case of mapping for the boundary surface, we do not use associative

containers (maps). In addition, it should be noted that the *std::unordered_map* containers may also consume a lot of memory for hash data.

For a mesh that is so big (billions of cells) that the G2L array cannot fit in memory, we can use multilevel partitioning and distributed storage. Firstly, the mesh is decomposed into small enough pieces stored in separate files. Then these pieces are fragmented further into the needed number of MPI subdomains. In this case, none of the MPI processes works with the entire mesh at all, and the G2L mapping is limited to a sufficiently small subdomain of the upper-level decomposition. It should also be noted that the cells are reordered several times at the initialization stage. Firstly, the cells are ordered by the owned/halo sets, so that the owned cells always have a smaller index that the halo cells. Then, the halo cells are ordered ascending ranks of their owners, and the owned cells are ordered by inner/interface sets (see Fig. 8).

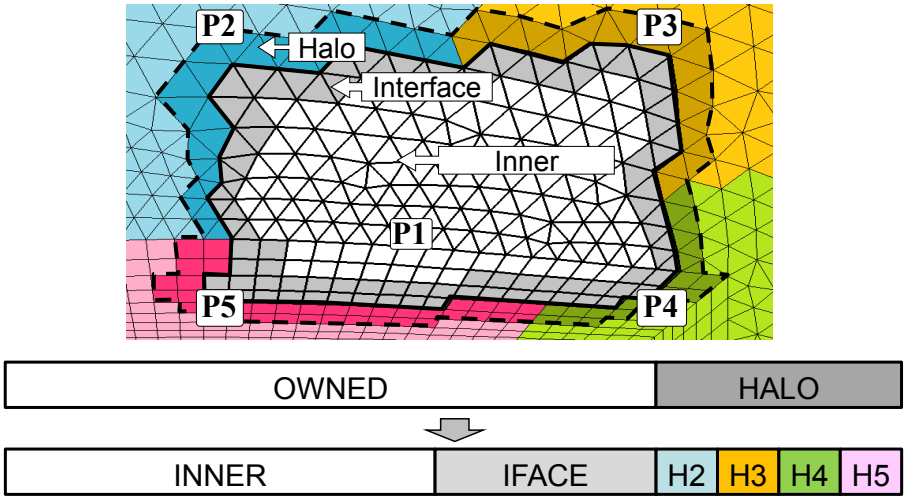


Figure 8. Reordering of cells in MPI subdomains

The OpenMP open standard is used for shared-memory parallel model on the second level. Instead of applying OpenMP loop parallelization directives, each MPI subdomain is further decomposed into second-level subdomains of OpenMP threads. Accordingly, the cells are reordered ascending their thread numbers. Then, each thread reorders its set of cells using Cuthill–McKee algorithm [3] in order to improve memory access pattern.

With this fixed decomposition, the sets of mesh objects associated with threads remain constant. This fact allows us to use the "first touch rule" in order to allocate data on NUMA system more efficiently. Each thread initializes its data in memory to help it be located in the nearest physical memory of the corresponding processor core. This would not make much sense in the case of loop parallelism, because loops for objects of different kind operate with data distributed differently among threads.

Finally, the OpenCL open standard is used for computing on accelerators of various architecture. Our containers are supplemented with OpenCL buffer handles for representing data on accelerators (devices). Since the containers are based on plain continuous 1D arrays, data can be transferred directly between the host and the devices without any conversions.

Regarding the representation of block arrays on GPUs, we use transposed access to block arrays in OpenCL kernels to improve memory access pattern (see [9]). The problem is that when blocks of a block array are associated with OpenCL threads, neighboring threads read their values with a non-unit stride (the distance is equal to the size of blocks), which is inefficient. In order to achieve coalescing of memory transactions, each local workgroup firstly reads its

fragment of a block array into the shared local memory linearly (neighboring threads read neighboring values in memory, as if the fragment was transposed) . Then the threads of the local workgroup access this fragment normally (as a block array), but in the fast local memory. This approach gives us an efficient memory access pattern and preserves compatibility with the host-side data structure. Additionally, fragments of local workgroups in block arrays can be aligned (using padding) to the cache line size, so that the memory access is properly aligned.

Further information on parallelization technology can be found in [8, 9].

5. Runtime Diagnostics and Memory Allocation

We have upper-level and lower-level checks in the code. In simple terms, functions that have a loop over mesh objects of some kind are upper-level functions. Functions that process a single mesh object are lower-level functions. The former are computationally heavy, while the latter are light enough so that checks may produce some notable overhead. Therefore, error handling is implemented via two macro definitions, *ASSERT* and *SAFE_ASSERT*, for upper and lower levels, respectively (Fig. 2). Lower-level checks are deactivated in release configuration, so we don't waste time on high-frequency checks if everything goes fine. These checks are always enabled for quality assurance (QA) testing, as well as in the case if something goes wrong in order to quickly catch the problem.

Access errors can be easily localized, as our basic containers have names, and we know which array is it. Furthermore, we use an internal call stack tracker. In the case of an error, it reports the list of called functions or even particular code blocks that lead to the crash place. It is implemented as a class *tStackAgent* (see Fig. 9) that operates with a global singleton object, which, in turn, just stores some thread-private list of text labels. The constructor of this class adds a label of a function into the list, and the destructor removes it. These "agents" are manually placed at the beginning of functions or important code blocks via macro definitions *INFORM_KGB* and *INFORM_KGB_LOW* for upper and lower levels, respectively. Lower-level agents are disabled in the release configuration.

```
struct tStackAgent{
    tStackAgent(const char *Text, int Level=1); // Adds label to the list
   ~tStackAgent(); // Removes label from the list
};
#define INFORM_KGB(X)     tStackAgent KGB_agent(X/*name*/, 1/*level*/);
#ifdef SAFE_MODE
#define INFORM_KGB_LOW(X) tStackAgent KGB_agent(X/*name*/, 2/*level*/);
#else
#define INFORM_KGB_LOW(X) // disabled
#endif

void Function1_low_level(tIdx<IDX_E> ie){
    {
        INFORM_KGB_LOW("Function1_low_level block1");
        // ... some calculations ...
    }
}

void Function1_high_level(){
   INFORM_KGB("Function1_high_level");
   for(tIdx<IDX_E> ie=e_beg; ie<e_end; ++ie) // Loop over elements
       Function1_low_level(ie);
}
```

Figure 9. Code fragments of the built-in call stack diagnostics

If an assertion fails, the *Crash* function is called with an error message in *printf* format as input. It prints the message to the log file of the parallel process, and to the standard output and error streams. Additionally, it prints information about the actual call stack. If an error occurred at memory allocation, a full memory allocation report for all the arrays is also printed. Then the *Crash* function calls *MPI_Abort* that terminates the group of MPI processes. Note that the *Crash* call is wrapped with the *exit* function in order to inform the compiler about the exit point (for proper code analysis).

Now consider the memory allocation. For performance reasons, dynamic memory allocation is only allowed in upper-level functions. The direct usage of the *new* and *delete[]* operators is forbidden by our coding standard. Memory allocations are wrapped by the template function *GimmeMem* (Fig. 2). It takes the number of elements to be allocated and the text label of this allocation as input. It checks correctness and tries to allocate memory using the *new* operator, which is covered by the try-catch exception handing. If the allocation is successful, the pointer, its size and text label are added to some singleton object that stores the list of allocated pointers. The *Crash* function is called otherwise. This list is used for a full memory allocation report, if needed. It also gives us runtime protection from cyclic memory leaks (by checking the number of allocations). Deallocation is done by the *FreeMem* function, which calls the *delete[]* operator and removes the pointer from the list.

Finally, in order to save memory, the basic containers support the sharing (or aliasing) of data. For instance, if the portrait of a sparse matrix coincides with the mesh topology of some kind, it just shares this data.

6. Performance Evaluation

In terms of performance, we first need to make sure that our containers do not cause overhead in the release configuration compared to plain arrays. This will show the possibility of achieving maximum performance if the mesh data is properly reordered to minimize cache misses (which is beyond the scope of this work). Operations with block vectors with minimal arithmetic intensity were used for tests: assignment, elementwise addition, summation, etc. Different C++ compilers were used (Microsoft, GNU, Intel) on various computing equipment. Results for the release configuration demonstrated identical performance compared to plain arrays in all the tests. This means that compilers correctly substitute inline access functions with corresponding plain pointers.

Secondly, we need to check that the safe mode is not critically slower than the release configuration. The performance of the safe mode may seem unimportant at first glance (and usually nobody pays attention to it). However, when the safe mode is used in a very big simulation for debug purposes, reproducing the problem may consume notable amount of CPU hours just to initialize the simulation and perform several time steps. Therefore, the safe mode must not be by orders of magnitude slower. We tested sequential and parallel executions our supercomputer CFD code [8] on fine and coarse unstructured hybrid meshes (in a typical range from 10^5 to 10^6 cells per CPU socket) using explicit and implicit schemes on various computing equipment, including Lomonosov-2 supercomputer (14-core Intel Xeon Xeon E5 v3). The safe mode configuration appeared to be around 20% slower on average than the release configuration. It can be noted that in the case of an implicit scheme, which is mainly used in simulations, the overhead is smaller, around 10–15%. This is because of the solver for the sparse linear system with the Jacobi matrix, which must be solved at each iteration of the Newton process. The solver oper-

ates only with plain data and consumes nearly half of the overall computing time. Respectively, in case of an explicit scheme the overhead is bigger, around 30%. We can conclude that the observed overhead in the safe mode is insignificant for debug purposes in all the tests.

Finally, the overhead in compilation time has also been measured. It appeared to be around 5%, which is also negligible.

Conclusions

A set of containers with improved access safety has been proposed for storing mesh data in a CFD code. The presented containers for arrays, block arrays, and block arrays with variable block size can be associated with mesh objects of a certain kind. Such containers, which are in fact template wrappers around plain pointers to 1D data vectors, cause no overhead in the release configuration. At the same time, this approach provides throughout access control and extra protection from programming errors. The correctness of the kind of mesh objects is ensured at compilation time, while index range checking works at runtime.

Our QA procedure (about 200 short tests) for the safe mode configuration with low-level data access checks filters out programming errors much more efficiently. Those tricky errors that infiltrate through the QA procedure are captured and localized in the safe mode execution of a supercomputer simulation. To help bugs get caught, we use named arrays to know which data is being accessed incorrectly, a built-in memory allocation monitor to know distribution of memory costs and prevent cyclic memory leaks, an internal call stack tracker that helps to localize problems.

Using the safe mode configuration with improved-reliability containers helped us to notably reduce the costs of warfare against unstable “magic” bugs. It greatly simplified the debugging process and allowed us to improve the quality of our supercomputer simulation software.

Acknowledgements

The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University. The research is funded by the Russian Science Foundation, project 19-11-00299. The authors thankfully acknowledge these institutions.

