Performance Models for Data Transfers: A Case Study with Molecular Chemistry Kernels

Suraj Kumar
Pacific Northwest National Laboratory
Richland, Washington, USA
suraj.kumar@pnnl.gov

Lionel Eyraud-Dubois
Inria Bordeaux – Sud-Ouest
Université de Bordeaux, France
lionel.eyraud-dubois@inria.fr

Sriram Krishnamoorthy
Pacific Northwest National Laboratory
Richland, Washington, USA
sriram@pnnl.gov

ABSTRACT

In distributed memory systems, it is paramount to develop strategies to overlap the data transfers between memory nodes with the computations in order to exploit the full potential of these systems. In this paper, we consider the problem of determining the order of data transfers between two memory nodes for a set of independent tasks with the objective of minimizing the makespan. We prove that, with limited memory capacity, the problem of obtaining the optimal order of data transfers is NP-complete. We propose several heuristics to determine this order and discuss the conditions that might be favorable to different heuristics. We analyze our heuristics on traces obtained by running 2 molecular chemistry kernels, namely, Hartree–Fock (HF) and Coupled Cluster Single Double (CCSD) on 10 nodes of an HPC system. Our results show that some of our heuristics achieve significant overlap for moderate memory capacities and resulting in makespans that are very close to the lower bound.

CCS CONCEPTS

• **Computer systems organization** → High Performance Computing ; • **Computing Methodologies** → Modeling and Simulation; • **General** → Performance.

KEYWORDS

Communication Scheduling, Memory Nodes, Runtime Systems, Communication-Computation Overlap, Molecular Chemistry

ACM Reference Format:
Suraj Kumar, Lionel Eyraud-Dubois, and Sriram Krishnamoorthy. 2019. Performance Models for Data Transfers: A Case Study with Molecular Chemistry Kernels. In *Proceedings of ICPP 2019: 48th International Conference on Parallel Processing (ICPP '19).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the advent of multicore, and the use of accelerators, it is notoriously cumbersome to exploit the full capability of a machine. Indeed, there are several challenges that come into picture. First, every architecture provides its own efficacy and interface. Therefore, a steep learning curve is required for programmers to take good utilization of all resources. Second, scheduling is a well known NP-complete optimization problem, and hybrid and distributed resources make this problem harder (we refer [11] for a survey on the complexity of scheduling problems and [9] for a recent survey in the case of hybrid nodes). Third, due to shared buses and parallel resources, it is challenging to obtain a precise model based on prediction of computation and communication times. Fourth, the number of architectures has increased drastically in recent years, therefore it is almost impossible to develop hand tuned optimized code for all these architectures. All these observations led to the development of different task based runtime systems. Among several runtimes, we may cite QUARK [27] and PaRSEC [10] from from ICL, Univ. of Tennessee Knoxville (USA), StarPU [7] form Inria Bordeaux (France), Legion [8] from Stanford Univ. (USA), StarSs [12] from Barcelona Supercomputing Center (Spain), KAAPI [16] from Inria Grenoble (France). All these runtime systems allow programmers to express their algorithms at the abstract level in the form of direct acyclic graphs (DAG), where vertices represent computations and edges represent dependencies among them. Sometimes some static information is also provided along with the DAG, such as distance to exit (last) node as a priority or affinity of computation towards resources. The runtime is then responsible for managing scheduling of computations and communications, data transfers among different memories, computation-communication overlap, and load balance.

In the last few decades, we have witnessed a drastic improvement in the hardware to provide higher rate of computation, but comparatively smaller improvement has been achieved for the rate of data movement. With extreme scale computing, supercomputers face bottlenecks due to the need of large amount of data [2, 28]. Therefore, the HPC community is now focusing on avoiding, hiding and minimizing communication costs.

Certain applications such as dense linear algebra kernels have regular structure. Therefore, it is possible to associate priorities to computations, based on the task graph structure, and to use them at runtime to make the execution efficient. In irregular applications, programmers do not know the precise structure of the task graphs in advance: tasks are added recursively based on certain sentinel constraints. For such applications, the runtime system sees a set of independent tasks and schedules them on processing units of different memory spaces. It is extremely important for runtimes to decide the order of data transfers for these scheduled computations so as to maximize the overlap between computations and communications. This is the main topic of this paper. We prove

ICPP '19, August 05–08, 2019, Kyoto, Japan

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

that the order of communications on two memory nodes with the objective of minimizing the makespan is a NP-complete problem if the memory of the target node is limited. Our proof is inspired from work by [21], which applies a similar technique for 2-machine flowshop problem with bounded capacity. The main difference between both approaches is that they consider all tasks have the same occupation on the second machine and the memory occupation starts when the processing finishes on the first machine. On the contrary, our approach is designed for tasks appearing in scientific workloads whose memory requirements are highly irregular and we consider that memory is acquired before starting the data transfer on the communication resource. We propose different runtime strategies in order to maximize the overlap of computations and communications. We evaluate our strategies on the context of a cluster of homogeneous nodes. However, our approach is generic and easily adaptable to any system which operates on different memory spaces. Here are the important contributions of this paper:

- NP-completeness proof for the general data-transfer problem
- Different scheduling strategies with the objective to minimize the makespan
- Linear programming formulation of the problem
- Numerous experiments to assess the effectiveness of our strategies on molecular chemistry kernels

The outline of the paper is the following. Section 2 describes past work on the computations with limited memory and similar problems in the literature. In section 3, we present an algorithm to obtain the order of data transfers when there is not any memory capacity restriction. Then, we also prove that in general data transfer problem is NP-complete. In Section 4, we propose several heuristics and discuss the conditions that might be favorable to them. We mainly consider three categories of heuristics: static order based heuristics, dynamic selection based heuristics, and static order with dynamic correction based heuristics. Sections 5 describes our experimental setup and we evaluate our proposed heuristics on two molecular chemistry kernels in Section 6. Our results show that static order with dynamic correction based heuristics achieve good performance in most cases. We finally propose conclusions and perspectives in Section 7.

2 RELATED WORK

Historically there has been a great emphasis on the development of parallel algorithms and minimizing the complexity of computations. As the number of computation cores has increased drastically in recent years, supercomputers face bottleneck due to communication required by an application. Hence, in recent years the focus has changed towards developing communication avoiding algorithms, strategies to hiding communications and minimizing the data accessed by applications [28].

The problem of scheduling tasks has been highly studied in the literature and many formulations are known to be NP-complete [13]. Many static and dynamic strategies have been proposed and analyzed for scheduling tasks on heterogeneous resources [5, 26]. There is also a number of studies in the direction of task scheduling with the emphasis on improving locality and minimizing the communication cost [7, 26]. Stanisic et. al [25] proposed a heuristic to schedule tasks on a computational resource where most of its data is available. A similar approach has been adopted by Agullo et. al for the scheduling of sparse linear algebra kernels [6]. Predari et. al proposed heuristics to partition the task graph across a number of processors such that inter-processor communication can be minimized [22].

