Collecting HPC Applications Processing Characteristics to Facilitate Co-scheduling

Ruslan Kuchumov(✉) and Vladimir Korkhov

Saint Petersburg State University, 7/9 Universitetskaya nab., St. Petersburg 199034, Russia
kuchumovri@gmail.com, v.korkhov@spbu.ru

Abstract. In this paper we describe typical HPC workloads in terms of scheduling theory models. In particular, we cover machine environments that are common for high performance computing (HPC) field, possible objective functions and available jobs characteristics. We also describe resources that are required by HPC applications and how to monitor and control their usage rates. We provide the basis for defining mathematical model for application resource usage and validate it on experimental data.

Keywords: High performance computing · Co-scheduling · Scheduling theory

1 Introduction

The broader goal of this research is to reduce queue wait time of high performance computing (HPC) applications in cluster schedulers by applying co-scheduling strategy. Co-scheduling allows to assign multiple applications that have different requirements for resources to a single cluster node that have enough resources to execute them simultaneously and without interference. For example, some applications that are only disk IO-intensive can be executed simultaneously with applications that only network-intensive as they are using completely different resources.

This strategy allows to improve resources utilization of a cluster and reduce wait of jobs in the queue as compared to the commonly used scheduling strategies where a node can execute a single application at a time. Now, a lot of attention in the scientific community is focused on the feasibility of this strategy, and in particular on practical aspects [11].

Since co-scheduling requires more information about jobs resource requirements that is used in a common scheduler (e.g. number of cores and time of computation) it's important to find out these jobs parameters and metrics and to collect them together in a mathematical model that can later be used for making scheduling decisions.

© Springer Nature Switzerland AG 2020
O. Gervasi et al. (Eds.): ICCSA 2020, LNCS 12254, pp. 168–182, 2020.
https://doi.org/10.1007/978-3-030-58817-5_14

Scheduling theory has emerged long before HPC field. Its main focus is to formalize scheduling problems and to apply different mathematical methods for creating schedules that are subject to certain constraints and the same time optimize the objective function.

In scheduling theory, models are distinguished by their notation, which consists of three fields usually denoted as α, β and γ. Each field may be a comma-separated list of words. The first field describes the machine environment, the second describes the task characteristics and constraints, and the last one – objective function. Later in this paper we will use term "job" as a model for HPC applications and "machine" as a modeling term for cluster computing node.

In classical theory, there are important assumptions [4] that may limit its applicability to the co-scheduling problem. At first, each job can be processed by at most one machine and each machine can process at most one job (operation). Second, job execution time does not change in time. Third, job execution time is known in advance. In the latest development of the scheduling theory at least one of these assumptions is changed.

In the problem of co-scheduling all of the propositions of classical scheduling theory are changed, which makes the problem of defining the model challenging. Moreover, from a practical point of view, there are also a lot of open questions.

Sharing a common resource between multiple applications without introducing some sort of contentions is not always possible. Additional information regarding job's resources consumption has to be collected and taken into account during scheduling. What makes matter worse is that required information is not easily available, it can not be provided by the user at the job submission stage, and it is not easy to obtain it by monitoring jobs behavior during its execution. There is a lot of work done already by scientific community on collecting such information by interpreting available metrics, such as hardware counters or operating system events.

In this paper we will focus instead on collecting and abstracting this information for defining a formal model.

The rest of this paper is organized as follows. In Sect. 2 we describe possible ways and their rationale for defining machine environment. In Sect. 3 we describe possible objective functions for the model. In Sect. 4 we describe which job characteristics from the scheduling theory are common in HPC. In Sect. 5 we expand on job resources characteristics that are relevant for co-scheduling and propose an abstraction for describing these resources. In Sect. 6 we validate our hypotheses from the previous section using benchmarks in cluster environments.

2 Machine Environments

Scheduling theory defines multiple different machine environments [10]. Machines are considered to process a single job or stay idle at any point in time.

There are three commonly used environments with parallel machines that are distinguished by job processing speed. The first one is identical machines, where

all jobs are executed at the same rate. The second one is uniform machines, where machines have different processing speeds. The third one is the environment with unrelated machines, that is, the speed of processing of each job depends on the machine where it is being processed.

There are also environments with dedicated machines. In this case it is assumed that a job consists of multiple stages or operations which are performed on different machines. Each job must be executed on every machine. There are flow shop, open shop and job shop environments and their variation. In the flow shop each job must be processed in a specific order. In job shop each job may have a specific route for processing. In open shop there is no constrains to jobs processing routes.