References

1. Alumbaugh, T.J., Jiao, X.: Compact Array-Based Mesh Data Structures. In: Hanks B.W. (eds) Proceedings of the 14th International Meshing Roundtable. Springer, Berlin, Heidelberg pp. 485–503 (2013), DOI: 10.1007/3-540-29090-7_29

2. Cantwell, C.D., Moxey, D., Comerford, A., Bolis, A., Rocco, G., Mengaldo, G., De Grazia, D., Yakovlev, S., Lombard, J.E., Ekelschot, D., Jordi, B., Xu, H., Mohamied, Y., Eskilsson, C., Nelson, B., Vos, P., Biotto, C., Kirby, R.M., Sherwin, S.J.: Nektar++: An open-source spectral/hp element framework. Computer physics communications 192, 205–219 (2015),

DOI: 10.1016/j.cpc.2015.02.008

3. Cuthill, E., McKee, J.: Reducing the Bandwidth of Sparse Symmetric Matrices. In: Proceedings of the 1969 24th National Conference. pp. 157–172. ACM '69, ACM, New York, NY, USA (1969), DOI: 10.1145/800195.805928

4. Danilov, A.A., Terekhov, K.M., Konshin, I.N., Vassilevski, Y.V.: INMOST Parallel Platform: Framework for Numerical Modeling. Supercomputing Frontiers and Innovations 2(4), 55–66 (2015), DOI: 10.14529/jsfi150404

5. Dyedov, V., Ray, N., Einstein, D., Jiao, X., Tautges, T.J.: AHF: array-based half-facet data structure for mixed-dimensional and non-manifold meshes. Engineering with Computers 31(3), 389–404 (2015), DOI: 10.1007/s00366-014-0378-6

6. Fogerty, S., Martineau, M., Garimella, R., Robey, R.: A comparative study of multi-material data structures for computational physics applications. Computers & Mathematics with Applications 78(2), 565–581 (2019), DOI: 10.1016/j.camwa.2018.06.010

7. Garimella, R., Perkins, W., Buksas, M., Berndt, M., Lipnikov, K., Coon, E., Moulton, J., Painter, S.: Mesh infrastructure for coupled multiprocess geophysical simulations. Procedia Engineering 82 (12 2014), DOI: 10.1016/j.proeng.2014.10.371

8. Gorobets, A.: Parallel Algorithm of the NOISEtte Code for CFD and CAA Simulations. Lobachevskii Journal of Mathematics 39(4), 524–532 (2018), DOI: 10.1134/S1995080218040078

9. Gorobets, A., Soukov, S., Bogdanov, P.: Multilevel parallelization for simulating turbulent flows on most kinds of hybrid supercomputers. Computers and Fluids 173, 171–177 (2018), DOI: 10.1016/j.compfluid.2018.03.011

10. Tautges, T.J.: MOAB-SD: Integrated structured and unstructured mesh representation. Engineering With Computers 20(3), 286–293 (2004), DOI: 10.1007/s00366-004-0296-0

11. Vazquez, M., Houzeaux, G., Koric, S., Artigues, A., Aguado-Sierra, J., Aris, R., Mira, D., Calmet, H., Cucchietti, F., Owen, H., Taha, A., Burness, E.D., Cela, J.M., Valero, M.: Alya: Multiphysics engineering simulation toward exascale. Journal of Computational Science 14, 15–27 (2016), DOI: 10.1016/j.jocs.2015.12.007

12. Weinbub, J., Rupp, K., Selberherr, S.: A Flexible Dynamic Data Structure for Scientific Computing. IAENG Transactions on Engineering Technologies. Lecture Notes in Electrical Engineering 229, 565–577 (2013), DOI: 10.1007/978-94-007-6190-2_43

DOI: 10.14529/jsfi190404

Survey on Software Tools that Implement Deep Learning Algorithms on Intel/x86 and IBM/Power8/Power9 Platforms

Denis Shaikhislamov[1], *Andrey Sozykin*[2,3], *Vadim Voevodin*[1]

Neural networks are becoming more and more popular in scientific field and in the industry. It is mostly because new solutions using neural networks show state-of-the-art results in the domains previously occupied by traditional methods, eg. computer vision, speech recognition etc. But to get these results neural networks become progressively more complex, thus needing a lot more training. The training of neural networks today can take weeks. This problems can be solved by parallelization of the neural networks training and using modern clusters and supercomputers, which can significantly reduce the learning time. Today, a faster training for data scientist is essential, because it allows to get the results faster to make the next decision.

In this paper we provide an overview of distributed learning provided by the popular modern deep learning frameworks, both in terms of provided functionality and performance. We consider multiple hardware choices: training on multiple GPUs and multiple computing nodes.

Keywords: HPC, neural networks, deep learning frameworks, distributed training.

Introduction

Neural networks are currently one of the most popular methods for creating AI systems. They show state-of-the-art results in many areas, including computer vision [55, 59, 62, 69], natural language processing [53, 58], speech recognition [56, 71]. However, to achieve such results, neural networks are becoming more and more complex and more and more data is needed for their training [53, 60, 64]. As a result, the training of such neural networks requires considerable time: days, weeks, and even months.

Problems with training neural networks can be solved using modern clusters and supercomputers. In this case, the neural network is trained in parallel on several computing nodes of the cluster, which can significantly reduce the learning time [54]. In addition, you can train a large network on a parallel computing system that does not fit in the memory of one computer [52, 60]. As a result, distributed training of neural networks is rapidly gaining popularity.

In this paper, we provide an overview of distributed training of neural networks that exist in modern deep learning frameworks. The usage of various hardware options for distributed training are considered: distributed training on several GPUs and several computing nodes. It also describes approaches to the logic of organizing distributed learning. The most popular approaches of distributed training are model and data parallelism. In data parallelism, each device contains a complete copy of the neural network and performs training on parts of the data. In the case of model parallelism, the neural network is divided into separate parts, which are distributed among devices and are trained on a complete data set.

There are two types of distributed learning organization: asynchronous and synchronous. In asynchronous training, parallelization occurs due to the breakdown of work between workers and parameter servers. Workers are used for training (independently of each other), and parameter servers only store model parameters and their modification. The synchronous approach is

[1]Research Computing Center of Lomonosov Moscow State University, Moscow, Russian Federation
[2]Ural Federal University, Ekaterinburg, Russian Federation
[3]N.N. Krasovskii Institute of Mathematics and Mechanics, Ekaterinburg, Russian Federation

organized as follows. Workers have their own copy of the model, but the data is different. After the workers have processed their part of the data, they exchange results with each other and at the same time change the parameters of the model. You can combine these two methods: use synchronous learning within a node with several GPUs and use asynchronous learning between nodes.

Currently there is a large number of frameworks for training neural networks, the most popular of which are TensorFlow, Caffe, Caffe2, Torch, PyTorch, MXNet, Theano, PaddlePaddle, Microsoft Cognitive Toolkit, Deeplearning4j, Keras and OpenCV. All these frameworks can use parallel training on several computing cores and processors, conduct training on the GPU, but the possibilities of using distributed training are much more limited. In this review, we included libraries that can only parallelize the training of a neural network on several GPUs and computing nodes. Therefore, it was necessary to exclude such popular frameworks as Theano and OpenCV, in which distributed learning tools are not developed.

This paper discusses the possibility of using parallel computing systems only for training deep neural networks. The inference requires other optimizations, including the features of parallel execution and use of GPUs on mobile systems [70]. Supercomputers are rarely used for inference.

The rest of the paper is organized as follows. Section 1 describes the most common frameworks. The following items are highlighted for each framework:

- A brief description of the development history and current status.
- Implementation features and a list of supported algorithms – we give description of the basic principles of the framework. The support for the implementation of the main types of neural networks (fully connected, convolutional (CNN), recurrent (RNN), deep autoencoders (DAE), generative adversarial network (GAN) and networks with Transformer architecture) is indicated for each framework.
- Optimizations for Intel, Power, Nvidia platforms – describes the availability of optimizations, as well as their application in terms of availability.
- Use of several GPUs for training – the possibility of using several GPUs for training both with the standard package and with the help of add-ons.
- Support for training on multiple nodes – the ability to start training a neural network on multiple machines (for example, on different nodes of a cluster). Both standard support and the use of add-ons are considered.

Section 2 shows the comparison of frameworks both in terms of functionality and performance on the base models that are used by the community to evaluate the performance of various frameworks. It is worth noting that in Section 1 the questions of the frameworks performance are not addressed.

1. Deep Learning Frameworks Overview

1.1. TensorFlow

1.1.1. Overview

TensorFlow (hereinafter, TF) [47] is a relatively young framework for high-performance computing, which is mainly used as a framework for implementing deep learning methods and is developed by the Google Brains team. It was developed in a closed manner until 2015, but

after its revamp it was released to the public. The TF core is mostly written in a combination of C++ and CUDA, but there are also parts written in Python. TF has an API for both Python and C++, but since TF is mainly used in Python, this review will look at the Python API.

TF is in an active development. Current version is TensorFlow 2.0, which focuses on ease of use, including the ease of distributed learning. Detailed development plans can be seen in [45].

1.1.2. Implementation features and a list of supported algorithms

TF relies on the concept of a computation graph that describes data and operations on it. Because of this mechanism, TF can be used for almost any calculation. Examples of simple graphs are shown in Fig. 1.

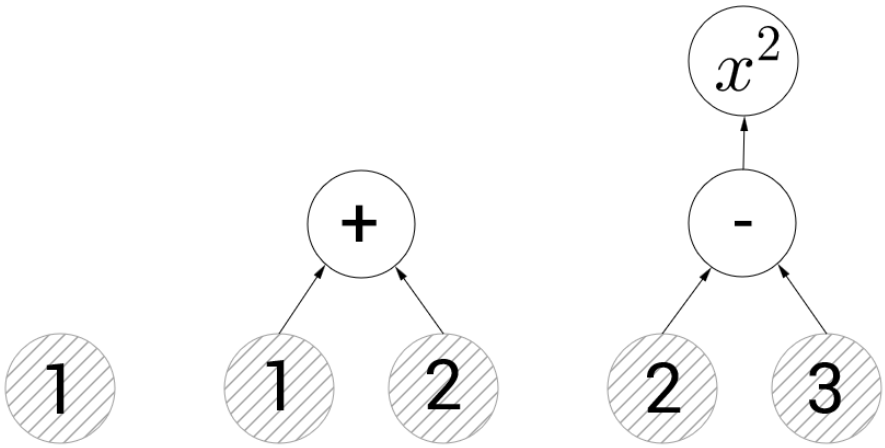


Figure 1. Examples of simple computation grahs

The computation graph can be created by the user himsel. Also, the graph is created by default when the framework is used, which will be used if you do not specify which graph the operation is performed with. TF implements a lot of algorithms for both deep learning and traditional machine learning.

Inside the graph, the data is represented by the so-called tensors (n-dimensional matrices). The size of the matrix can be specified on its definition, or it may not be specified if the size of the matrix changes dynamically during the execution of the calculation graph.