The problem considered in this paper also can be viewed as a flow shop problem: the communication link can be seen as a processing resource, and each task needs to first be handled by the communication link and then by the computational resource. Communication and computation times of a task can thus be considered as processing times on different machines. Johnson has provided scheduling strategies for 2 and 3-machine flow shop problems with infinite memory capacity [17]. 2-machine flow shop problem with finite buffer has been proven NP-complete by Papadimitriou et. al [21], in which a constraint is imposed on the number of tasks that can await execution on the second machine.

A number of other studies have focused on scheduling with limited memory and storage, starting with the work of register allocation for arithmetic expressions by Sethi and Ulman [24]. Sarkar et. al worked on the scheduling of graphs of smaller-grain tasks with limited memory, where each task requires homogeneous data size [23]. The same work has been extended by Marchal et. al for task graphs where memory requirement of each task is highly irregular [19].

3 PROBLEM FORMULATION

To exploit the full potential of a system it may be necessary to execute tasks on processing units where all of their data does not reside. A task may require all of its input data in local memory before starting the computation. There may be multiple tasks scheduled on a processing unit, which require to transfer data from the same memory node. Ordering data transfers for such tasks is very crucial for the communication-computation overlap, thus for the overall performance. In general, order of task execution with input and output data transfers can be viewed as a 3-machine flowshop problem, where processing time on the first machine is input data transfer time, processing time on the second machine is task computation time, and processing time on the third machine is output data transfer time; and the objective is to minimize the total makespan. This is a well known NP-complete problem [14].

In many cases, output data that needs to be retrieved after task execution is much smaller than the input data. It is often the case that future tasks running on the same memory node require output data of the past tasks. Therefore, most runtime systems transfer data to other memory nodes based on the demand – not immediately after they were produced. It is also possible that all output data can be stored in a preallocated separate buffer on a memory node. Hence, we do not consider output data explicitly in our analysis and assume that output data is negligible or stored in a separate buffer for each task. Thus problem considered here is more similar to a 2-machine flowshop problem. We prove that ordering the execution of such tasks with finite memory capacity is a NP-complete problem:

Problem DT : A set of tasks $ST = \{T_1, \cdots, T_n\}$ is scheduled on a processing unit P with memory unit M of capacity C. Input data for tasks of ST reside on another memory unit M'. $COMM_i$ is the communication time to transfer input data from M' to M for task

i and $COMP_i$ is the computation time of task i on P. We assume that these tasks do not produce any output data. There can be only one communication at a time, and P can only process one task at a time. A task uses an amount of memory in M from the start of its communication to the end of its computation.
Given L, is there a feasible schedule S for ST such that makespan of S, $\mu(S) \leq L$?

Given a schedule, $S_{\text{COMM}}(i)$ and $S_{\text{COMP}}(i)$ represent the start times of task i on communication and computation resources. A schedule is feasible if for every time t, the amount of memory required by all tasks such that $S_{\text{COMM}}(i) \leq t \leq S_{\text{COMP}}(i) + COMP_i$ is not more than the memory capacity C. For simplicity, we assume throughout the paper that tasks require memory only to store their input data, and thus that the amount of memory required by a task is proportional to its communication time. Without loss of generality, we consider in all examples of Sections 3 and 4 that the memory requirement of a task is equal to its communication time.

We call a task i compute intensive if $COMP_i \geq COMM_i$, and communication intensive otherwise.

3.1 Special Case: Infinite Memory

When the computational resource has a very large memory, our problem becomes a classic 2-machine flowshop problem: communication time is the processing time on the first machine and computation time is the processing time on the second machine. Johnson's algorithm [17] is known to provide an ordering for the tasks which results in an optimal makespan. This algorithm is rewritten in Algorithm 1.

Algorithm 1: Johnson's [17] algorithm (infinite memory case).

```
Divide ready tasks in two sets $S_1$ and $S_2$. If computation time
  of a task $T$ is not less than its communication time, then $T$ is
  in $S_1$ otherwise in $S_2$.
Sort $S_1$ in queue $Q$ by non-decreasing communication times
Sort $S_2$ in queue $Q'$ by non-increasing computation times
Append $Q'$ to $Q$
$\tau_{\text{COMM}} \leftarrow 0$        {Available time of communication resource}
$\tau_{\text{COMP}} \leftarrow 0$        {Available time of computation resource}
while $Q \neq \emptyset$ do
    Remove a task $T$ from beginning of $Q$ for processing
    $S_{\text{COMM}}(T) \leftarrow \tau_{\text{COMM}}$
    $S_{\text{COMP}}(T) \leftarrow max(S_{\text{COMM}}(T) + COMM_T, \tau_{\text{COMP}})$
    $\tau_{\text{COMM}} \leftarrow S_{\text{COMM}}(T) + COMM_T$
    $\tau_{\text{COMP}} \leftarrow S_{\text{COMP}}(T) + COMP_T$
end while
```

We also prove optimality of Algorithm 1 differently in an extended version [18].

3.2 Finite Memory

We now consider the general case, in which the memory limit is a constraint for the schedule. This is related to previous work by Papadimitriou et. al [21], in which the second machine can only handle a bounded number of tasks. Our problem generalizes this work to heterogeneous memory consumption among tasks, with an additional difference: memory usage starts at the beginning of the first part of a task (instead of at the end of the first part). This requires to provide a slightly different NP-completeness proof, as given below.

Theorem 3.1. *Problem DT is NP-complete.*

Proof. It is easy to see that the DT belongs in NP: given a schedule, one can check in linear time that at each start of a communication, the memory constraint is satisfied, and that task starts computation only after its input data is transferred to M.

In order to prove NP-hardness, we use a reduction from the well-known NP-complete problem 3 Partition [13]:
Three Partition Problem (3Par): Given a set of $3m$ integers $A = \{a_1, \cdots, a_{3m}\}$, is there a partition of A into m triplets $TR_i = \{a_{i_1}, a_{i_2}, a_{i_3}\}$, such that $\forall i, a_{i_1} + a_{i_2} + a_{i_3} = b$, where $b = (1/m) \sum a_i$?

Let us first show that 3Par problem reduces in polynomial time to problem DT. Suppose that we are given an instance $A = \{a_1, \cdots, a_{3m}\}$ of 3Par. It is immediately obvious that $a_i > 1$, since we can always add sufficiently large integers to the a_i values and scale the problem accordingly. This scaling will not affect in any way the existence of a solution for the instance of 3Par problem.

From such an instance, we define $x = max\{a_i : 1 \leq i \leq 3m\}$, and we construct an instance I of the problem DT with $4m + 1$ tasks, whose characteristics are given in Table 1.