In the environments above, it is assumed that machines are available continuously and do not have downtime. However, there are also environments that consider machines' non-availability periods.

In HPC systems, there are clusters with computational nodes that can process any job. These nodes are not dedicated and they can process any eligible job. Each node of a cluster consists of multiple CPU cores. Typical HPC job is a hierarchy of processes and threads, some processes can be executed on different nodes and may use cluster interconnect for its communications. Threads, on the other hand, are spawned by a process and can only be executed within a single node, as they share a common memory. Threads are assigned for execution at CPU cores by operating system scheduler. Usually, this scheduler implements some sort of fair-scheduling policy.

In order to model scheduling in HPC environment, we need to define machine environment and decide what to use as a machine. If we assume that cluster nodes can not be overcommitted with jobs, that is, each node can execute a single job at a time, then nodes can be used as machines in scheduling theory. This model can be applied to the commonly used cluster schedulers, that do not allow node overcommitting.

Since co-scheduling in HPC implies overcommiting nodes with multiple jobs, then they can not be represented by machine models in classical scheduling theory. As an alternative, CPU cores in each node can be modeled as machines, since they can not execute multiple threads simultaneously. But, this would introduce multiple layers of complexity. At first, CPU cores should be grouped into sets that represent each node. Then, jobs have to be defined as a hierarchy of processes and threads, where processes can be executed on different nodes and threads can only be within the CPU cores of the same node. Lastly, scheduling problems also become hierarchical: there is scheduling of processes between nodes and there is scheduling of threads between CPU cores. The last one is done by the operating system, but in this model definition its scheduling policy has to be considered.

Another alternative is to deviate from the classical scheduling theory and to model cluster nodes as machines but allow for execution of multiple jobs at any point in time. In this case, CPU cores can be modeled later as one of the resources.

If the cluster nodes are heterogeneous, then they can be modeled as a set of parallel machines. In a more general case, computing nodes may have different hardware, which results in different job execution speeds, so the model would be with uniform machines. In even more general case, hardware may have a different effect on each job (e.g. there can be different kinds of accelerators that are utilized by some jobs and not by others) which requires a more general model with unrelated machines.

In some cases, the network interconnect between computing nodes should be taken into account. For example, there can be different bandwidth and latency between nodes in the cluster depending on the relative position of nodes in the cluster. As a result, the job processing speed may depend on the position of assigned nodes within the network topology. Some models consider different network topologies in their machine environments [2].

3 Objective Functions

In scheduling theory there is a large variety of objective functions. For example, the makespan – completion time of the last job in the schedule, flowtime – the sum of completion times of all jobs, there are many due-dates related objective functions (e.g. lateness or tardiness). Sometimes job parameters in objective functions can be weighed (e.g. weighted completion times), weights can also be functions of job completion times. Some objective functions may not be regular. In some models multiple objective functions are used.

In HPC commonly considered objective functions can be classified into completion time related functions (makespan and flowtime), fairness objective functions and resource utilization objective function (e.g. cost of computation, power consumption).

4 Job Processing Characteristics

In the scheduling theory, each model may define multiple processing characteristics. These characteristics affect the set of feasible schedules and the process of decision making for constructing an optimal schedule. Below we discuss some of these characteristics that are relevant for describing applications in HPC field.

Precedence constraint is one of those characteristics. It implies that there is predecessor/successor constraint defined for each job forming a decency graph defined for a set of jobs. In the scheduling theory different special cases of these graphs are usually considered such as in-tree, out-tree and chain trees. In HPC it is common to have dependencies between applications, for example, when one application may require data files created by another application.

Sequence-dependent setup-time defines the amount of time the job has to wait before it can start its processing. This required time may be different for each job and for each machine where the job is being processed. In some models there may be setup-times families that do not require waiting time between jobs from the same families. In HPC applications may require transmission of

large data files to the local disk storage or may require compilation for the local hardware architectures. These activities may take a long time and can be modeled as sequence-dependent setup-time.

Machine feasibility requires for each job to have a set of machines where it can be processed. There are usually special cases considered where feasibility sets are non-intersecting or nested. This requirement may appear in HPC when applications have specific requirements for the hardware environment where they are being processed. For example, some applications may require computing nodes with specific accelerators such as GPGPUs or FPGAs that may be located only on some nodes in the cluster.

When multiprocessor job characteristic is present in the model definition, it means that each job may require multiple machines at the same time for its processing. The number of machines is fixed. In HPC it is very common to have this requirement for distributed applications. Sometimes the special case of power of two number of machines is considered in the scheduling theory literature, which is also very common in HPC.

