Locality-Aware Scheduling Of Independent Tasks For Runtime Systems

Maxime Gonthier, Loris Marchal, Samuel Thibault

Locality-Aware Scheduling Of Independent Tasks For Runtime Systems

Maxime GONTHIER - Samuel THIBAULT - Loris MARCHAL maxime.gonthier@ens-lyon.fr samuel.thibault@inria.fr - loris.marchal@ens-lyon.fr

Interest of our research

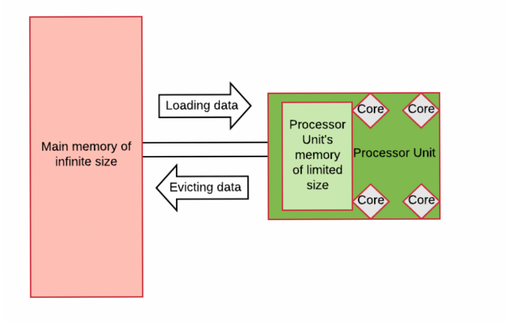


Figure 1: Model representing the link between main memory and local memory of the processor unit

- There are increasing demand for computing speed
- Scientific application use a huge quantity of data
- GPU can calculate quickly but their memory are limited
- The communication speed is limited between the GPU and the main memory which induce communication cost

The goal is to minimize the number of access to the main memory

Formalization of the problem

We use a set of independant tasks sharing input data. We want to produce a task order σ(t) and an eviction policy ν(t)

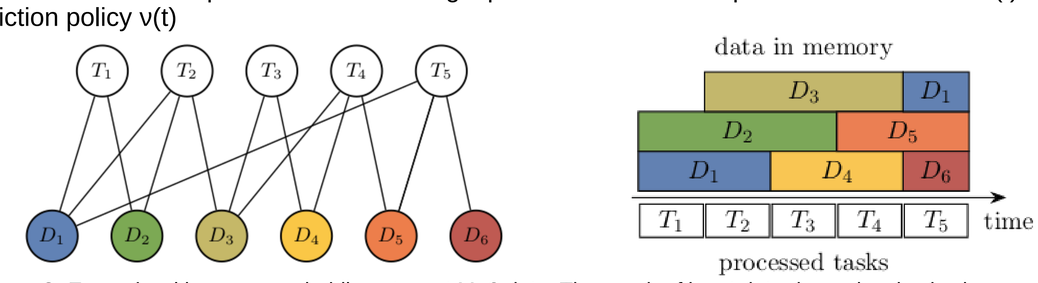


Figure 2: Example with a memory holding at most M=3 data. The graph of input data dependencies is shown on the left. The schedule on the right corresponds to processing the tasks in the natural order with the following eviction policy: ν(1) = ν(2) = ø, ν(3) = {1}, ν(4) = {2}, ν(5) = {3, 4}. This results in 7 loads.

For a given set of tasks T sharing data in D, what is the task order σ and the eviction policy ν that minimizes the number of loads?

Tested algorithms

From the state of the art

- **Reverse-Cuthill-McKee (RCM):**
  Permutes a sparse matrix into a minimal band matrix. Vertices sharing an edge are at most k edges away. Thus all tasks sharing a data are processed in the interval [t – k ; t + k].

Tasks are vertices and data share are edges

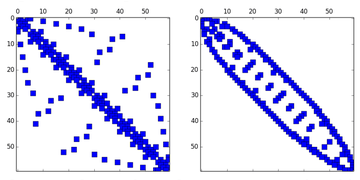


Figure 3: Transformation of a sparse matrix with RCM

- **Maximum Spanning Tree (MST):**
  The processing order is the order in which vertices are added to the spanning tree.

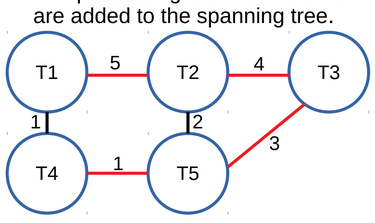


Figure 4: Exemple of a maximum spanning tree on a graph

From StarPU

- **Deque Model Data Aware Ready (DMDAR):**
  Schedules tasks where their completion times is expected to be minimal. Uses a ready strategy to favor tasks whose data has already been loaded into memory.