The graph is calculated within the session, to where all the data necessary for calculation is passed, after which the graph is calculated, and the result of the execution is returned. In version TF 2.0, Eager execution has appeared, which makes it possible to start calculations without creating a session.

In TensorFlow you can implement algorithms that efficiently utilize tensors (for example, there are ray tracing implementations on TF). But since all the necessary algorithms for deep learning are provided inside the framework, it is most often used for deep learning. In TF most of the optimizers (gradient descent, Adagrad, AdaDelta, momentum, Adam, etc.) and activation functions (relu, relu6, elu, sigmoid, tanh, softplus, softsign, dropout, softmax etc.) are implemented.

In TensorFlow you can implement all the main types of neural networks: fully connected, convolutional, recurrent, deep autoencoders, GAN and Transformer.

1.1.3. Intel, Power, Nvidia platform optimizations

TF optimization for Intel are mentioned even on the official TF website. For example, TF will work optimally if it is built from the source files provided on the official TF website, which will incorporate all the features of Intel processors. Also, Intel has MKL-DNN, which is supported by TF (TF with Intel MKL-DNN).

TF is available in the PowerAI framework package, which includes Power-optimized versions of the popular frameworks and Distributed Deep Learning (DDL), which is a communication library optimized for distributed training of neural networks. It is worth noting that PowerAI supports only GPU systems.

It is also worth noting that the versions of Intel and PowerAI do not differ in terms of implemented algorithms. This is also true for subsequent frameworks, unless otherwise is specified.

TF has an optimized version for Nvidia GPUs, which can be downloaded from the official website. This version uses the CUDA library, which is used to run general-purpose computing on the GPU, and cuDNN, containing effective operations for working with neural networks on the GPU. In addition, NVidia has a special platform called the Nvidia GPU Cloud, which has a container docker directory with preinstalled and optimized NVidia software, including TensorFlow.

1.1.4. Multi-GPU and Multi-node training of neural networks

The TF framework was developed with the goal of providing distributed training in large clusters, so it includes training tools on several GPUs and computing nodes.

Data-parallel training in TF is implemented in the Distribution Strategy API (tf.distribute.Strategy), which includes several strategies for distributed training [14]. The most popular approach is synchronized distributed training on several GPUs, both on one node (MirroredStrategy) and on several nodes, each of which can have several GPUs installed (MultiWorkerMirroredStrategy). It is also possible to use asynchronous training using the ParameterServerStrategy strategy. The strategy allows the use of several nodes as parameter servers and workers, and nodes with workers can contain several GPUs. When using the Distribution Strategy in conjunction with the high-level TensorFlow APIs (Keras and Estimators), parallelization of training is performed automatically, both on several GPUs within the same node, and between cluster nodes. Developing your own layers or training methods requires you to explicitly use the parallel programming.

Model-parallel training in TF is possible using the Mesh TensorFlow library [66]. This library is an add-on for TF and defines the language of distributed processing of tensors. The library allows you to explicitly indicate by which dimension the tensors will be divided for distributed processing. As a result, it is possible to parallelize operations both by model and by data, which provides data-parallel or model-parallel training, as well as its combination. Training is performed in synchronous mode; all synchronization operations are provided automatically by the Mesh TensorFlow library.

There is a separate library from TF, Horovod [65], which runs on top of TF and PyTorch and provides a simpler interface for implementing distributed synchronous training on clusters. The library uses MPI for communication between processes, and it is also possible to use several GPUs on a node. More details about the use of Horovod can be found in [20]. It is worth noting that Horovod can be used with the DDL communication library from IBM; details are presented in [13].

1.2. Caffe and Caffe2

1.2.1. Overview

Caffe (Convolutional Architecture for Fast Feature Embedding) [57] is a deep learning framework developed by the University of California, Berkeley. The main emphasis in this framework was made on the task of image classification and segmentation. Caffe is distributed under the BSD license. It is believed that Caffe is one of the very first successful deep learning frameworks, considering many research papers that have used this framework. Caffe has a library of pre-built and pre-trained models of neural networks that have been successfully used in a particular subject area. This resource is called Model Zoo and it is considered a big advantage of Caffe, since there are many useful models there, which is why other frameworks usually implement Caffe model converters into their own.

The framework is written in C++ with an interface for Python.

Caffe is no longer supported, since the development team has switched its interest towards the development of Caffe2 [5]. At the end of March 2018, the PyTorch and Caffe2 teams merged, after which Caffe2 moved to the PyTorch repository and became a part of it. The latest version of Caffe, 1.0, was released on April 18, 2017.

1.2.2. Implementation features and a list of supported algorithms

In Caffe, everything is built on layers. Layer is a description of a data processing operation. It can implement both a neural network layer and pooling (non-linear compression of a feature map, in which a group of pixels is compressed to 1 pixel using a non-linear transformation, such as a maximum function), filters, non-linear transformations, an error function, etc. The neural network (Net) itself will consist of many layers, which must be described in the configuration file in the Protobuf format.

The layers also have two functions: forward and backward. In the forward function, the error function is calculated; in the backward function the gradients are calculated to further change the model parameters. Inside the layer, two versions of each of these functions are implemented, one for the CPU, the second for the GPU. Depending on the mode (CPU or GPU), which is specified at the beginning of the program, the corresponding method will be called (CPU is used by default). In the neural network itself, the functions of the forward and backward passage will invoke the corresponding functions of the layers included in the neural network.

The final element is the optimizer (Solver), which implements the optimizer of your choice. Supported: SGD, AdaDelta, AdaGrad, Adam, Nesterovs Accelerated Gradient and RMSProp.

Initially, Caffe implemented algorithms only for machine vision (based on convolutional neural networks), but later added support for recurrent neural networks in the form of LSTM. Now it is possible to implement fully connected, recurrent, convolutional networks, deep autoencoders and GAN. Transformer network implementation was not found.

1.2.3. Intel, Power, Nvidia platform optimizations

Intel provides Intel Distribution of Caffe – a version optimized for Intel processors [24]. This version of Caffe uses Intel MKL-DNN, which, in its turn, allows you to use OpenMP by default.

Caffe2 worked together with Intel to integrate Intel MKL functions to optimize the performance of the framework when used in production, but at the moment Caffe2 does not have full

MKL support. MKL still can be used with Caffe2, but some users had problems with neural network training using Caffe2 with Intel MKL.

Caffe is included in PowerAI. There are two versions of Caffe in PowerAI: Caffe BVLC – a standard version of Caffe developed in BVLC (Berkeley Vision and Learning Center); Caffe IBM is an IBM-optimized version of the framework. The default version is Caffe IBM. Caffe2 is included in PyTorch, which in turn is included in PowerAI.

Caffe and Caffe2 support CUDA and CuDNN, and also have containers in the Nvidia Cloud.

1.2.4. Multi-GPU training of neural networks

To use several GPUs for training in Caffe, you must use your own Caffe scripts; a complete guide is provided in [4]. Synchronized distributed training is used. Asynchronous distributed training, as well as model parallelism, are not supported.

You can use a special version of Caffe, NVCaffe, which is supported by NVidia. This version was created specifically for the use of several GPUs. User instructions can be found in [35].

Caffe2 has a special distributed training module that implements various algorithms (for example, SynchronousSGD) of synchronous distributed training models. Caffe2 uses NCCL for synchronization, a communication library for multi-GPU nodes, which provides very good scalability. An example of use is given on the official website for training the ResNet-50 neural network both at several GPUs and at several nodes in [6]. A more general instruction is provided in [7]. Model parallelism in Caffe2 is possible, but manual distribution of neural network layers across devices is required before training the model.

Caffe2 has no official support for asynchronous training.

1.2.5. Multi-node training of neural networks

When using Intel Distribution of Caffe, [19] provides guide on setting up and starting the training process. Multi-node mode uses synchronized distributed training. For synchronization after processing their data, the nodes use the Intel Machine Learning Scaling Library (MLSL) for communication, which implements communication primitives for both model and data parallelism. But for multi-node training you need to configure the nodes separately, therefore the problem: the user may not know which nodes his application will work on.

For HPC clusters with GPU nodes, Inspur developers developed a version of Caffe – Caffe MPI [3] – which utilizes MPI to synchronize results between nodes after processing its part of the data.

In Caffe2, similar to multi-GPU training, you can also run model training on multiple nodes. You can select several communication backends to synchronize the results: Gloo and Redis. It is also worth noting that it is necessary for all nodes to have a common shared folder through which, after starting the program, the nodes can detect each other and start the training. More detailed instructions for using this mode are given in the ResNet-50 training example in [6].

1.3. Torch

1.3.1. Overview

Torch [51] – MATLAB-like framework for the Lua programming language, which provides a huge set of algorithms for deep learning. The core of the framework is written in C, but the

programs themselves are written in Lua. LuaJIT is used for JIT compilation, which significantly speeds up the program. Framework is distributed under the BSD license. Despite the fact that Torch is a very powerful and convenient framework, programs for the framework must be written in Lua. As noted in [31], Lua is not a popular programming language in the field of data science. Users work around this problem by using the PyTorch package, which has an interface for Python and not only includes all the functionality of Torch, but also complements it.

Torch is not in active development, as the team switched to PyTorch development. The latest version, torch7, was released on February 27, 2017.

1.3.2. Implementation features and a list of supported algorithms

The core of the framework is in the torch package, in which there is a tensor implementation – similar to TensorFlow, n-dimensional arrays, basic indexing operations, getting array slices, transposing, etc., as well as BLAS operations, implementation of mathematical functions (max, min, etc.) and statistical distributions.

The next important package is nn, which is used to build neural networks. The package has many different parts, but every one of them has a common part called "Module", which implements the forward and backward functions, that allow you to make forward and backward passes through the network. Modules can be connected using classes such as Sequential, Parallel, Concat, which allows you to build complex models of neural networks. There is also a list of basic modules, such as Tanh, Linear, Max, etc.

Error functions are implemented as subclasses of the Criterion class, which is similar to the Module class, since it has the same forward and backward functions. Torch include implementations of the basic error functions, such as cross-entropy, mean squared error, and stochastic gradient descent.

Torch allows you to implement all the most popular types of neural networks: fully connected, convolutional, recurrent, deep autoencoders, GAN and Transformer.

1.3.3. Intel, Power, Nvidia platform optimizations

Torch has a separate development branch that has Intel optimizations, especially for Intel Xeon. You can use Intel MKL in conjunction with Torch. The repository branch is located in [25].

Torch is included in the PowerAI framework.

Optimizations for the NVIDIA platform are included in cutorch – the CUDA backend for Torch. Torch supports CUDA and cuDNN. Nvidia Cloud does not contain a container with an optimized version of Torch.

1.3.4. Multi-GPU training of neural networks

Support for multiple GPUs is included in the standard version of Torch. To utilize multi-GPU training, you need to use the module in nn – DataParallelTable – which allows you to use data parallelism. Torch supports synchronous distributed training. It should be noted that parallelization is done quite simply: you just need to wrap the Module that implements the neural network in a DataParallelTable with the list of GPUs on which the training will be conducted. A more detailed guide can be found in [9].