The alternative to having multiprocessor jobs is to have moldable and malleable jobs. Moldable jobs utilize as many machines as specified when the job is started and the set of these machines does not change during job processing. Malleable jobs allow to change the number of machines they are using during their processing. Similar to multiprocessor jobs moldable are common in HPC among distributed applications. Malleable jobs, on the other hand, are rare as implementing them would require additional scheduler-job negotiation protocol or would require jobs to be fault-tolerant.

There is also a large number of models which define resources and consider how applications may use them during their processing. For example, there are renewable resources (when usage in every time is constrained), non-renewable resources (when total consumption is constrained) and double-constrained resources (when both usage and consumption are constrained). In HPC applications also require resources, such as main memory, licenses, disk space, computation cost and so on. These resources may also be categorized into renewable, non-renewable and double-constrained categories. Below in the paper we describe some of the resources and provide some basis for the defined model.

5 Resource Usage Metrics

In this section we describe some resources that are commonly required by applications in HPC field. Some of these resources, like the number of computing nodes, do not change during application execution, others are constant and may change during the runtime. For the latest, we describe how to monitor those requirements in real time. Rates of usage of some of the resources can be changed either directly or indirectly affecting the application's execution time.

5.1 Required Number of Nodes

Applications that are commonly used in HPC are moldable and not malleable. When started they take in some form a list of nodes; processes can be started and after that, the number of nodes does not change. Some applications may be moldable, but they may require the number of nodes to be a power of two.

For MPI-based applications, that are very common in HPC, there are extension interfaces, such as PMIx [3] or MPI-FT [8], that allow to scale up or down the working application. There are also possibilities to migrate working MPI processes between nodes, such as BLCR [5] and CRIU [9]. It may change the number of nodes used by the application, but it would not change the number of MPI processes. As a result of migration, some nodes may be overcommitted with processes, but scaling up would not be possible.

Another option for scaling applications that support checkpoint restarts is to simply restart an application with a new number of nodes. In this case the application would continue from the latest checkpoint on a new nodes configuration.

In order for the application to be malleable for the number of nodes, it would have to be completely fault-tolerant. Some application-level HPC schedulers provide fault-tolerance: any node can be turned off and the application would continue working with the remaining nodes, when a new node is added the application would be using it as well.

Since all of these solutions require support from the application and are not completely transparent, they can not be taken into account for the general case.

Usually processing time of each application depends on the number of nodes. Some scheduling models take that into account and define some functional dependency. There are a lot of models of dependency between processing time and the number of nodes, usually linear and convex models are defined, some are more complicated and consider the diminishing returns on the number of nodes.

5.2 CPU Cores

Similar to the number of nodes, the number of threads in a running application can not be changed from the outside. Otherwise, it would require some sort of negotiation protocol between the application and the scheduler.

But unlike the number of nodes, the number of threads may change in the runtime. The application may spawn and terminate its threads, some threads may become idle for some time. This number of threads of the application can be observed as a function of time during application execution. In order to do so, it is sufficient to monitor the number of threads of the application and their states. Then, to count the number of required threads it is sufficient to count all active (runnable) threads.

Linux's procfs pseudo-filesystem provides information about the states of all threads. The thread, for example, maybe in the runnable state, indicating that it is waiting for its turn in the runqueue, it may be in interruptable sleeping state when it executes sleep instruction or waits for a lock or another event, it may

in an uninterruptible sleep state when it is performing disk I/O operations or in any other state. To find out which threads belong to the same application we have placed its parent process before starting an application in a control group, this way the kernel reports children threads and processes identifiers when they are spawned.

This number of the running threads can be used as the required number of CPU cores in every time point. If this number is larger than the number of available CPU cores, then it means that some threads are ready to run, but they are waiting in run-queues for their turn. This number may also decrease when threads enter sleeping states, for example, when the are waiting on a lock or performing disk I/O operations.

In the ideal case, all of the runnable threads should run in parallel. For example, if they are performing inter-thread communications through the shared memory and they are assigned to the same CPU core, then the round trip time of these communications would include wait time in the queue. Besides that, over-committing would cause frequent context switching and possible cache invalidations, if threads are using different memory domains. Although it may introduce significant delays, changing the number of available CPU cores in runtime is possible.

Another metric related to the usage of CPU cores by the application is the number of executed instructions. This metrics depends on application behavior (i.e. which instruction applications execute), the hardware as different CPU architectures may have different speeds for the same instructions and on the way the application was compiled. There is also no simple way to control it directly on per-application basis. Nevertheless, we have found a use for it in our experiments as a measure of how application execution speed is affected by limits in other resources.