Task	Communication time	Computation time
K_0	0	3
$K_1, \cdots, K_{m-1}$	$b' = b + 6x$	3
K_m	$b' = b + 6x$	0
$1 \leq i \leq 3m, A_i$	1	$a'_i = a_i + 2x$

Memory capacity: $C = b' + 3$
Target makespan: $L = m(b' + 3)$

Table 1: Definition of tasks in the reduction from 3Par.

We show that I has a schedule S with makespan at most L if and only if the original 3Par instance has a solution. Notice that the sum of communication times and the sum of computation times are both equal to L, therefore a valid schedule of makespan at most L has makespan exactly L, with no idle time on both resources. It indicates that the first task is K_0 and the last task is K_m.

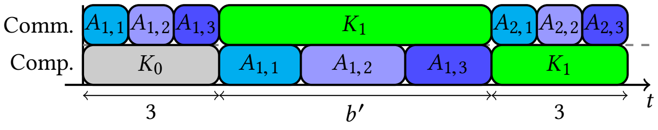


Figure 1: Pattern of feasible schedule S.

If the 3Par instance has a solution, A can be partitioned into m triplets $TR_i = \{a_{i_1}, a_{i_2}, a_{i_3}\}$ such that $\forall i, a_{i_1} + a_{i_2} + a_{i_3} = b$, then we can construct a feasible schedule S without idle times by the pattern depicted in Figure 1. The communications of tasks in TR_i take place during the computation of task K_{i-1}, and the computations of tasks in TR_i take place during the communication of task K_i. Since the memory capacity is $C = b' + 3$, all tasks from a triplet can fit in memory with a task K_i, and their computation times are exactly equal to the communication time of K_i. This schedule is thus feasible, and has length exactly L.

We now prove that any feasible schedule of I corresponds to a valid decomposition of A for 3Par. Indeed, we argue that every feasible schedule has to consist of m segments like the one shown in Figure 1. Each segment provides a triplet $\{a_{i_1}, a_{i_2}, a_{i_3}\}$ such that $a_{i_1} + a_{i_2} + a_{i_3} = b$.

Any schedule S of I having no idle time must start with K_0. We first show that no other K_i task can be active with K_0, otherwise we would get idle time on the computation resource. Indeed, the communication of such a task K_i would end at time at least $b' > 3 + 6x$, but at most two A_i tasks can be computed, and they end at time at most $3 + 2max\{a'_i : 1 \leq i \leq 3m\} = 3 + 6x$.

Hence three A_i tasks must follow K_0. The memory requirement of other K_i tasks is b' and $2b' > C$, therefore at any point in the schedule at most one K_i task can be active. Since the total duration of all K_i tasks is $3 + (m-1)(b'+3) + b' = m(b'+3) = L$, at each point in S exactly one K_i task is active.

With these K_i tasks in place, the schedule on the computation resource contains m slots of length exactly b', in which all $3m$ A_i tasks must fit without preemption. We thus define TR_i as the set of tasks which execute during the communication phase of task K_i. At each point in S, exactly one K_i task is active and the total memory capacity is $b' + 3$, hence TR_i contains exactly 3 A_i tasks. Since S has no idle time on the computation resource, the total computation time of tasks in TR_i is exactly b', and thus $a_{i_1} + a_{i_2} + a_{i_3} = b$. This partition is thus a valid solution for the 3Par instance A.

□

This theorem shows that adding a memory constraint to our problem makes it more difficult, just like it does for limited capacity 2-machine flowshop [21]. One additional difficulty of our problem however is that it may not be optimal to consider the same order on both machines.

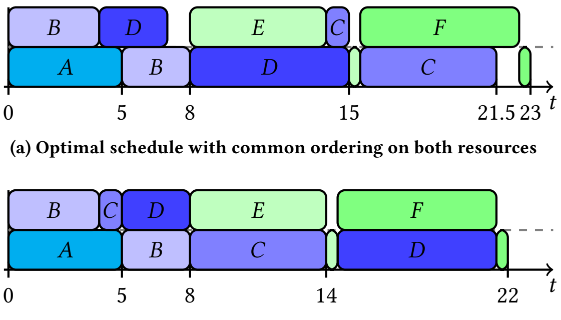
Proposition 3.2. *There exists an instance of DT for which in all optimal schedules, the communication order of tasks is different from their computation order.*

Task	Memory Req	Comm Time	Comp Time
A	0	0	5
B	4	4	3
C	1	1	6
D	3	3	7
E	6	6	0.5
F	7	7	0.5

Table 2: Example instance where ordering on both resources has to be different.

Proof. Consider the instance described on Table 2, in which memory capacity is $C = 10$. Figure 2a shows the best possible schedule when tasks are scheduled in the same order on both resources (obtained by exhaustive search). On the other hand, Figure 2b shows another schedule with lower makespan, in which the order is different.

In the infinite memory case, the standard proof that an optimal schedule exists with the same order on both resources claims that it is possible to swap two tasks which do not satisfy this property. On Figure 2, this would mean swapping tasks D and E. But the communication of task E can not start earlier because it would not fit in memory with tasks B and C, and delaying the computation of task E after task D would delay task F because E and F do not fit in memory together. We can see that this claim does not hold in the constrained memory case. □



(a) Optimal schedule with common ordering on both resources

(b) Schedule with different ordering on both resources

Figure 2: Schedules for the instance of Table 2 with a memory capacity of 10.

4 DATA TRANSFER ORDER HEURISTICS

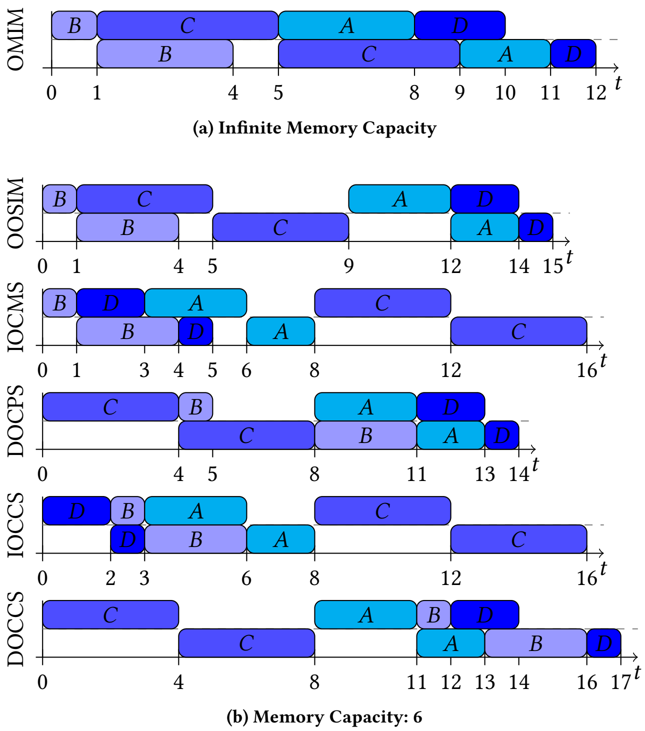
Algorithm 1 presented in Section 3 achieves an optimal makespan when there is no memory constraint. This optimal value indicates a lower bound on the makespan of the constrained case. We denote this value with *optimal makespan infinite memory* (*OMIM*). In this section, we propose different heuristics for the limited memory case, and we assess their efficiency with respect to *OMIM* in Section 6.