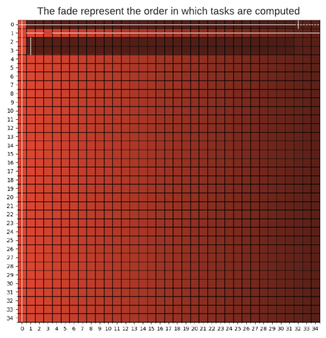


Figure 5: Visualisation of DMDAR's ordering on a 2D matrix multiplication

Ready is also used with Hierarchical Fair Packing

- **Eager**
  Process tasks in the order in which it receives them. Serves as a baseline.

Our contribution : Hierarchical Fair Packing

Initial intuition: Group tasks with common data into packages of maximum size M = GPU's memory. The goal is to minimize the number of packages. This will maximize data reuse

Hierarchical Fair Packing step by step:

0 Initially each task is an elementary package
1 Compute the number of common data on all pairs of packages
2 Consider packages P_i with the fewest tasks
3 Merge couples (P_i, P_j) which reach max number of common data such that Weight($P_i \cup P_j$) ≤ M
4 Repeat from step 1 until there are no more possible merge
5 Consider that M = ∞ and repeat from step 1 as long as there is more than 1 package

Package flipping: When merging two packages, we look at the extremities of each packages to put one after another extremity sharing the most data

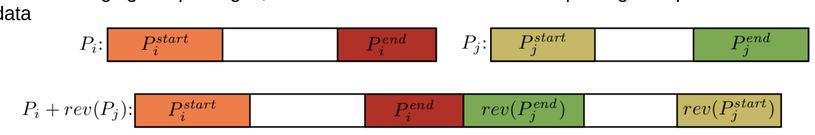


Figure 6: Package flipping. The pair of sub-packages (P_i^{end} , P_j^{end}) is the one with the most shared data. Thus P_j is reversed before merging packages.

Belady's eviction policy: Known optimal offline eviction policy. Always evict a data whose next use in the task order is the latest

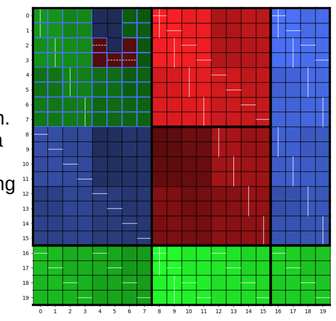


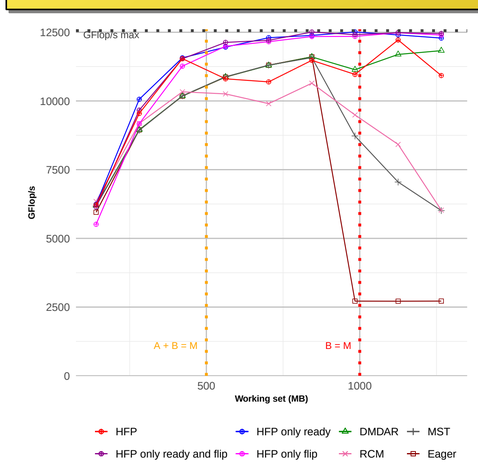
Figure 7: Visualisation of HFP's ordering on a 2D matrix multiplication. With 3 GPUs. Each color represent a GPU. The framed boxes are tasks stolen by a package for load balancing

Experimental settings

Experiments on a tesla V100 GPU as well as simulations on Simgrid using the same performance model.
- 6000 MB/s PCI bandwidth to represent the bandwidth share typically available for a given GPU in a multi-GPU platform
- GPU memory limited to 500 MB in order to better distinguish the performance of different strategies even on small datasets

Applications:
- Square 2D matrix multiplication: Each task is the multiplication of one block-row of A per one block-column of B
- Square 3D Matrix multiplication: Each task requires one tile of A, B and C
- Cholesky factorization

Experiments with a single GPU



The dotted horizontal black line represents the maximum Gflop/s that the GPU can achieve when processing elementary matrix product and is our asymptotic goal.
The red dotted vertical line denotes the situation when the GPU memory can fit exactly only one of the two input matrices, and the orange line denotes the situation when it can accommodate both input matrices.

- Pathological matrix size corresponding to a little less than the size of the GPU memory
- All HFP variants are close to the optimal
- The cost of Belady overcome it benefit

Figure 8: GFlop/s on 2D matrix multiplication using a Tesla V100 GPU

The black diagonal dottedline is the maximum number of transfers that can be done during the minimum time for computation, thus the hard limitation induced by the PCI bus bandwidth.