5.3 Memory Bus Bandwidth

Access to the main memory is shared between all CPU cores. In a simple case, each CPU core assesses memory through the same memory controller which communicates over the sequential bus with memory modules. Memory bandwidth of the bus is limited, and, in general, the communication frequency of the bus is lower than the clock frequency of CPU cores. As a result, a single CPU core may fully saturate memory bandwidth if it does not work with CPU caches effectively.

Because of that, memory bandwidth can be considered as a shared resource for the applications running on different CPU cores within the same computing node. When, for example, two applications that are capable of utilizing full memory bandwidth are scheduled together, they would interfere with each other and it would lead to the slowdown of their execution time.

For our purposes, we estimated memory throughput of the application with the number of cache misses at the last cache level. When cache miss happens at the last level, the memory controller accesses the main memory through the memory bus. The number of cache misses is provided by

CPU hardware counters and it can be periodically monitored by Linux kernel (PERF_COUNT_HW_CACHE_MISSES perf event). Clearly, this metric depends on cache configuration, and can not be reproduced on different hardware.

Estimating the required memory bandwidth of an application in a portable, hardware-independent way is not trivial. The reason for that is multiple levels of memory caches. Some regions of the memory may be cached, and when an application is accessing these regions, the memory accesses may be satisfied from the cache without reaching the memory bus. Associative cache, memory prefetch and the fact that some caches are shared between CPU cores make estimation even harder. There are many research papers on this topic, e.g. [6].

The maximum memory bus bandwidth available to an application can not be controlled directly. Having bandwidth requirements for each application, the decision maker may schedule for simultaneous execution applications that together do not consume all available bandwidth. In case combined memory bus utilization of all applications reaches the maximum value, the performance of all applications would degrade, as memory accesses would take more time.

5.4 Network Bandwidth

For distributed applications that require multiple computation nodes and an interconnect network between them for communication, network bandwidth can be considered as another shared resource. Network bandwidth may be shared between applications not only when they are running on the same set of nodes, but also when their sets of nodes do not intersect and they are connected to the same switch with limited bandwidth. When all available bandwidth is utilized communication time between nodes would increase so as the total execution time of an application.

There are multiple ways to measure network throughput of each application. For example, network packets coming from the application may be marked, using control groups or iptables rules, and then counted by network filters. Another possible way is to create virtual network interfaces using network namespaces for each application and count the number received and transmitted bytes on these interfaces. For our purposes and as we work in an isolated environment, it was sufficient to count the number of bytes transmitted by the physical network interface.

Network bandwidth available to each application can be controlled directly using Linux kernel traffic control policies. Several different queuing disciplines allow to shape network traffic and change network bandwidth. For our purposes and environment it was sufficient to use HTB (Hierarchical Token Bucket) queuing discipline.

Network bandwidth can also be shared equally between all running applications using SFQ (Stochastic Fairness Queue). In this case, all available bandwidth would be shared between all applications that require network access. So, for example, when at some time point there is only one application that transmits data, it would receive all available bandwidth. We have described that in more detail in our previous paper [7].

5.5 Resource Usage Model

Assume that the aforementioned metrics of resource consumption are measured for each application in ideal conditions when no other application was running at the same time and interfering with the one being measured. Having these metrics along with application execution time is not sufficient for making decisions for co-scheduling. The reason for that is that when an application is sharing common resources with another application, its resource usage metrics may change from the one measured in the ideal conditions. They are likely to decrease in this case in co-scheduling conditions which would affect total execution time.

Regardless of available resource shares and other applications that are running simultaneously, the total amount of work performed by the application should not change from the ideal conditions. This total amount of work can be represented by the total amount of CPU instructions, the total number of bytes written and read from the memory and the total number of bytes transmitted and received through the network card.

Another characteristic of the application that is invariant to the changes of available rates is the sequence of instructions. For example, for every 10 executed instructions application may issue 1 instruction that causes LLC cache miss, or for every 10 Kb of received data, the application may execute 300 million instructions. As a result, ratios between resource rates stay constant regardless of resource limits. This assumption is applicable only to the time intervals when resource rates are constant or periodical with a very small period.

On a larger scale, an application may consist of multiple stages that have different resource rates and amounts of work. For example, these stages may include initialization, a loop of computation and data synchronizations followed by the output stage. In each of these stages application may perform a different amount of work at different rates, so resource rate limitations may have different effects of the duration of each stage.