We classify our heuristics into mainly three categories. In the first category, the order of all computations and communications is computed in advance and the same order is followed on both resources. In the second category, the next task to schedule is dynamically chosen based on different criteria. The final category is based on combining strategies from the first two categories: a static order is precomputed and corrected dynamically to avoid idle time caused by memory limitations. In all of our strategies (except linear programming based strategy), communication and computations take place in the same order.

4.1 Static Ordering

In this class of strategies, we compute the order of processing in advance based on criteria such as communication time and computation time. After computing the order, we follow the same sequence on computation and communication resources and make sure that the memory constraint is respected at each point in the schedule.

In Algorithm 1, compute intensive tasks are sorted in increasing order of communication times. It allows tasks to utilize the computation resource maximally and make enough margin on the communication resource to accommodate more communication intensive tasks with maximum overlap. Communication intensive tasks are sorted in decreasing order of computation time, which allows tasks to utilize the margin created on communication resource. Hence, in this section, we obtain the orders by sorting tasks based on different combinations of communication and computation times.



(a) Infinite Memory Capacity

(b) Memory Capacity: 6

Figure 3: Static order heuristic schedules for Table 3.

Task	Memory Req	Comm Time	Comp Time
A	3	3	2
B	1	1	3
C	4	4	4
D	2	2	1

Table 3: A task set for static order schedules.

i) *order of optimal strategy infinite memory* (*OOSIM*): This heuristic uses the order given by Algorithm 1, but respects the memory constraint at each point in the schedule. Hence the makespan of this heuristic may be completely different from *OMIM*.

ii) *increasing order of communication strategy* (*IOCMS*): Tasks are ordered in non-decreasing order of communication time.

iii) *decreasing order of computation strategy* (*DOCPS*): Tasks are ordered in non-increasing order of computation time.

iv) *increasing order of communication plus computation strategy* (*IOCCS*): Tasks are ordered in non-decreasing order of the sum of their communication and computation times.

v) *decreasing order of communication plus computation strategy* (*DOCCS*): Tasks are ordered in non-increasing order of the sum of their communication and computation times.

In order to highlight the different behaviors of these static heuristics, we propose on Table 3 an example instance, and on Figure 3 the corresponding schedules for all these heuristics.

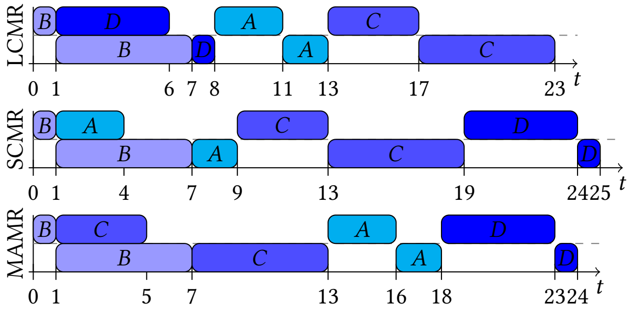


Figure 4: Different dynamic heuristic schedules for a task set of Table 4 with a memory capacity of 6.

Task	Memory Req	Comm Time	Comp Time
A	3	3	2
B	1	1	6
C	4	4	6
D	5	5	1

Table 4: A task set for dynamic schedules.

4.2 Dynamic Selection

Dynamic strategies are based on the following principle: when the communication resource is idle, a task is chosen based on a selection criterion which differs depending on the heuristic, among those which fit in memory and induce minimum idle time on the computation resource. For example, if the selection criterion is to choose a highly compute intensive task, then we compute the ratio of computation time and communication time for all tasks, and we select a task with the maximum ratio among those which induce minimum idle time on the computation resource and fit in the currently available memory. If no task fits in memory then we leave the resource idle at that point and proceed to the next event point.

i) *largest communication task respects memory restriction* (*LCMR*): A task with the largest communication time is chosen.

ii) *smallest communication task respects memory restriction* (*SCMR*): A task with the smallest communication time is chosen.

iii) *maximum accelerated task respects memory restriction* (*MAMR*): A task with the maximum ratio of computation time to communication time is chosen.

We highlight the different dynamic heuristics with the instance described on Table 4 (these heuristics are too similar on the instance from the previous class), and Figure 4 shows the corresponding schedules.

4.3 Static Order with Dynamic Correction

In this class of strategies, we precompute the order of tasks based on some criterion and then follow this ordering as much as possible. But when the communication resource is idle because the memory requirement of the next task is too high, then we select a task with a dynamic strategy. After a task is dynamically selected, we update the remaining order without this task. This class of strategies takes advantage of static information in the form of precomputed order and dynamic correction to minimize the idle time due to memory constraint.

In all strategies of this class, the initial order is *OMIM* order obtained by Algorithm 1. We define the following heuristics based on how we select a task from the set of tasks which fit in memory and induce minimum idle time on the computation resource. If no task fits in memory then we leave the resource idle and forward to the next event point.

i) *optimal order infinite memory largest communication task respects memory restriction* (*OOLCMR*): A task with the largest communication time is chosen from the set.
ii) *optimal order infinite memory smallest communication task respects memory restriction* (*OOSCMR*): A task with the smallest communication time is chosen from the set.
iii) *optimal order infinite memory maximum accelerated task respects memory restriction* (*OOMAMR*): A task with the maximum ratio of computation time to communication time is chosen from the set.

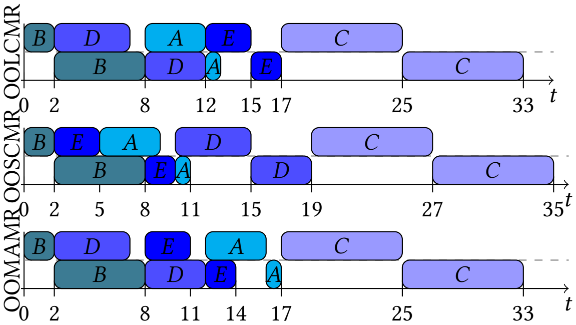


Figure 5: Different static order dynamic correction heuristic schedules for a task set of Table 5 with a memory capacity of 9. The *OMIM* order is *BCDEA*.

Task	Memory Req	Comm Time	Comp Time
A	4	4	1
B	2	2	6
C	8	8	8
D	5	5	4
E	3	3	2

Table 5: A task set for static order dynamic correction schedules.

As previously, we propose on Table 5 an example instance for this class of heuristics, and provide on Figure 5 the corresponding schedules.