There is no official support for asynchronous training and model parallelism with Torch.

1.3.5. Multi-node training of neural networks

Torch does not have official support for distributed training on multiple nodes, but there is a separate development branch, Torch MPI, which allows you to use nodes not only exclusively with the CPU, but also with the GPU. More detailed information can be found in [46].

1.4. PyTorch

1.4.1. Overview

PyTorch [42] is a deep learning framework that was based on Torch and is developed primarily by the Facebook's artificial intelligence research group. PyTorch also includes the Caffe2 framework, so PyTorch has advanced distributed training capabilities.

Pytorch is currently under active development. The stable version, 1.3.0, was released on October 10, 2019.

1.4.2. Implementation features and a list of supported algorithms

Due to the fact that PyTorch is based on Torch, they are very similar. The main module, torch, includes tensors and operations (transformations, mathematical functions, etc.), serialization operations.

The next important module that is used to build neural networks: torch.nn. This module includes a description of the base class Module, which is inherited by all types of neural network layers implemented in PyTorch (convolutional, pooling, linear, recurrent, etc.). The module has implementations of a large number of activation functions (for example, ReLU6, sigmoid, softsign, tanh, etc.), error functions (MSE, L1Loss, CrossEntropy, BCELoss, etc.). Each module has, similar to Torch, forward and backward functions. The neural network itself is a Module object, that is a combination of other Module, which allows to automatically make forward and backward passes through the resulting network.

With PyTorch, you can implement almost all the types of architecture used in neural networks: fully connected, convolutional, recurrent, deep autoencoders, GAN, and Transformer.

1.4.3. Intel, Power, Nvidia platform optimizations

Intel optimizations mainly consists of integrating Intel MKL-DNN into PyTorch [22]. Prior to version 1.0, the PyTorch distribution did not utilize Intel MKL-DNN, but now PyTorch supports it by default.

IBM has included PyTorch in PowerAI, which uses OpenBLAS.

For NVidia, PyTorch is supported on Nvidia Cloud. It is also worth noting that there was a collaboration between PyTorch and Nvidia, which resulted in Apex – A PyTorch Extension – a set of utilities that make it easy to use distributed training technologies and Automatic Mixed Precision. PyTorch supports CUDA and CuDNN.

1.4.4. Multi-GPU training of neural networks

Just like Torch, PyTorch has a DataParallel module in the nn package, and if you wrap a neural network model in this module, then the training will be done on several GPUs. It uses data parallelism and synchronous distributed training. Model parallelism is also possible: when

defining a model, it is necessary to distribute the layers across different devices and implement its forward and backward pass function, including data transfers between GPUs [43].

PyTorch has no official support for asynchronous training.

1.4.5. Multi-node training of neural networks

PyTorch has a DistributedDataParallel module that uses the torch.distributed package to parallelize training across multiple processes/nodes. This module can be used in two modes: 1 process with several GPUs and several processes with 1 GPU. Developers do not recommend using the use case with several GPUs, noting that the framework will work faster if you create 1 process for each GPU. The module supports 2 backends for communication: gloo and nccl. Developers recommend using the nccl backend, as it showed better performance in their experiments.

In version 1.0, developers changed the core of the torch.distributed module, and now it depends on the C10D library, which works asynchronously for all supported backends. It sped up all of its dependent modules (DataParallel, DistributedDataParallel).

It is also worth noting that Horovod supports PyTorch. Instructions can be found in [20].

1.5. Apache MXNet

1.5.1. Overview

MXNet [1] is a popular deep learning framework known for its flexibility and ability to scale across multiple GPUs and nodes. It is developed by a team from the Apache Software Foundation and distributed under the Apache 2.0 license. The core is written mainly in C++; there are interfaces for a large number of programming languages: C++, Python, R, MatLab, JavaScript, Go, Scala, Perl, Julia, Wolfram Language. Due to its scalability, Amazon has chosen MXNet for its AWS cloud environment. Like Caffe, MXNet has its own library of pre-trained models - Gluon model zoo.

MXNet is currently under active development. The latest version, 1.5.0, was released on June 8, 2019.

1.5.2. Implementation features and a list of supported algorithms

MXNet is similar to TensorFlow: it operates on NDArray (similar to a tensor, it is an n-dimensional array) and a computation graph. Graphs include variables - objects whose type and size are not determined during its initialization, but will be calculated as the data is fed into the graph.

A neural network is built using the Module API, which implements almost all common types of layers of neural networks, activation functions and optimizers.

MXNet has a higher level and simpler interface for creating neural networks - Gluon. Gluon is very similar to Keras API in its simplicity of neural network creation. An initial guide to using Gluon can be found in [2].

MXNet supports most popular neural network architectures such as fully connected, convolutional, recurrent, deep autoencoder, GAN, and Transformer (using the GluonNLP extension).

1.5.3. Intel, Power, Nvidia platform optimizations

Intel and Apache MXNet released version 1.2.0, in which the main point was to optimize MXNet for CPUs using Intel MKL-DNN, which significantly accelerated the work of the framework. Detailed information about both the installation and how MXNet has accelerated can be found in [17].

Authors could not find the information about optimizing the framework for the Power architecture. MXNet is not part of PowerAI but installing MXNet is still possible.

For NVidia, MXNet is supported by Nvidia Cloud. MXNet also has support for CUDA and CuDNN.

1.5.4. Multi-GPU training of neural networks

By default, data parallelism is used in MXNet, but model parallelism is also supported. [34] provides an example of using several GPUs for parallelizing a LSTM model.

Data parallelism is quite simple. When initializing the module, it is necessary to give a list of GPUs on which training will be conducted. There is also built-in support for static load balancing: if one GPU is faster than another, then you can set the proportions in which the GPUs will process the data. Detailed information on GPU parallelization can be found in [32, 33].

1.5.5. Multi-node training of neural networks

MXNet officially supports both asynchronous and synchronous distributed training. For distributed training, MXNet uses three kinds of processes. The first of these is a worker in which training will occur, and which can use multi-GPU training. The second process is called the server (similar to the parameter server in Tensorflow), which stores the model parameters. There can be several servers, and they can be located both on the machine with the worker or on another machine. Servers store parameters in a key-value format. The third type of process is the scheduler, which initializes the cluster and is responsible for ensuring that other processes can interact with each other.

The training depends on which mode the server is created with. There are 4 modes: dist_sync – for synchronous training; dist_async – for asynchronous training; dist_sync_device – similar to dist_sync, but is used for training on several GPUs, which allows to skip time-consuming communications between the CPU and GPU and synchronize the results only between GPUs (this method uses more memory on the GPU); dist_async_device – similar to dist_async, but is used for training on several GPUs.

If the communication becomes a bottleneck, you can use compression of the gradients to reduce the load on the communication.

It is also worth noting that MXNet added integration with Horovod for distributed training in 1.4.0 version.

1.6. PaddlePaddle

1.6.1. Overview

PaddlePaddle [36] is a deep learning framework introduced by Chinese search giant Baidu in 2016. The main highlighted features are simplicity, scalability and flexibility. It is written in C++, but it also has interfaces for C++, Python.

PaddlePaddle is currently under development. The latest version, PaddlePaddle 1.5.1, was released on July 16, 2019.

1.6.2. Implementation features and a list of supported algorithms

The main concept in PaddlePaddle, like TensorFlow, is a computation graph, but the operation is different: in PaddlePaddle a Python program that describes a neural network model builds a computation graph in protobuf format [41], and then sends it for execution to the so-called Executor: either process responsible for distributed training, or to the libpaddle.so library for local execution. A description of how the architecture works and what were the reasons for such a design can be found in [45].

PaddlePaddle supports most of the neural network architectures used: convolutional, recurrent, fully connected, deep autoencoders, GAN, and Transformer.

1.6.3. Intel, Power, Nvidia platform optimizations

PaddlePaddle, like other frameworks, can use Intel MKL to speed up the framework on the CPU and Intel MKL-DNN to speed up the convolutional neural networks.

Information about optimizing the library for the Power IBM architecture could not be found.

PaddlePaddle is supported on Nvidia Cloud and can also utilize CUDA and CuDNN.

1.6.4. Multi-GPU training of neural networks

PaddlePaddle has official support for multi-GPU training and has two ways to use it. The first way is that ParallelExecutor is used instead of the usual Executor. PaddlePaddle uses synchronous distributed training. Information on the implementation of asynchronous training in official sources could not be found. More detailed instructions can be found in [39]. Developers state that current version of Fluid provides only data parallelism training mode [38].

The second way is to compile the computation graph using CompiledProgram, which transforms the graph for faster execution. Then you need to call with_data_parallel, which transforms the graph so that several devices (in CPU mode, threads) can be used. PaddlePaddle automatically detects all available GPU devices and distributes the work between them. Developers recommend using this method. An example of usage can be found in [40].

1.6.5. Multi-node training of neural networks

PaddlePaddle has official multi-node support, and there are two possible uses. In case of the RPC communication backend, training is based on the Trainer-ParameterServer architecture. Both synchronous and asynchronous distributed training are supported. Trainer is engaged in the training, and the ParameterServer is responsible for storing and modifying the model parameters. The peculiarity is that it is necessary to start the processes of parameter servers and the training processes separately, in which it is necessary to specify the address and port of the parameter server. DistributeTranspiler is used to distribute training, which, depending on how many trainers and parameter servers, can give individual processes the computation graph that they need. The output computation graphs also contain all the routines of synchronization and parameter updates.

In case of NCCL communication backend, the parameter server is no longer needed, since the trainers themselves are communicating and updating model parameters. All the examples of multi-node training can be found in [38].

1.7. Microsoft Cognitive Toolkit

1.7.1. Overview

Microsoft Cognitive Toolkit [29] (hereinafter, CNTK) is a deep learning framework developed by Microsoft Research. It is written in C++ and has interfaces in C++, Python and BrainScript.

The developers stopped developing this framework after version 2.7.0, released on March 29, 2019.

1.7.2. Implementation features and a list of supported algorithms

CNTK is similar to the Keras API in building neural networks. CNTK provides basic modules that implement the most used units in neural networks.

The cntk.layers module includes various types of layers of neural networks: recurrent, Embedding, Dense, etc. Basic models, which consist of a sequence of layers of different or identical types, can be built using Sequential. The cntk.learners module provides set of the most popular optimizers that are used in practice, such as SGD, Adam, Nesterov, RMSProp, etc. The cntk.losses module implements error functions like binary cross-entropy, squared error, etc.

After constructing the neural network model from the provided blocks, it is necessary to create a Trainer object, which receives the model at the input and the selected optimizer for training the model. Then you can start training the model using train_minibatch.

CNTK supports such types of neural network architectures as fully connected, convolutional, recurrent, deep autoencoders, GAN and Transformer.

1.7.3. Intel, Power, Nvidia platform optimizations

Microsoft CNTK has two versions: CPU-only, which uses Intel MKL-DNN by default, and a version with a GPU that uses CUDA (CUB and cuDNN).