Because of all of these features, we propose to describe resource consumption of an application as a sequence of low-resolution stages, where each stage is defined by the total amount of work that needs to be done and a function describing dependencies between resource usage rates. Using these parameters it is possible to estimate the total execution time of the application and its required resource rates depending of the resource limits.

Since during a single stage application may require multiple resources, its execution time may not depend linearly on resource limits, according to the Amdahl's law. For example, an application may perform 1 unit of work with one resource and another 1 unit of work with another resource per single time interval. If we increase the rate for one of the resources two times, the total amount of work during this interval would increase 1.5 times. Linear dependency is only possible when all of the resource rates are scaled at the same time, which is not feasible in real world as it would require controlling all of the activities of the application.

6 Experiments

To validate our resource model we have used benchmark applications from NAS Parallel Benchmarks suite [1], executed it with different resource constraints and monitored how they affect application behavior. There are two goals for performing these experiments. The first one is to show that an application performs a constant amount of work regardless of the limits on resource rates. The second one to show that dependencies between resource rates do not depend on resource limits.

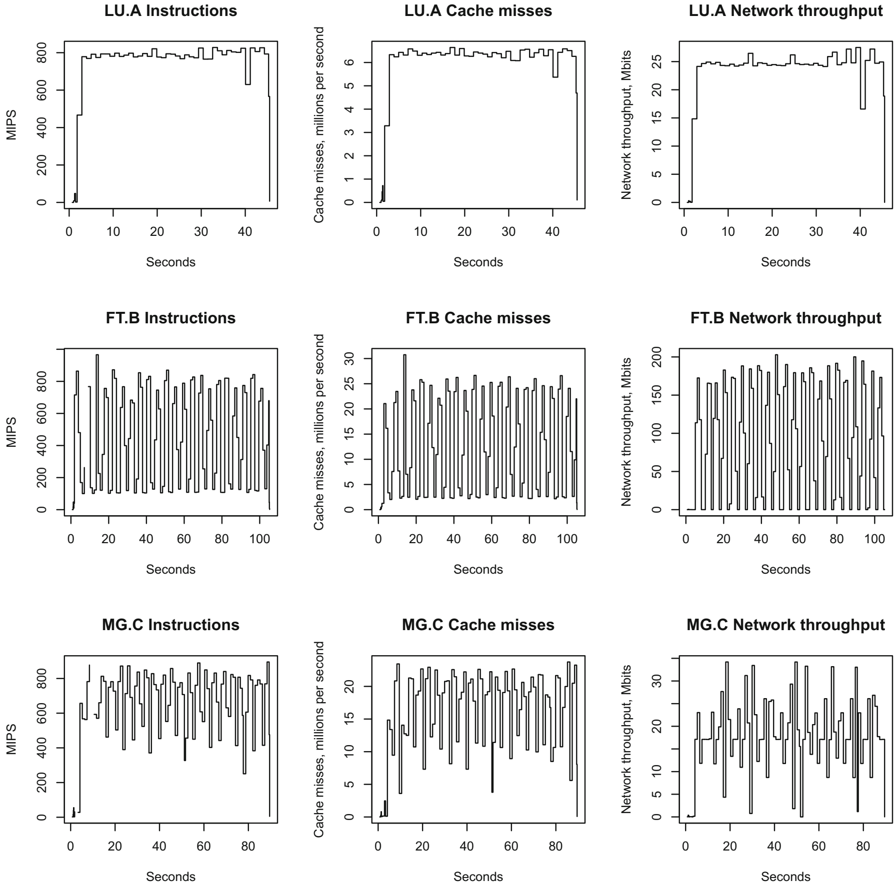

Fig. 1. Resource usage rates in every second of application run-time. For these applications resource rates are almost constant (LU) or periodical with a small period (FT and MG)

We have implemented a program for monitoring resource usage rates periodically (every 0.5 s). We have monitored the number of instructions per second

(in millions per second), the memory bus throughput (in millions cache misses per second) and the network throughput (number of transmitted and received megabits per second). The technique for measuring these resources was described in the previous sections. Example of observed resource usage rates for some applications are shown in Fig. 1.

Applications in NAS Parallel Benchmark (NPB) suite solve mathematical problems that are commonly used in HPC applications. Among these tests are the following: CG (conjugate gradient), MG (Multi-Grid), FT (discrete 3D fast Fourier Transform), BT (Block Tri-diagonal solver), LU (Lower-Upper Gauss-Seidel solver) and UA (Unstructured Adaptive mesh). Sizes of these problems are distinguished by so-called classes and they are denoted by the last letter in the benchmark name (“A”, “B”, “C” and so on).