4.4 Additional Heuristics from Previous Work

We also consider two other static heuristics for evaluation. The first heuristic is based on an algorithm, proposed by Gilmore and Gomory, to obtain the minimal cost sequence for a set of jobs [15]. This is a classical algorithm for 2-machine no-wait flow shop problem. In this algorithm, each job has a start and end state and a cost is associated to change the state. In our context, this cost can be seen as non-overlap time of computation for two adjacent tasks. Here is the main idea of this algorithm. Initially, a partial sequence of jobs is represented by a graph such that their overlap is maximum. Subsequently edges are greedily added to this graph to connect two components while minimizing the total non-overlap cost. When the graph is connected, an edge interchange mechanism is used to determine the sequence of jobs, which ensures that the sequence has minimal cost. More details can be found in the original paper [15] and our implementation is publicly available [3]. This algorithm does not take memory constraints into account and only provides the sequence of processing. We use this sequence with a memory capacity restriction just like for other static heuristics, and we call this heuristic *Gilmore-Gomory* (*GG*).

The second heuristic is based on the First-Fit algorithm for the bin packing problem. The idea of this heuristic is to identify groups of tasks which can fit in memory together, called *bins*. In First-Fit, tasks are considered in an arbitrary order and added to the first bin in which they can fit. If no suitable bin is found then a new bin is created and this task is added to it. When all tasks have been assigned to bins, we consider the sequence made of all tasks from the first bin, then tasks for the second bin, and so on. We call this heuristic *Bin Packing* (*BP*).

4.5 Solving Linear Program Iteratively

We use a mixed integer linear program to obtain the order of communications and computations. Recall that $COMP_i$ and $COMM_i$ represent computation and communication times of task i, and the memory capacity of the target system is C. In the linear program formulation, s_i and e_i (resp. s'_i and e'_i) represent the start and end times of communication (resp. computation) for task i, and $MC(i)$ is the memory capacity requirement of task i. The formulation also contains for each pair of tasks i and j *i)* a boolean variable a_{ij} to denote the order of i and j on the communication resource *ii)* a boolean variable b_{ij} to denote the order of i and j on the computation resource, and *iii)* a boolean variable c_{ij} to denote the order of s_i and e'_j.

Let $L = \sum_i (COMP_i + COMM_i)$. It is evident that $e_i = s_i + COMM_i$ and $e'_i = s'_i + COMP_i$. The linear program is given below.

$$
\begin{array}{lll}
 & \text{Minimize } l \text{ subject to:} & \\
\forall i, & e'_i \leq l & \text{(task } i \text{ completes)} \\
\forall i, & e_i \leq s'_i & \text{(task } i \text{ valid ordering)} \\
\forall i, \forall j \neq i, & \begin{cases} e_j \leq s_i + (1 - a_{ij})L \\ e_i \leq s_j + a_{ij}L \end{cases} & \begin{array}{l}\text{(exclusive use of} \\ \text{ communication link)}\end{array} \\
\forall i, \forall j \neq i, & \begin{cases} e'_j \leq s'_i + (1 - b_{ij})L \\ e'_i \leq s'_j + b_{ij}L \end{cases} & \begin{array}{l}\text{(exclusive use of} \\ \text{ computation resource)}\end{array} \\
\forall i, \forall j \neq i, & \begin{cases} e'_j \leq s_i + (1 - c_{ij})L \\ s_i < e'_j + c_{ij}L \end{cases} & \begin{array}{l}\text{(respect ordering} \\ \text{ of variables } c_{ij})\end{array} \\
\forall i, & \sum_{r \neq i} (a_{ir} - c_{ir}) MC(r) + MC(i) \leq C & \text{(memory constraint)}
\end{array}
$$

We use GLPK solver v4.65 to solve the above formulation. We also add the following constraints to help the solver: $\forall i, \forall j \neq i$, $a_{ij} + a_{ji} = 1$, $b_{ij} + b_{ji} = 1$, $c_{ij} \leq a_{ij}$, $c_{ij} \leq b_{ij}$, and $c_{ij} + c_{ji} \leq 1$. The solver was unable to solve this MILP at the scale of our interest in limited time. Hence, we solve the linear program iteratively for a small subset of size $k = 3, 4, 5, 6$: at the boundary of two iterations we fix the event (communication or computation) of an unfinished task started before the boundary point and consider other events flexible. The subsets are formed in the order in which tasks are submitted, which is arbitrary. For a given size k, we represent the makespan calculated by this heuristic as $lp.k$. We compute various $lp.k$ values for different memory capacities and observe that most of the other heuristics perform better than this heuristic. Hence, we do not include this heuristic for the comparison in Section 6. Performance comparison of different heuristics with MILP based heuristics is available in an extended version [18].

4.6 Favorable Situations for Heuristics

Based on the definition of proposed heuristics and the optimality of Algorithm 1, we discuss the scenarios which might be more favorable for each heuristic in Table 6. This allows programmers to use appropriate strategies to maximize communication-computation overlap for their applications. In this table, "moderate memory capacity" refers to the case where memory is constrained, but close to the maximal memory requirement of the *OMIM* schedule.

Heuristic	Favorable Situation
OOSIM	Memory capacity is not a restriction (Optimal)
IOCMS	Memory capacity is not a restriction and tasks are compute intensive (Optimal)
DOCPS	Memory capacity is not a restriction and tasks are communication intensive (Optimal)
IOCCS	Moderate memory capacity and most tasks are highly compute intensive
DOCCS	Moderate memory capacity and most tasks are highly communication intensive
LCMR	Limited memory capacity and significant percentage of tasks with large communication times are compute intensive
SCMR	Limited memory capacity and significant percentage of tasks with small communication times are compute intensive
MAMR	Limited memory capacity and significant percentage of all types of tasks
OOLCMR	Moderate memory capacity and significant percentage of slightly communication intensive tasks have large communication times
OOSCMR	Moderate memory capacity and significant percentage of slightly communication intensive tasks have small communication times
OOMAMR	Moderate memory capacity and significant percentage of all types of tasks

Table 6: Heuristics and their favorable scenarios.

Some of these favorable scenarios can be clearly observed in our experimental results, on Figures 7 and 9. For example, HF compute intensive tasks have small communication times, which explains why the *SCMR* heuristic exhibits very good performance in limited memory cases. CCSD has significant percentage of large as well as small types of slightly communication intensive tasks, and indeed the performance of *OOLCMR* and *OOSCMR* is very close to optimal in moderate memory cases.