The only mention authors found about CNTK for Power platform is in Keras docs: Keras supports the CNTK backend.

CNTK is supported in the Nvidia Cloud.

1.7.4. Multi-GPU training of neural networks

CNTK has distributed training support. Synchronous (DataParallelSGD, BlockMomentumSGD, ModelAveragingSGD) and asynchronous (DataParallelASGD) optimizers can be used. The prerequisite for distributed training is that you need to install the MPI for communication between processes. CNTK does not support other communication backends (for example, you can use Gloo with Caffe2). It is worth noting that multi-GPU training on a node occurs through MPI, where each process uses 1 GPU, and processes are communicating through MPI.

Parallelization is quite simple – after the definition of the selected optimizer, you need to wrap it in data_parallel_distributed_learner. A detailed guide to using the distributed CNTK package can be found in [30].

CNTK does not support model parallelism.

1.7.5. Multi-node training of neural networks

Multi-node training is completely same as the multi-GPU training, since MPI is used for communication between processes for several GPUs as well as for several nodes.

1.8. Deeplearning4J

1.8.1. Overview

Deeplearning4j [10] (hereinafter, DL4J) is a deep learning framework written in Java. It has interfaces for Java, Scala, Python (via Keras), Clojure. It is the first large-scale deep learning library for Java, and provides a convenient and scalable interface for distributed training when used with the default integration with Hadoop and Spark. DL4J needs an additional ND4J library if the GPU computing is needed that implements CUDA. DL4J is distributed under the Apache 2.0 license.

The development of the framework is quite slow. The latest stable version, 0.9.1, was released on August 12, 2017, and there is also a beta version 1.0.0-BETA4.

1.8.2. Implementation features and a list of supported algorithms

There are two ways to work with the framework. The first way most people deal with when they start working with this framework is the usage of MultiLayerNetwork. This is a high-level API for building neural networks consisting of a sequence of layers of a certain type. The syntax for this API is very similar to the Keras API.

The second method is the manual construction of a computation graph that describes the architecture of the neural network. It is worth noting that everything that can be done through MultiLayerNetwork can also be done by constructing a graph of calculations, but the configuration of this graph will be much more complicated. However, this approach allows you to implement any desired network architecture.

For data processing, the DataVec module is included in the framework, which implements almost all the necessary functions for loading, saving and converting data, and developers recommend its use wherever possible.

DL4J uses an additional ND4J library to work with tensors, which provides the ability to work with n-dimensional arrays, as well as the ability to use not only CPUs for processing, but also GPUs.

The framework supports most of the popular types of neural network architectures: convolutional, recurrent, fully connected, deep autoencoders. In DL4J there is no way to create a GAN by your own means, but you can import the GAN model described in Keras. Transformer implementation in DL4J could not be found.

1.8.3. Intel, Power, Nvidia platform optimizations

Like other frameworks, DL4J can utilize Intel MKL BLAS.

DL4J was optimized for Power platforms, the process of which is described in [11], but the information is outdated, since the page to which the link had been provided in the article to the optimized version of the library no longer exists.

NVidia GPUs can be used in DL4J via ND4J library, which supports the CUDA and cuDNN libraries. DL4J is not supported in Nvidia Cloud.

1.8.4. Multi-GPU training of neural networks

DL4J supports multi-GPU training, and the parallelization process is quite simple and transparent for the developer: after defining the model using, for example, MultiLayerNetwork, you need to pass it to the ParallelWrapper and start the training process. This wrapper implements synchronous distributed training; synchronization of model parameters are implemented in the wrapper. A detailed guide can be found in [16]. It is claimed that DL4J supports model parallelism, but the authors could not find any guides.

1.8.5. Multi-node training of neural networks

Multi-node training, similarly to PaddlePaddle and MXNet, takes place according to the parameter server - worker architecture, in which the worker processes their part of the data, sends the results to the parameter server, which, after modifying the parameters, sends all updated model parameters to everyone. All examples of usage provided on the official website are designed to work on Spark clusters [12, 68].

1.9. Keras

1.9.1. Overview

Keras [50] is an open library used to work with neural networks that was written in Python. The library was developed as part of the research efforts of the ONEIROS project and is an add-on to other frameworks (front-end) for deep learning. The basic principle followed by the developers is to make the interface between the developer and the backend as intuitive and convenient as possible for quick development. Keras supports the following frameworks as a backend: TensorFlow, Theano, CNTK, MXNet.

Keras is under active development. The latest version, 2.2.4, was released on October 3, 2018. In addition, Keras has been included with TensorFlow and has been recommended to use as a high-level API since TF 2.0. The TensorFlow version of Keras includes a large number of optimizations for TensorFlow that are not found in the standalone version of Keras.

1.9.2. Implementation features and a list of supported algorithms

There are two ways to build neural network models in Keras - using the Sequential class or functional API.

Sequential is a tool for building neural networks in which layers go sequentially one after another. Keras implements a large number of layers of neural networks, such as LSTM, layers for convolutional neural networks (Conv1D, Conv2D, Conv3D), etc. After building the model, you need to compile it: call the compile method with the specified optimizer and error function. Keras provides an extensive set of implemented optimizers, such as SGD, Adam, Nadam, Adagrad, RMSProp, etc.

Using the functional API makes it possible to implement more complex types of neural networks in which layers can connect to each other arbitrarily, including several layers that work in parallel. One example of using the functional API is to build a neural network with multiple inputs. This neural network receives part of the input data at the input of the first layer, and the next part of the input data is supplied to the neural network only after merging

with the output of one of the internal hidden layers. A more detailed guide to the functional API and examples of its use can be found in [18].

Keras supports most of the popular neural network architectures: convolutional, recursive, fully connected, deep autoencoders, GAN and Transformer.

1.9.3. Intel, Power, Nvidia platform optimizations

Due to the fact that Keras is the interface between the developer and the backend, all the optimizations applied to the framework chosen as the backend are also applicable to Keras.

At Nvidia Cloud Keras is available as part of the container with TensorFlow.

1.9.4. Multi-GPU training of neural networks

Keras officially supports multi-GPU training, but only with the TensorFlow backend using the Distribution Strategy API.

It is also possible to use distributed training with the MXNet backend. To use the MXNet backend, you must use the Keras version supported by MXNet developers [26]. When compiling the model, it is necessary to pass context parameter to the input of the method, which contains a list of GPUs on which training can be conducted. A more detailed usage guide is provided in [44].

There is no support for model parallelism in Keras.

1.9.5. Multi-node training of neural networks

In Keras with the TensorFlow backend, multi-node training support is available in an experimental mode with the MultiWorkerMirroredStrategy strategy. Alternatively, you can use the Horovod framework, which has good support for Keras models.

2. Frameworks Comparison

2.1. Basic Comparison

Table 1 provides a comparison of frameworks by common parameters. The designations used in the application: sync / async – support for synchronous / asynchronous distributed training, "?" in the column about the availability of optimizations for the platform – the authors could not find information on the highlighted items. A + in the column about the maximum number of GPU / nodes on which training was started means that there is support for the specified mode, but the authors could not find quantitative results.

In terms of the supported types of neural networks, the frameworks are very close. Almost all frameworks, excluding only Caffe and DeepLearning4J, support the popular neural network architectures: fully connected, recurrent, deep autoencoders, GAN, and Transformer. Caffe and DeepLearning4J do not support architectures that have become popular relatively recently: GAN and Transformer.

Table 1. Basic comparison of frameworks

Title	Written in	Available interfaces	Supported types of neural networks					Supported modes of distributed training	Distributed training models	The maximum number of GPU/nodes on which training was run	Optimizations for		
			CNN	RNN	DAE	DAE	Transformer				Intel	Power	NVidia CUDA/ Nvidia GPU Cloud
TensorFlow	C++, CUDA, Python	Python, C/C++, Java, Go, JS, R, Julia, Swift	+	+	+	+	+	Sync/ acync	Data parallel, model parallel	8 / 64[4]	+ (MKL BLAS + DNN)	+	+/+
Caffe	C++	C++, Python	+	+	+	+	-	sync	Data parallel	256/64[5]	+ (MKL BLAS + DNN)	+	+/+
Caffe2	C++	C++, Python	+	+	+	+	-	sync	Data parallel, model parallel	+/+	+ (MKL BLAS)	+	+/+
Torch	C, Lua	Lua, C	+	+	+	+	+	Sync/ async	Data parallel	256/64[6]	+ (MKL BLAS + Xeon optimizations)	+	+/-
PyTorch	Python, C, Lua	Python	+	+	+	+	+	sync	Data parallel, model parallel	8/8	+ (MKL-DNN)	+	+/+
MXNet	C++	C++, JS, Julia, Go, Python, R, Scala, Perl, Matlab	+	+	+	+	+	Sync/ async	Data parallel, model parallel	256/16[7]	+ (MKL-DNN)	-	+/+
PaddlePaddle	C++, Python	Python	+	+	+	+	+	sync	Data parallel	+/+	+ (MKL BLAS + DNN)	-	+/+
Microsoft Cognitive Toolkit	C++	Python, C++, BrainScript	+	+	+	+	+	Sync/ async	Data parallel	8/8	+ (MKL-DNN)	-	+/+
Deeplearning4j	C++, Java	Java, Scala, Clojure, Python, Kotlin	+	+	+	-	-	sync	Data parallel	+/+	+ (MKL BLAS + DNN)	?	+/-
Keras	Python	Python, R	+	+	+	+	+	Sync/[8] async	Data parallel	+/+[9]	+ (MKL BLAS + DNN)	+	+/-

Most of the frameworks are implemented in C++ for high performance but provide APIs in many languages. The most popular API is for Python, which allows you to quickly develop prototypes of neural networks. An exception is the Torch framework, which is written in C and Lua and provides APIs in the same languages. However, Torch is now inferior in popularity to PyTorch, which provides a Python API. Another exception is the DeepLearning4J framework, which is written in Java and provides an API for languages that use the JVM. Thanks to this, DeepLearning4J integrates well in distributed computing systems from the JVM ecosystem: Hadoop and Spark.

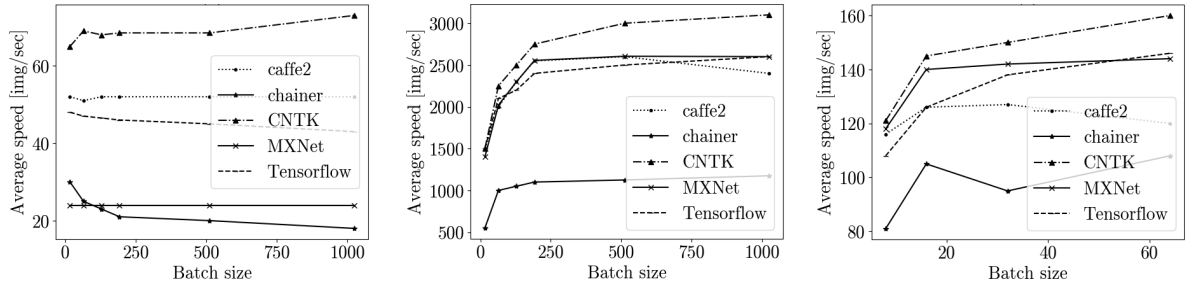
All frameworks support synchronous distributed training, while TensorFlow, Torch, MxNet, CNTK and Keras additionally support asynchronous. Also, all frameworks provide the possibility of distributed training using data parallelism, and TensorFlow, Caffe2, MxNet and Pytorch – using model parallelism. Thus, TensorFlow and MxNet have the advantage in the amount of distributed training modes.