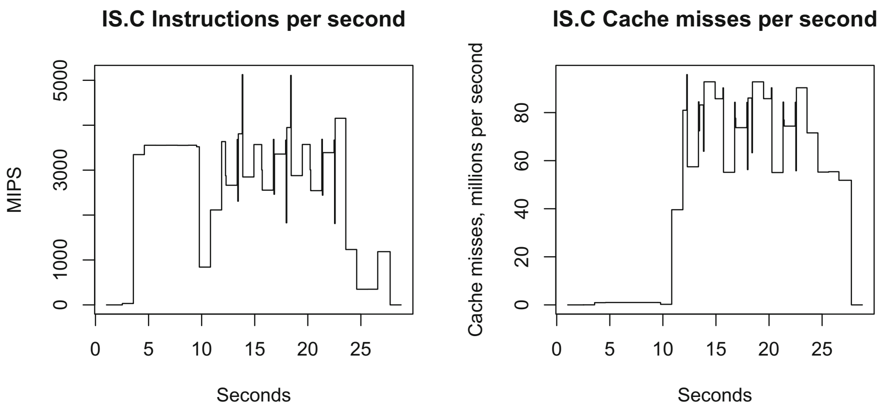

Fig. 2. In IS.C benchmark three stages can be noticed. The first one ends at 10th second, the second one ends at 22nd second.

For our goals, we have selected only the applications that have either constant resource usage rates or periodical rates with a small period. Some applications in NPB, have multiple stages with different rates (for example, IS.C in Fig. 2). As each stage would require a separate model, we have omitted such tests.

We have executed MPI versions of NPB on the 4-node cluster with 4 cores and have measured resource rates on one of them (the one that does not have MPI master process). For controlling memory bandwidth we have used HTB queening discipline in Linux traffic control.

In Table 1 and Fig. 3 you can see that for all tests there are linear dependencies between network transmission rates and the number of instructions and caches misses per second. R^2 values of linear regression for these parameters are close to 1. The total amount of transmitted (and received) bytes is constant and is not affected by the changes in network bandwidth.

The total number of instructions and cache misses is affected by the changes in network transmission rates. The possible explanation for that is when the

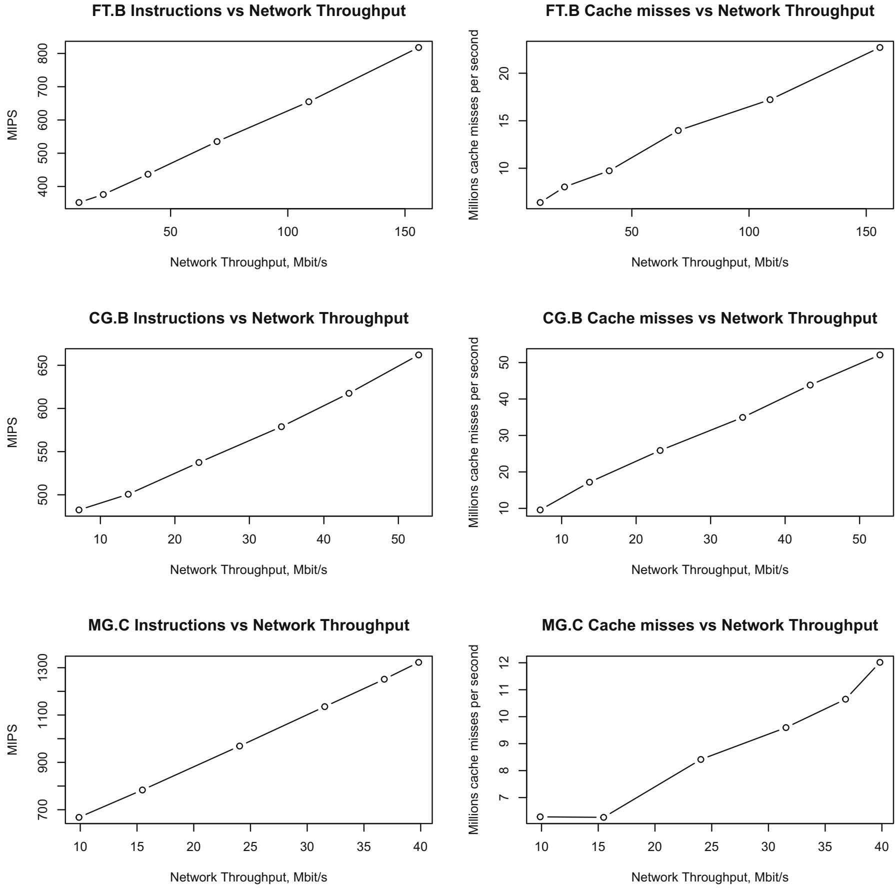

Fig. 3. An average number of instructions per second and cache misses per second as a function of the network throughput of an application.