5 EXPERIMENTAL SETTINGS

We consider a machine called Cascade [4], available at PNNL, for our experiments. We obtain traces by running two molecular chemistry applications, double precision version of HF and CCSD of NWChem [1] package on 10 nodes of this machine. Each node is composed of 16 Intel Xeon E5-2670 cores. NWChem takes advantages of a Partitioned Global Address Space Programming Model called Global Arrays (GA) [20] to use shared-memory programming APIs on distributed memory computers. GA dedicates one core of each node to handle other cores, hence we can view a node as being composed of 15 computational cores. We use 150 processes for each application and obtain 150 trace files. We run CCSD with Uracil molecules input and HF with SiOSi molecules (for Uracil molecules, HF has a much smaller workload, each processor executes only around 20 tasks, that is why we chose SiOSi input for HF execution). Each process executes around 300-800 tasks. Our data transfer model is quite simple and we consider that all data transfers between the local memory of each process and the GA memory take the same route. Modeling of different routes of data transfers for the same source-destination pair, bandwidth sharing for different source-destination pairs and network congestion is more challenging and part of our future work. This simple model is enough to provide insight to the application developers (or runtime system) about the ordering of data transfers for the same source-destination pair so as to maximize communication-computation overlap. Our model is easily adaptable to any source-destination pair when there is one fixed route between source and destination (such as between CPU and GPU, one copy engine to transfer data from CPU (resp. GPU) to GPU (resp. CPU)).

Both applications mainly perform two types of computations, tensor transpose and tensor contraction. HF expects to specify a tile size and we set it to 100, so that each core can be used efficiently. CCSD automatically determines tile sizes at different program points based on the input molecules. Hence, HF operates on almost homogeneous tiles while CCSD uses more heterogeneous tiles.

5.1 Workload Characteristics

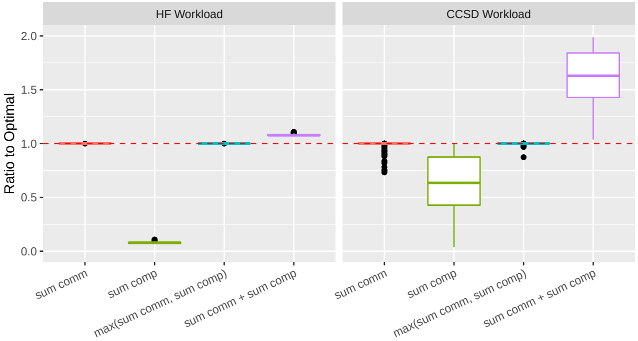


Figure 6: HF and CCSD tasks characteristics.

To get more insights into the considered workloads, we provide information about the instances we consider in Figure 6. For each

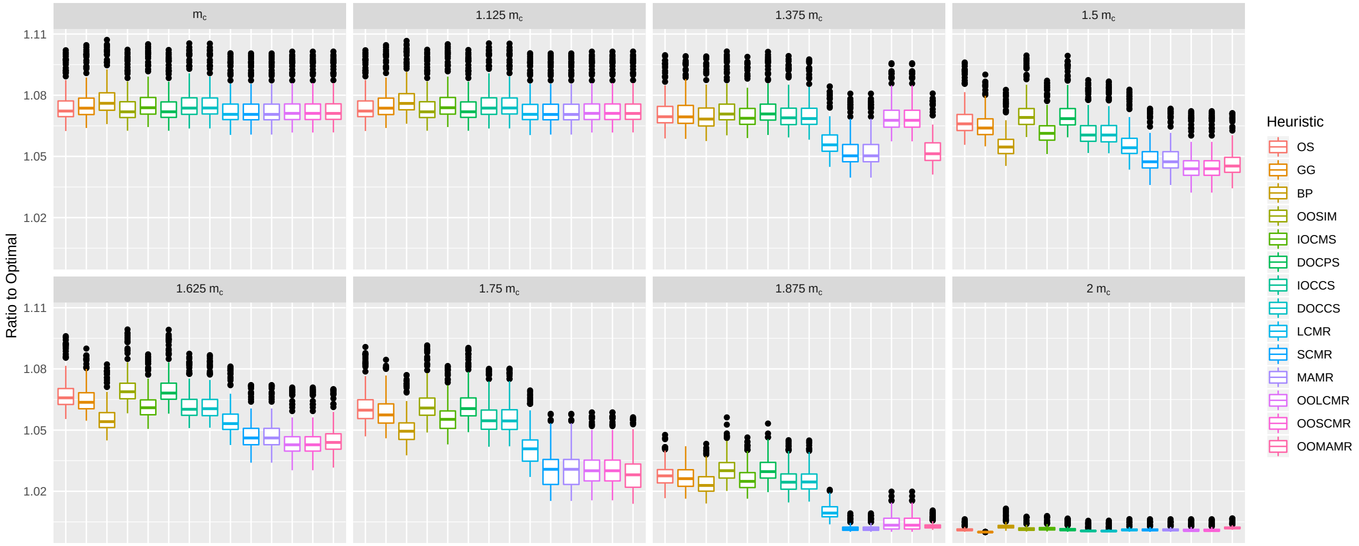


Figure 7: Comparison of different heuristics for HF with $m_c = 176KB$.

instance, we compute the sum of communication times (*sum comm*) and the sum of computation times (*sum comp*), and normalize it relatively to the *OMIM* value. Figure 6 also shows the maximum of both values, which is a lower bound on the possible makespan of a schedule for this instance, and their sum, which is an upper bound: this represents the makespan of the sequential schedule, obtained with zero overlap between computation and communication. We can see that HF is a communication intensive application and at most 20% overlap can be expected in the best scenario. On the other hand, in the CCSD workload, communications and computations are almost evenly distributed and a more significant overlap is possible.

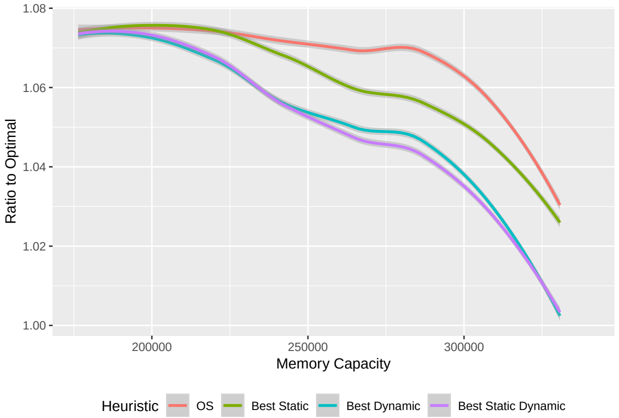


Figure 8: Best variants of all categories for HF.

6 EXPERIMENTAL RESULTS

We evaluate our scheduling heuristics for several memory capacities. From the obtained traces, we first determine the minimum requirement of the memory capacity m_c to execute all tasks. Then we observe the behavior of all heuristics with memory capacity m_c to $2m_c$, in increments of $0.125m_c$. Our performance metric is the ratio to optimal r: if heuristic H has makespan M_H on an instance, and the optimal makespan for the infinite memory case is *OMIM*, then $r(H) = \frac{M_H}{OMIM}$ (lower values are better). This ratio is at least 1, and a value close to 1 indicates a well-suited heuristic which achieves maximum possible communication-computation overlap.