[4] by using Horovod [28]
[5] PowerAI, 4GPU/node
[6] PowerAI, 4GPU/node
[7] 16 entities of AWS P2.16x1
[8] Asynchronous training is only possible with pure TF after model conversion
[9] Distributed multi-node training through the use of pure TF or Horovod

2.2. Scalability Comparison

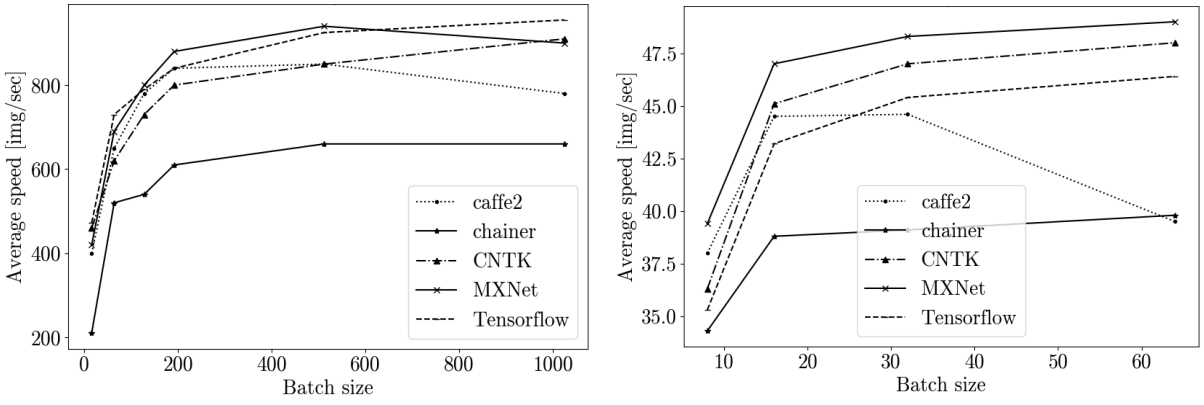
In a review of frameworks for distributed deep learning [63], the authors looked into the simplicity of parallelization, as well as performance on both several GPUs and several nodes. For testing, the authors used the AWS P2.xlarge cloud platform, where each entity was equipped with one NVIDIA Tesla K80 and four 2.7GHz Intel processors. For communication between nodes, 10Gbps Ethernet is used, which can greatly affect their performance, but the authors argue that if all the training data was downloaded to the nodes beforehand, then this communication network is sufficient to transfer model parameters. Authors selected CNTK, TensorFlow, Caffe2, MXNet, and Chainer (not covered in this review). Also, OpenMPI 3.0.0 was used. All experiments were conducted for the ResNet50 neural network; Cifar-10 and ImageNet were used as data for training and testing.

As mentioned earlier, the article looked into the simplicity of the framework for distributed training. The authors concluded that TensorFlow has the most complex architecture and parallelization methods for the user, which is why TensorFlow is excluded in some tests. There is only one scenario with the CPU – for inference, all other scenarios use the GPU.



(a) Forward in CPU for Cifar-10 (b) Forward in GPU for Cifar-10 (c) Forward in GPU for ImageNet

Figure 2. Frameworks performance for image classification (inference, measured in images/sec)



(a) Forward+Backward in GPU for Cifar-10 (b) Forward+Backward in GPU for ImageNet

Figure 3. Frameworks performance on GPUs, measured in images/sec

As you can see in the figures above, Chainer loses everywhere in terms of performance. On the CPU side, CNTK is clearly ahead of the competition in terms of performance, which shows how much this framework is optimized for working on the CPU. On the GPU side, all frameworks except Chainer show good performance, but you can highlight CNTK in the classification, and MXNet in training – these frameworks showed the best results.

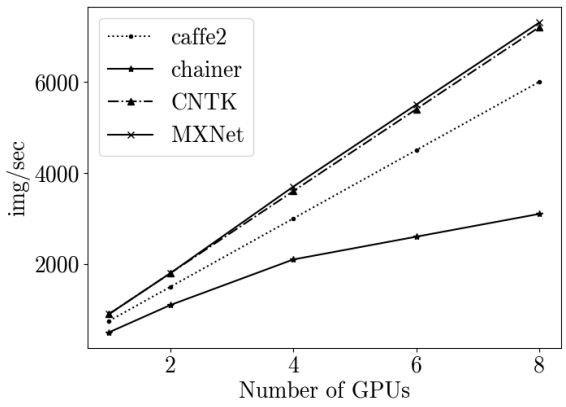
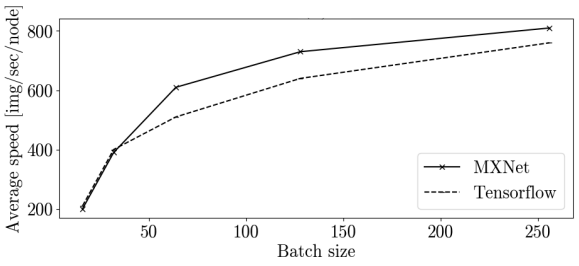
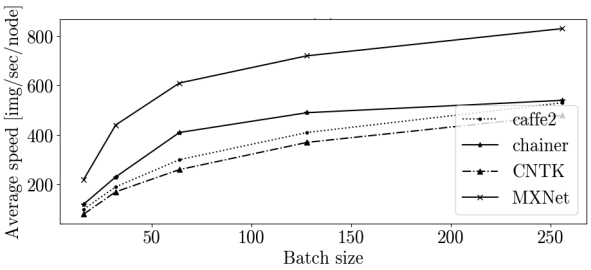


Figure 4. Performance with synchronous distributed multi-GPU training on Cifar-10

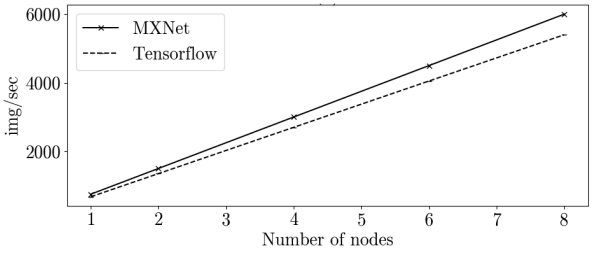
Figure 4 shows the performance of distributed multi-GPU training. In the version of TensorFlow, used by the authors, it was necessary to manually implement the mechanism for updating parameters after synchronization, which is why the authors excluded the framework from this test. As you can see, CNTK and MXNet clearly stand out by showing near-linear acceleration and having better performance. The scalability of Caffe2 is also good, but it lags behind due to the training speed of 1 GPU, which is lower than that of competing frameworks.



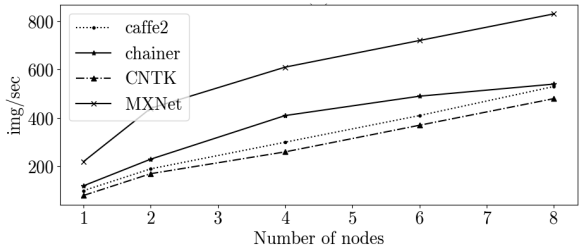
(a) 8 nodes with asynchronous update



(b) 8 nodes with synchronous update



(c) Fixed batch size of 128 with asynchronous update



(d) Fixed batch size of 128 with synchronous

Figure 5. Performance with distributed multi-node training for ResNet-50 on Cifar-10

Figure 5 shows the results of multi-node training performance. The authors tested scalability to just 8 nodes (1 GPU per node), but it still gives an idea of which framework scales better. TensorFlow was excluded from the tests with synchronous training due to the lack of a ready-made solution for this use case. As you can see from the graphs, MXNet is the clear leader, which is far ahead of all the frameworks in all tests, showing scalability close to linear.

Table 2. The processing speed of the batch on a different number of GPUs for various neural network architectures, in sec

		# of GK210		
		1	2	4
FCN-R	Caffe	0.239	0.131	0.094
	CNTK	0.1811	0.111	0.072
	TF	0.208	0.144	0.121
	MXNet	0.184	0.104	0.86
	Torch	0.165	0.110	0.112
AlexNet-R	Caffe	0.137	0.085	0.047
	CNTK	0.108	0.062	0.037
	TF	0.385	0.332	0.321
	MXNet	0.122	0.070	0.041
	Torch	0.141	0.077	0.046
ResNet-56	Caffe	0.378	0.254	0.177
	CNTK	0.562	0.351	0.170
	TF	0.523	0.291	0.197
	MXNet	0.270	0.167	0.101
	Torch	0.301	0.182	0.096

In terms of scalability, MXNet generally shows the best results. However, the authors also checked the quality of the models in [63], since a higher speed of the framework does not mean that the quality of the resulting model will be higher. The training speed of MXNet is 1.5+ times higher than that of TensorFlow; however, MXNet in 30 epochs cannot achieve the accuracy on the test set, which TensorFlow reaches in 5, which indicates a very fast convergence in TensorFlow. Caffe2 can also be highlighted, which shows good convergence (the next after TensorFlow), but which in terms of training speed is very close to MXNet (faster than TensorFlow by around 1.5 times).

A similar review was made in [67], where several neural network architectures were selected and CNTK, MXNet, Caffe, Torch, and TF were tested. This article did not use multi-node training, only multi-GPU training (maximum 4). The processing speed of one batch on several GK210 GPUs is shown in Tab. 2. For AlexNet-R and ResNet-56, Cifar-10 was used as data, for FCN-R – MNIST. According to the results, you can see that TF scales very poorly on this platform. Torch and CNTK are the best in scalability, but despite this, Torch loses in speed to both CNTK and MXNet.

In terms of the rates of convergence, authors drew a conclusion, that Torch and CNTK successfully cope with FCN-R; MXNet with Torch show the best performance with AlexNet-R and ResNet-56.

In [49] authors explored Power AI DDL, which shows the results of the IBM-Caffe and Torch versions provided in PowerAI on ResNet-50 (ImageNet with 1K classes was used). The results are shown in Tab. 3 and 4.

You can notice that these frameworks from PowerAI show very good scalability on Power IBM platforms, and the frameworks were tested on a large number of nodes, which is very rare.