Table 1. Per-benchmark results for MPI NPB tests. First two columns contains R^2 values for the number of instructions per second and memory bus throughput versus application's network throughput linear regression models. In the next columns there are absolute deviation from the mean value (in percents) of the number of transmitted bytes, executed instructions and cache misses.

	MIPS vs TP, R2	Misses vs TP, R2	TX Data, %	Instr., %	Misses., %
CG.B	0.9956	0.9985	0.3305	56.3411	10.8913
LU.A	0.9998	0.9620	0.2474	23.9812	24.9657
BT.A	0.9999	0.9789	0.6518	32.4141	39.5725
FT.B	0.9992	0.9955	0.5180	59.9892	47.1960
MG.C	0.9968	0.9929	0.4928	23.6905	10.0647

application is waiting for network data to be received or transmitted, it performs instructions in user-space which are counted in the total number. This may happen when a single thread waits for data on the socket in the system space and other threads wait for notification from this thread in the user-space.

In order to show that the total number of executed instructions and cache misses is also constant regardless of the constraints, we have executed OpenMP version of NPB applications. We have executed it on a single node with 4 cores. 2 cores of this node were used for running and measuring the test itself, other 2 cores were used to run workloads that differ in their intensity of memory bus communications. The reason for this approach is that memory bus bandwidth can not be controlled directly as, for example, network bandwidth, and we can affect it only by interacting with it.

For controlling memory bus bandwidth indirectly, we have used a synthetic test that allocates 512 Mb of memory in a loop, then accesses every 64th or 4096th byte of this memory and after that frees this memory. Such access pattern causes an allocation of physical memory pages and populating cache levels with cache lines from this memory, as the result available bandwidth for the test application decreases. We have also run multiple copies of this synthetic test to have a different effect on memory bus bandwidth of the application.

Table 2. Per-benchmark results for OpenMP NPB tests. First columns contains R^2 values for the number of instructions per second versus LLC cache misses linear regression model. In the next column there are absolute deviation from the mean value (in percents) of the number of executed instructions and cache misses.

	MIPS vs Cache misses, R2	Instr., %	Misses., %
CG.B	0.9999403	0.1745595	0.0883977
FT.B	0.9988642	0.0023119	0.7274650
LU.A	0.9554370	0.4167756	2.1089845
MG.B	0.6842153	0.0574678	8.9376413
UA.A	0.9879625	0.9258455	1.7547919

In Table 2 you can see that similar to the MPI version, in OpenMP version of NPB benchmarks there is a linear dependency between the number of cache misses and the number of executed instructions per second and R^2 values are close to 1 for all tests. Also, the total number of executed instruction and cache misses does not depend on the memory bus throughput of the application.

As mentioned previously, execution time has non-linear dependency from the resource usage rates. You can see in Fig. 4 that for network throughput the dependency is not linear, although for memory bus throughput this dependency is close to linear.