Figures 7 and 9 show the distribution of the performance of each heuristic for different memory capacities, where plots are categorized by memory capacities. For each memory capacity and each heuristic, the box on the plot displays the median, first and last quartile, and the whiskers indicate minimum and maximum values, with outliers are shown by black dots.

6.1 HF Performance

As indicated above, HF tasks operate on less heterogeneous tiles, this is also noticeable in Figure 7. All heuristics depict similar behavior for minimum memory capacity m_c and increasing the memory capacity slightly does not change the performance of all heuristics. As memory capacity is increased further, dynamic variants of heuristics start performing better. For the moderate memory capacities (close to $2m_c$), static order with dynamic correction variants outperform others. *GG* heuristic does not achieve good performance, because its task sequence is obtained considering no extra memory is available, but is then applied in a different scenario where memory is limited. Surprisingly, the *BP* heuristic which considers only memory constraint obtains good performance for a static heuristic, but is outperformed by more dynamic approaches.

Figure 8 shows the performance comparison of the best variant in each category, in addition to the *order of submission* (*OS*) strategy which orders tasks in the (arbitrary) sequence in which they are given. Static strategies are expected to perform better when there is not any memory capacity restriction, and indeed this plot shows that static strategies face capacity bottleneck and underperform with limited memory. Dynamic strategies achieve slightly better performance with limited memory capacity, but when memory capacity is larger, static order with dynamic correction strategies outperform all others.

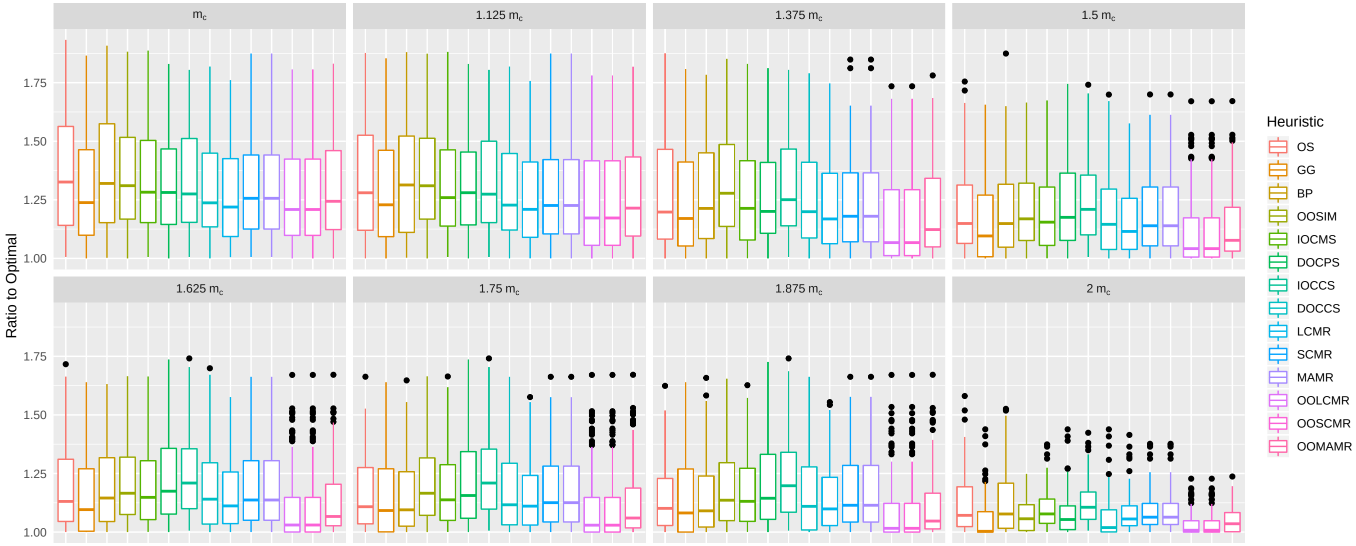


Figure 9: Comparison of different heuristics for CCSD with $m_c = 1.8GB$.

6.2 CCSD Performance

The CCSD application operates on tasks of different sizes, hence different heuristics exhibit distinct behaviors even at minimum memory capacity m_c. Heterogeneity favors dynamic strategies, as can be seen by the fact that both dynamic and static order with dynamic correction based strategies perform better than static based strategies. Similar to HF, static order with dynamic correction based strategies outperform others as memory capacity becomes moderate.

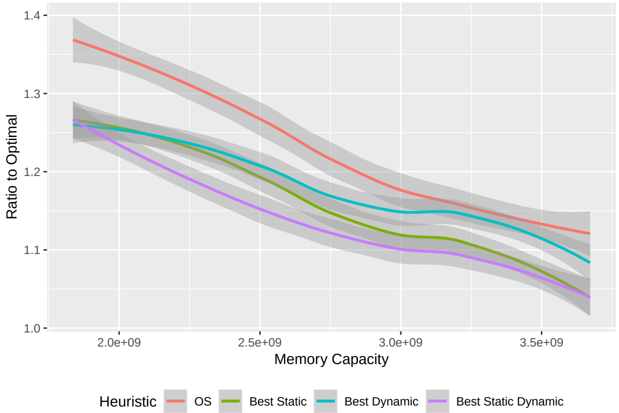


Figure 10: Best variants of all categories for CCSD.

Figure 10 shows that best variants of dynamic and static order with dynamic correction strategies achieve similar performance at minimum memory capacity m_c. But as memory capacity increases, heterogeneity allows static order with dynamic correction based strategies to take advantage of static knowledge to get maximum overlap and dynamic correction to select another task in case of memory capacity limitation. Static strategies also start performing better at the end, which indicates that this application has potential for significant communication-computation overlap and pure dynamic strategies are unable to take this information into account while making scheduling decisions.

6.3 Scheduling in Batches

In most applications, the runtime scheduler may only observe a limited batch of independent tasks. Therefore we organize tasks of each trace file in the batches of 100 (the last batch may have less than 100 tasks). We apply each heuristic on the batches in sequential order. Figure 11 shows the performance of the best variant of each category for both applications. The plots exhibit behavior similar to Figures 8 and 10: static order with dynamic correction variants attain maximum communication-computation overlap and outperform other heuristics.

7 CONCLUSION AND PERSPECTIVES

In this paper, we consider the problem of deciding the order of data transfers between two memory nodes such that overlap of communications and computations is maximized. With Exascale computing, applications face bottlenecks due to communications. Hence, it is extremely important to achieve the maximum communication-computation overlap in order to exploit the full potential of the system. We show that the problem of determining the optimal order of data transfers is NP-complete. We proposed several data transfer heuristics and evaluated them on two molecular chemistry kernels, HF and CCSD. Our results show that some of our heuristics achieve significant overlap and their makespans are very close to the lower bound. We plan to evaluate our strategies on different applications coming from multiple domains. We also plan to study the behavior of our strategies in the context of overlapping CPU-GPU communications with computations. A runtime system aiming to expose different heuristics to maximize the communication-computation overlap at the developer level and automatically select the best one is currently underway.