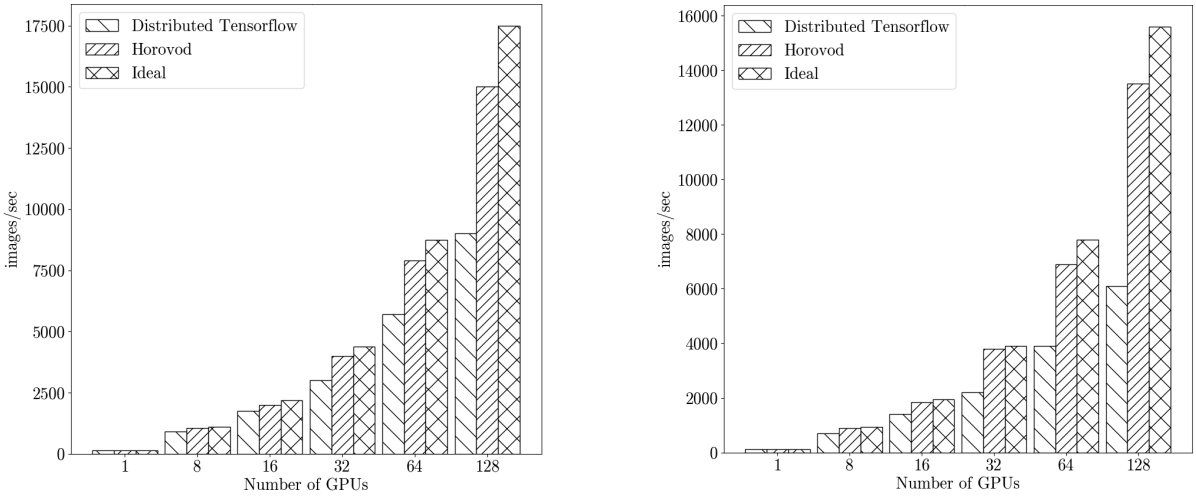
[28] shows the scalability of TF using the Horovod frontend for distributed training on two neural network architectures: Inception V3 and ResNet-101. The results are presented in Fig. 6. Details of what data and GPU are used were not indicated. It is worth noting that Horovod not

Table 3. Caffe multi-node training performance

#GPUs	4	8	16	32	64	128	256
#Nodes	1	2	4	8	16	32	64
Speedup	1.0	2.0	3.9	7.9	15.5	30.5	60.6
Scaling efficiency	1.00	1.00	0.98	0.99	0.97	0.95	0.95

Table 4. Torch multi-node training performance

#GPUs	4	16	64	96	128	192	256
#Nodes	1	4	16	24	32	48	64
Speedup	1.0	3.5	13.7	20.5	27.2	40.3	53.7
Scaling efficiency	1.00	0.86	0.86	0.85	0.85	0.84	0.84



(a) Inception V3

(b) ResNet-101

Figure 6. Multi-GPU distributed training performance of Horovod+TF

Table 5. Training speed comparison of PaddlePaddle and other Deep Learning Frameworks on selected models, in img/sec

	SE-ResNeXt50		**YOLOv3**	
	Paddle 1.5.0	PyTorch 1.1.0	Paddle 1.5.0	MXNet
1 GPU	168.334	163.130	29.901	18.578
8 GPUs	843.348	595.274	58.175	35.574
	DeepLab V3+		**Transformer**	
	Paddle 1.5.0	TensorFlow 1.12.0	Paddle 1.5.0	TensorFlow 1.12.0
1 GPU	13.695	6.4	4.865	4.750
8 GPUs	59.721	16.508	4.227	2.302

only provides more impressive results, but also has a more convenient and simple parallelization interface than the default TensorFlow.

Developers of PaddlePaddle benchmarked and compared their framework with PyTorch, MXNet and TensorFlow [37]. They focused on testing single-node multi-gpu Distributed Training and used SE-ResNeXt50, Mask-RCNN, DeepLab V3+ etc. models for evaluation. It is worth noting, that in every test PaddlePaddle is compared to only one of the competitors, eg. PyTorch for SE-ResNeXt50 and TensorFlow for DeepLab V3+. Table 5 shows the results of the experiments. It is clear, that in these experiments PaddlePaddle is superior to all of the other frameworks in terms of training speed, but it is hard to evaluate the quality of the trained models, because developers did not consider the rate of convergence of the models.

In the case of PyTorch, developers provided scalability data in their talk at GTC 2019 (Fig. 7 and 8). As you can see, PyTorch shows very good scalability while increasing number of nodes. Developers also noted that switching to the c10d backend accelerates training by 19%.

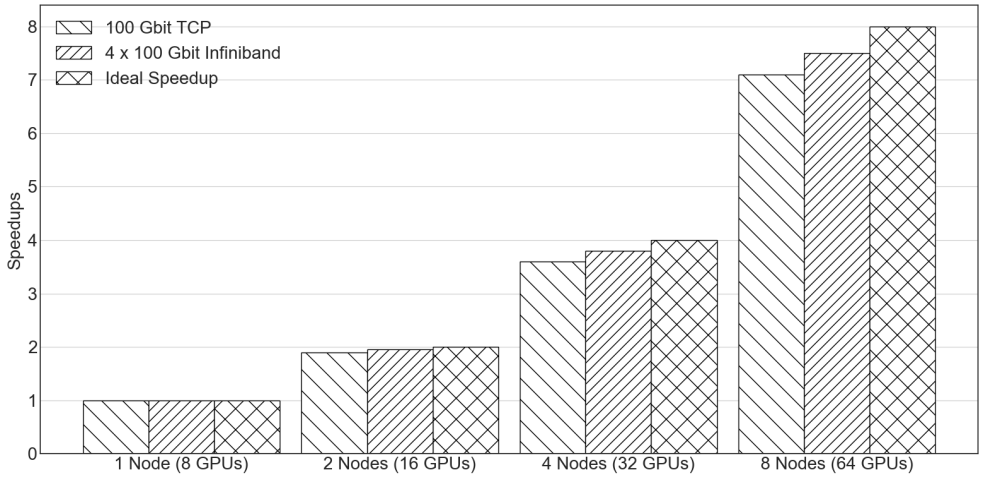


Figure 7. Pytorch 1.0 Distributed training performance on ResNet101 (NVIDIA V100)

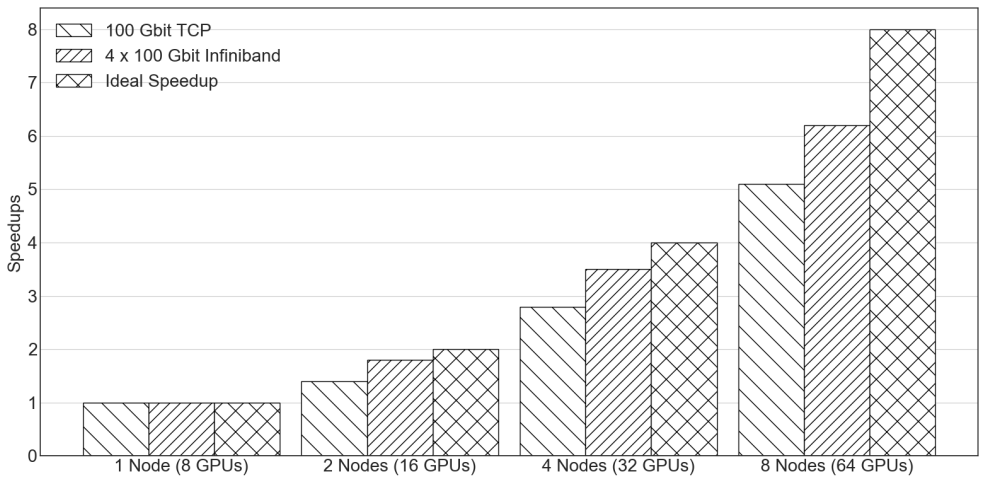


Figure 8. Pytorch 1.0 Distributed training performance on FAIR Seq (NVIDIA V100)

Conclusions

Distributed training of neural networks is becoming increasingly popular. It not only allows you to reduce the training time of a neural network, but also makes it possible to train large neural networks that cannot be fit into the memory of one machine. However, not all deep learning frameworks manage to develop distributed training quickly enough: while Multi-GPU and multithreaded training is present in all popular frameworks, the situation with distributed multi-node training is much worse. There is a good support for distributed training in the frameworks that were created with distributed training in mind: TensorFlow, MXNet and PadlePadle, as well as PyTorch and DeepLearning4J (due to integration with the Spark infrastructure).

The most popular method of distributed training is synchronous training with data parallelism. Tools for model parallelism have only recently begun to actively develop. There is Mesh TensorFlow solution; PyTorch provides the ability to manually implement model parallelism. Effective training on large supercomputers with a large number of nodes is impossible without model parallelism. We can expect that active researches will be carried out and new frameworks will appear in this area in the near future.

A large role is played not only by the performance and technical capabilities of the framework, but also by the ease of use of distributed training. This is because in order to search for a neural network that provides the necessary quality for solving the problem, it is necessary to train a large number of neural networks with different architectures and hyperparameters. Therefore, third-party libraries such as Horovod, which make distributed training using the TensorFlow, PyTorch, and MXNet frameworks easy and convenient, are gaining popularity. The TensorFlow framework is moving in the same direction, where in version 2.0 Keras has become the recommended high-level API which has integrated all distributed training tools.

You can also speed up the search for the most suitable neural network architectures using automated optimization tools for distributed hyperparameters, such as HyperOpt [48], Tune [61], Keras Tuner [27], etc. Due to the growing popularity of distributed training of neural networks, one can also expect the development of tools for distributed optimization of hyperparameters.

The process of consolidation of training frameworks for deep neural networks should not be missed out, primarily based on frameworks with the support of large Internet companies with large financial and computing resources: TensorFlow from Google and PyTorch from Facebook. The Keras framework has been included into TensorFlow; Horovod is underway to be included into the TensorFlow Distribution Strategy API [15]. PyTorch included the capabilities of the classic Torch, and it also included the Caffe2 framework, which has powerful distributed training tools. Other frameworks, unfortunately, do not develop in the field of distributed training as fast as TensorFlow and PyTorch do. It is most likely that in the future the trend of consolidating and including open projects for distributed training of neural networks and optimization of hyperparameters into TensorFlow and PyTorch will continue.

Hardware architectures specially designed for training neural networks, such as Google's Tensor Processing Unit (TPU) [8], Graphcore's Intelligence Processing Unit (IPU) [23], Nervana [21] from Intel and others, are also of interest for the development of distributed training. These architectures not only allow you to accelerate the training of neural networks, but also significantly affect the development of deep learning frameworks. In particular, the Mesh TensorFlow library was developed on the assumption that it will work on an n-dimensional grid of computing devices, which is typical for the cluster architecture on TPU [66]. The performance and quality of training a neural network in Mesh TensorFlow was evaluated in a cluster with

TPU. Although Mesh TensorFlow may work in a cluster with a different architecture, in that case performance will be lower. It is likely that over time the integration of DL frameworks and specialized equipment will become so deep and effective that it will become unprofitable to use clusters with a CPU or GPU to train neural networks.