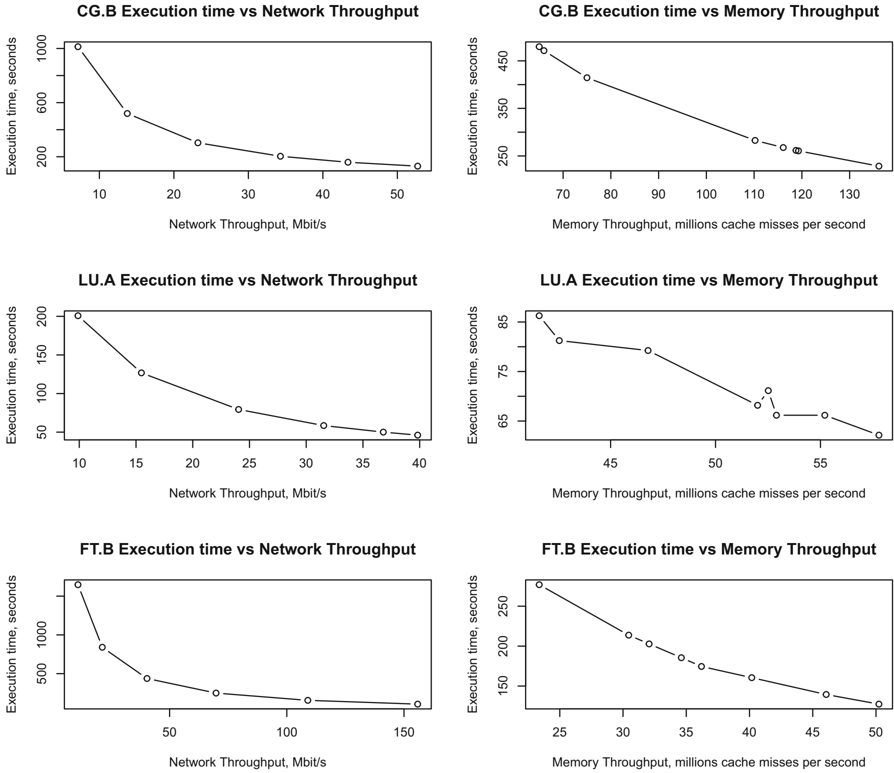

Fig. 4. Benchmark execution time as a function of the number of last level cache misses and network throughput

7 Conclusion

In this paper, we have described typical HPC workloads in terms of scheduling theory models. In particular, we have covered possible ways of defining machine environments in scheduling theory model to represent cluster environments in HPC field. We have also described which job processing characteristics from the scheduling theory can be applied for describing HPC applications.

We have covered different application's resource requirements such as the number of computing nodes, CPU cores, memory bandwidth and network bandwidth. We have described how to monitor them during application execution, how to control them when it is possible.

For these resources we have provided an abstraction based on the following propositions. The first one is that the total amount of consumed resource does not change regardless of resources usage rates. The second one is that relations between resource usage rates are linear and also do not change regardless of resource usage rates. Using these propositions it possible to describe application

usage rate as a sequence of stages where each stage would be defined by the amount of consumed resources and relations between resource usage rates.

We have tested these propositions using NAS Parallel Benchmark tests in 4-node cluster and we have found these propositions are valid. For different resource constraints the total amount of consumed resources does not deviate from the mean value more than 1% and coefficient of determination for the linear model of resource rates is very close to 1.

Using this approach it is possible now to estimate application execution time as a function of resource constraints. This later would allow to make decisions by how much the applications may be constrained when they are scheduled together in a co-scheduling strategy.

Acknowledgements. Research has been supported by the RFBR grant No. 19-37-90138.

References

1. Bailey, D.H., et al.: The NAS parallel benchmarks. Int. J. Supercomput. Appl. **5**(3), 63–73 (1991)
2. Błażewicz, J., Ecker, K.H., Pesch, E., Schmidt, G., Sterna, M., Weglarz, J.: Handbook on Scheduling: From Theory to Practice. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-319-99849-7
3. Castain, R.H., Hursey, J., Bouteiller, A., Solt, D.: Pmix: process management for exascale environments. Parallel Comput. **79**, 9–29 (2018)
4. Gawiejnowicz, S.: Time-Dependent Scheduling. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69446-5
5. Hargrove, P.H., Duell, J.C.: Berkeley lab checkpoint/restart (BLCR) for Linux clusters. In: Journal of Physics: Conference Series, vol. 46, p. 494. IOP Publishing (2006)
6. Haritatos, A.H., Papadopoulou, N., Nikas, K., Goumas, G., Koziris, N.: Contention-aware scheduling policies for fairness and throughput. Co-Sched. HPC Appl. **28**, 22 (2017)
7. Kuchumov, R., Korkhov, V.: Fair resource allocation for running HPC workloads simultaneously. In: Misra, S., et al. (eds.) ICCSA 2019. LNCS, vol. 11622, pp. 740–751. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24305-0_55
8. Louca, S., Neophytou, N., Lachanas, A., Evripidou, P.: MPI-FT: Portable fault tolerance scheme for MPI. Parallel Process. Lett. **10**(04), 371–382 (2000)
9. Pickartz, S., Eiling, N., Lankes, S., Razik, L., Monti, A.: Migrating LinuX containers using CRIU. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) ISC High Performance 2016. LNCS, vol. 9945, pp. 674–684. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46079-6_47
10. Pinedo, M.: Scheduling: Theory, Algorithms, and Systems, vol. 5. Springer, Heidelberg (2012). https://doi.org/10.1007/978-1-4614-2361-4
11. Trinitis, C., Weidendorfer, J.: Co-scheduling of HPC Applications, vol. 28. IOS Press (2017)