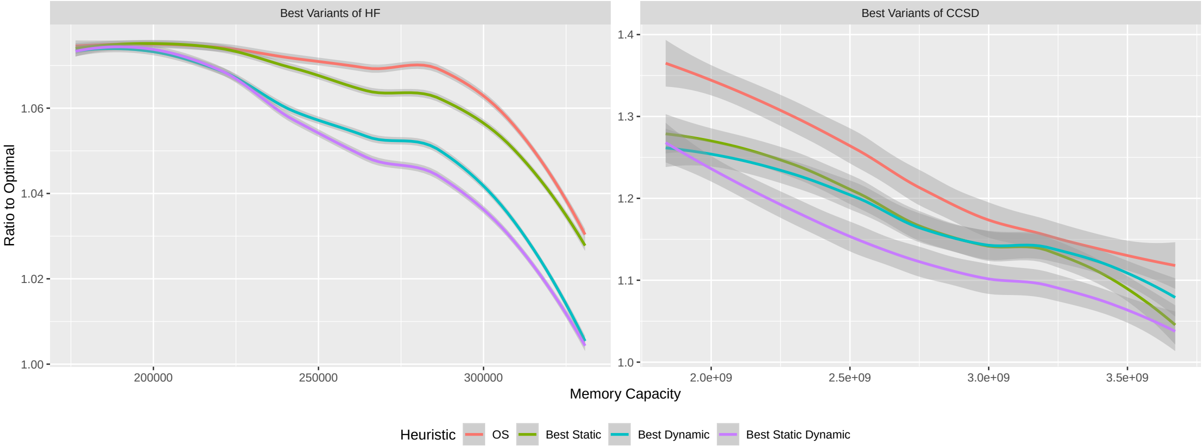


Figure 11: Best variants of all categories where heuristics are applied in the batches of 100 tasks.

ACKNOWLEDGMENTS

We are grateful to Ajay Panyala for his help in installation of NWChem on Cascade machine and providing internal details about the package and its inputs. This work was supported in part by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under award 63823.

REFERENCES

[1] 2010. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications* 181, 9 (2010), 1477 – 1489. https://doi.org/10.1016/j.cpc.2010.04.018

[2] 2014. Top Ten Exascale Research Challenges. ASCAC committee report, URL: https://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf.

[3] 2019. Commincation Scheduling. https://github.com/surakuma/communication-scheuling.

[4] 2019. Computing: Cascade. https://www.emsl.pnl.gov/emslweb/10.25582/inst.34218

[5] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, and Suraj Kumar. 2016. Are Static Schedules so Bad? A Case Study on Cholesky Factorization. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. 1021–1030.

[6] Emmanuel Agullo, Berenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. 2016. Task-based FMM for heterogeneous architectures. *Concurrency and Computation: Practice and Experience* 28, 9 (2016), 2608–2629. https://doi.org/10.1002/cpe.3723

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (Feb. 2011), 187–198. Issue 2. https://doi.org/10.1002/cpe.1631

[8] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 66, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389086

[9] Raphael Bleuse, Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, and Denis Trystram. 2015. Scheduling independent tasks on multi-cores with GPU accelerators. *Concurrency and Computation: Practice and Experience* 27, 6 (2015), 1625–1638. https://doi.org/10.1002/cpe.3359

[10] George Bosilca, Aurélien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack Dongarra. 2013. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *Computing in Science and Engineering* (2013). https://doi.org/10.1109/MCSE.2013.98

[11] Peter Brucker and Sigrid Knust. [n. d.]. Complexity results for scheduling problems. Web document, URL: http://www2.informatik.uni-osnabrueck.de/knust/class/. Accessed: 2019-04-15.

[12] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters* 21, 2 (2011), 173–193. https://doi.org/10.1142/S0129626411000151

[13] M. R. Garey and D. S. Johnson. 1979. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company.

[14] M. R. Garey, D. S. Johnson, and Ravi Sethi. 1976. The Complexity of Flowshop and Jobshop Scheduling. *Math. Oper. Res.* 1, 2 (May 1976), 117–129. https://doi.org/10.1287/moor.1.2.117

[15] P. C. Gilmore and R. E. Gomory. 1964. Sequencing a One State-Variable Machine: A Solvable Case of the Traveling Salesman Problem. *Operations Research* 12, 5 (1964), 655–679. https://doi.org/10.1287/opre.12.5.655

[16] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard. 2010. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In *Euro-Par (2)*. 235–246.

[17] S. M. Johnson. 1954. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly* 1, 1 (1954), 61–68. https://doi.org/10.1002/nav.3800010110

[18] Suraj Kumar, Lionel Eyraud-Dubois, and Sriram Krishnamoorthy. 2019. Performance Models for Data Transfers: A Case Study with Molecular Chemistry Kernels. *CoRR* abs/1904.06825 (2019). http://arxiv.org/abs/1904.06825

[19] L. Marchal, H. Nagy, B. Simon, and F. Vivien. 2018. Parallel Scheduling of DAGs under Memory Constraints. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 204–213. https://doi.org/10.1109/IPDPS.2018.00030

[20] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. 1996. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing* 10, 2 (01 Jun 1996), 169–189. https://doi.org/10.1007/BF00130708

[21] Christos H. Papadimitriou and Paris C. Kanellakis. 1980. Flowshop Scheduling with Limited Temporary Storage. *J. ACM* 27, 3 (July 1980), 533–549. https://doi.org/10.1145/322203.322213

[22] Maria Predari. 2016. *Load Balancing for Parallel Coupled Simulations*. Theses. Université de Bordeaux, LaBRI ; Inria Bordeaux Sud-Ouest. https://hal.inria.fr/tel-01518956

[23] D. Sbîrlea, Z. Budimlić, and V. Sarkar. 2014. Bounded memory scheduling of dynamic task graphs. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 343–355. https://doi.org/10.1145/2628071.2628090

[24] Ravi Sethi and J. D. Ullman. 1970. The Generation of Optimal Code for Arithmetic Expressions. *J. ACM* 17, 4 (Oct. 1970), 715–728. https://doi.org/10.1145/321607.321620

[25] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. 2014. Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-core Architectures. In *Euro-Par 2014 Parallel Processing*, Fernando Silva, Inês Dutra, and Vítor Santos Costa (Eds.). Springer International Publishing, Cham, 50–62.

[26] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. 2002. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. Parallel Distrib. Syst.* 13, 3 (March 2002), 260–274. https://doi.org/10.1109/71.993206

[27] A. YarKhan, J. Kurzak, and J. Dongarra. 2011. *QUARK Users' Guide: QUeueing And Runtime for Kernels*. UTK ICL.

[28] Kathy Yelick. 2016. Avoiding, Hiding and Managing Communication at the Exasacle. https://people.eecs.berkeley.edu/~yelick/talks/exascale/Communication-Yelick-China16.pdf