Acknowledgements

The results described in this paper were obtained with the financial support of the grant from the Russian Federation President Fund (MK-2330.2019.9).

References

1. Apache MXNet. `https://mxnet.apache.org/`, accessed: 2019-12-03

2. Apache MXNet crash course. `https://beta.mxnet.io/guide/crash-course/index.html`, accessed: 2019-12-03

3. Caffe-MPI for deep learning. `https://github.com/Caffe-MPI/Caffe-MPI.github.io`, accessed: 2019-12-03

4. Caffe, Multi-GPU usage. `https://github.com/BVLC/caffe/blob/master/docs/multigpu.md`, accessed: 2019-12-03

5. Caffe2. `https://caffe2.ai/docs`, accessed: 2019-12-03

6. Caffe2, ResNet-50 training example. `https://github.com/pytorch/pytorch/blob/master/caffe2/python/examples/imagenet_trainer.py`, accessed: 2019-12-03

7. Caffe2, Synchronous SGD. `https://caffe2.ai/docs/SynchronousSGD.html`, accessed: 2019-12-03

8. Cloud TPU. `https://cloud.google.com/tpu/`, accessed: 2019-12-03

9. Deep learning with multiple GPUs on Rescale: Torch. `https://blog.rescale.com/deep-learning-with-multiple-gpus-on-rescale-torch`, accessed: 2019-12-03

10. Deeplearning4J. `https://deeplearning4j.org`, accessed: 2019-12-03

11. DeepLearning4J: Deep learning with Java, Spark and Power. `https://www.ibm.com/developerworks/community/blogs/fe313521-2e95-46f2-817d-44a4f27eba32/entry/DeepLearning4J_Deep_Learning_with_Java_Spark_and_Power?lang=en`, accessed: 2019-12-03

12. Distributed deep learning with DL4J and Spark. `https://deeplearning4j.org/docs/latest/deeplearning4j-scaleout-intro`, accessed: 2019-12-03

13. Distributed deep learning with Horovod and PowerAI DDL. https://developer.ibm.com/linuxonpower/2018/08/24/distributed-deep-learning-horovod-powerai-ddl/, accessed: 2019-12-03

14. Distributed training in TensorFlow. https://www.tensorflow.org/guide/distribute_strategy, accessed: 2019-12-03

15. Distribution strategy - revised API. https://github.com/tensorflow/community/blob/master/rfcs/20181016-replicator.md, accessed: 2019-12-03

16. DL4J, parallel training. https://deeplearning4j.org/tutorials/14-parallel-training, accessed: 2019-12-03

17. Getting started with Intel optimization for MXNet*. https://software.intel.com/en-us/articles/getting-started-with-intel-optimization-for-mxnet, accessed: 2019-12-03

18. Getting started with the Keras functional API. https://keras.io/getting-started/functional-api-guide/, accessed: 2019-12-03

19. Guide to multi-node training with Intel Distribution of Caffe. https://github.com/intel/caffe/wiki/Multinode-guide, accessed: 2019-12-03

20. Horovod. https://github.com/uber/horovod, accessed: 2019-12-03

21. Intel Nervana neural network processors. https://www.intel.ai/nervana-nnp, accessed: 2019-12-03

22. Intel PyTorch. https://github.com/intel/pytorch, accessed: 2019-12-03

23. Intelligence processing unit. https://www.graphcore.ai/technology, accessed: 2019-12-03

24. Intel Distribution of Caffe. https://github.com/intel/caffe, accessed: 2019-12-03

25. IntelSoftware Optimization for Torch. https://github.com/intel/torch, accessed: 2019-12-03

26. Keras: Deep learning library for MXNet, TensorFlow and Theano. https://github.com/dmlc/keras, accessed: 2019-12-03

27. Keras tuner. https://github.com/keras-team/keras-tuner, accessed: 2019-12-03

28. Meet Horovod: Ubers open source distributed deep learning framework for TensorFlow. https://eng.uber.com/horovod/, accessed: 2019-12-03

29. Microsoft cognitive toolkit. https://www.microsoft.com/en-us/cognitive-toolkit, accessed: 2019-12-03

30. Microsoft cognitive toolkit, multiple GPUs and machines. https://docs.microsoft.com/en-us/cognitive-toolkit/multiple-gpus-and-machines, accessed: 2019-12-03

31. The most popular language for machine learning and data science is ... https://www.kdnuggets.com/2017/01/most-popular-language-machine-learning-data-science.html, accessed: 2019-12-03

32. MXNet, training on multiple GPUs with gluon. https://gluon.mxnet.io/chapter07_distributed-learning/multiple-gpus-gluon.html, accessed: 2019-12-03

33. MXNet, training with multiple GPUs from scratch. https://gluon.mxnet.io/chapter07_distributed-learning/multiple-gpus-scratch.html, accessed: 2019-12-03

34. MXNet, training with multiple GPUs using model parallelism. https://mxnet.apache.org/api/faq/model_parallel_lstm, accessed: 2019-12-03

35. NVCaffe. https://docs.nvidia.com/deeplearning/frameworks/caffe-user-guide/index.html#pullnvcaffe, accessed: 2019-12-03

36. PaddlePaddle. https://github.com/PaddlePaddle/Paddle, accessed: 2019-12-03

37. PaddlePaddle benchmark. https://github.com/PaddlePaddle/benchmark, accessed: 2019-12-03

38. PaddlePaddle, manual for distributed training with fluid. https://www.paddlepaddle.org.cn/documentation/docs/en/1.5/user_guides/howto/training/cluster_howto_en.html#training-in-the-parameter-server-manner, accessed: 2019-12-03

39. PaddlePaddle, parallel executor. https://www.paddlepaddle.org.cn/documentation/docs/en/1.6/api_guides/low_level/parallel_executor_en.html, accessed: 2019-12-03

40. PaddlePaddle, single-node training. https://www.paddlepaddle.org.cn/documentation/docs/en/1.5/user_guides/howto/training/single_node_en.html, accessed: 2019-12-03

41. Protocol buffers. https://developers.google.com/protocol-buffers/, accessed: 2019-12-03

42. PyTorch. https://pytorch.org/, accessed: 2019-12-03

43. PyTorch, model parallel best practices. https://pytorch.org/tutorials/intermediate/model_parallel_tutorial.html, accessed: 2019-12-03

44. Scaling Keras model training to multiple GPUs. https://devblogs.nvidia.com/scaling-keras-training-multiple-gpus/, accessed: 2019-12-03

45. TensorFlow Roadmap. https://www.tensorflow.org/community/roadmap, accessed: 2019-12-03

46. TorchMPI. https://github.com/facebookresearch/TorchMPI, accessed: 2019-12-03

47. Abadi, M., Barham, P., Chen, J., et al.: TensorFlow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 265–283 (2016), https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

48. Bergstra, J., Yamins, D., Cox, D.D.: Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In: Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28. pp. I–115–I–123. ICML'13, JMLR.org (2013), http://dl.acm.org/citation.cfm?id=3042817.3042832

49. Cho, M., Finkler, U., Kumar, S., et al.: Powerai DDL. CoRR abs/1708.02188 (2017), http://arxiv.org/abs/1708.02188

50. Chollet, F., et al.: Keras. https://keras.io (2015)

51. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: A Matlab-like environment for machine learning. In: BigLearn, NIPS Workshop (2011)

52. Dean, J., Corrado, G.S., Monga, R., et al.: Large scale distributed deep networks. In: Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1. pp. 1223–1231. NIPS'12, Curran Associates Inc., USA (2012), http://dl.acm.org/citation.cfm?id=2999134.2999271

53. Devlin, J., Chang, M., Lee, K., et al.: BERT: pre-training of deep bidirectional transformers for language understanding. CoRR abs/1810.04805 (2018), http://arxiv.org/abs/1810.04805

54. Goyal, P., Dollár, P., Girshick, R.B., et al.: Accurate, large minibatch SGD: training ImageNet in 1 hour. CoRR abs/1706.02677 (2017), http://arxiv.org/abs/1706.02677

55. He, K., Zhang, X., Ren, S., et al.: Deep residual learning for image recognition. CoRR abs/1512.03385 (2015), http://arxiv.org/abs/1512.03385

56. Huang, X., Baker, J., Reddy, R.: A historical perspective of speech recognition. Commun. ACM 57(1), 94–103 (Jan 2014), DOI: 10.1145/2500887

57. Jia, Y., Shelhamer, E., Donahue, J., et al.: Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014)

58. Johnson, M., Schuster, M., Le, Q.V., et al.: Google's multilingual neural machine translation system: Enabling zero-shot translation. CoRR abs/1611.04558 (2016), http://arxiv.org/abs/1611.04558

59. Krizhevsky, A., Sutskever, I., Hinton, et al.: ImageNet classification with deep convolutional neural networks. Commun. ACM 60(6), 84–90 (May 2017), DOI: 10.1145/3065386

60. Kurth, T., Zhang, J., Satish, N., et al.: Deep learning at 15PF: Supervised and semi-supervised classification for scientific data. CoRR abs/1708.05256 (2017), http://arxiv.org/abs/1708.05256

61. Liaw, R., Liang, E., Nishihara, R., et al.: Tune: A research platform for distributed model selection and training. CoRR abs/1807.05118 (2018), http://arxiv.org/abs/1807.05118

62. Litjens, G.J.S., Kooi, T., Bejnordi, B.E., et al.: A survey on deep learning in medical image analysis. CoRR abs/1702.05747 (2017), http://arxiv.org/abs/1702.05747

63. Liu, J., Liu, J., Dutta, J., et al.: Usability study of distributed deep learning frameworks for convolutional neural networks (2018)

64. Radford, A., Wu, J., Child, R., et al.: Language models are unsupervised multitask learners (2018), https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

65. Sergeev, A., Balso, M.D.: Horovod: fast and easy distributed deep learning in TensorFlow. arXiv preprint arXiv:1802.05799 (2018)

66. Shazeer, N., Cheng, Y., Parmar, N., et al.: Mesh-TensorFlow: Deep learning for supercomputers. CoRR abs/1811.02084 (2018), http://arxiv.org/abs/1811.02084

67. Shi, S., Wang, Q., Xu, P., Chu, X.: Benchmarking state-of-the-art deep learning software tools. CoRR abs/1608.07249 (2016), http://arxiv.org/abs/1608.07249

68. Strom, N.: Scalable distributed DNN training using commodity GPU cloud computing. In: INTERSPEECH Interspeech (2015)

69. Szegedy, C., Liu, W., Jia, Y., et al.: Going deeper with convolutions. CoRR abs/1409.4842 (2014), http://arxiv.org/abs/1409.4842

70. Wang, L., Chen, Z., Liu, Y., et al.: A unified optimization approach for CNN model inference on integrated GPUs. CoRR abs/1907.02154 (2019), http://arxiv.org/abs/1907.02154

71. Xiong, W., Droppo, J., Huang, X., et al.: The Microsoft 2016 conversational speech recognition system. CoRR abs/1609.03528 (2016), http://arxiv.org/abs/1609.03528