- HFP is 11% better than DMDAR
- HFP get better performance with more data transfers so it has a better ditribution of transfers over time

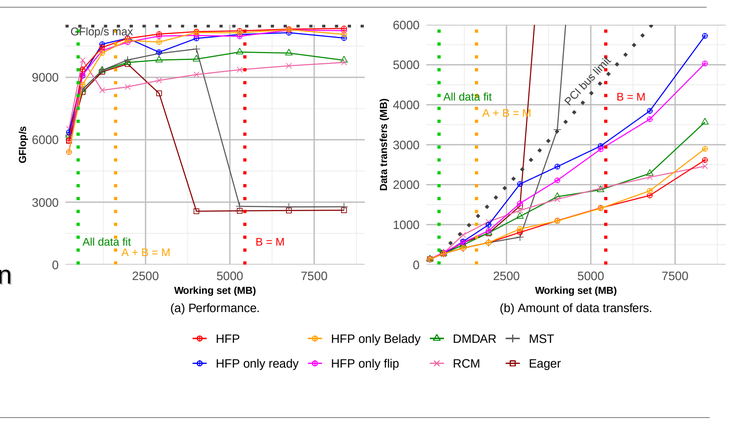
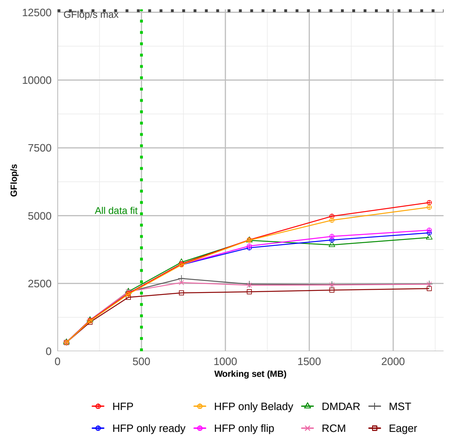


Figure 9: On 3D matrix multiplication on simulation: mesuring GFlop/s on the left and data transfers on the right



- Eager, RCM and MST get pathological performance once the whole matrix cannot fit in memory. They do not manage to reuse more than one tile between consecutive tasks, thus entailing a lot of tile reloads
- DMDAR has similar results with HFP. It takes advantage of the actual task submission order of the Cholesky algorithm
- HFP aims for data sharing as much as possible. It will thus introduce a lot of GEMM tasks at the beginning of the execution, and is thus impacted by the loading time. However HFP with Belady is the best heuristic here

Figure 10: Results on the task set of the Cholesky factorization on simulation

Experiments with multiple GPUs

HFP's packing allow us to easily adapt it to multi GPU cases. Load balancing was added in order to equilibrate the predicted computation time on each GPU.
We also added a partitionner, hMETIS, and used it in combination with HFP.

Results:
- HFP with load balance reaches good performances
- hMETIS coupled with HFP also achieve satisfying performance

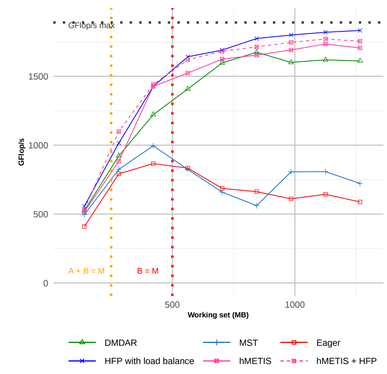


Figure 11: Gflop/s on a 2D matrix multiplication in simulation with 3 GPUs

Work in progress: Dynamic Outer

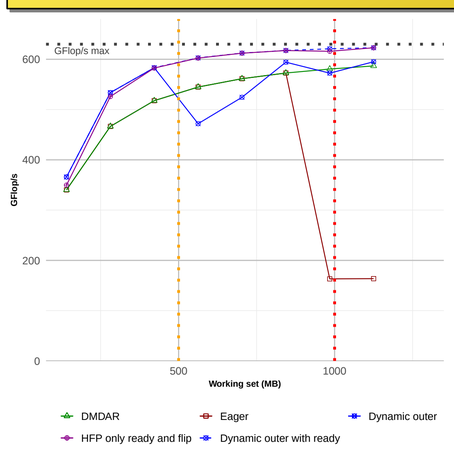


Figure 12: GFlop/s on a 2D matrix multiplication in simulation

Dynamic outer is a dynamic scheduler based on processing task whose data are already loaded. We adapted it to choose to load (resp. evict) data allowing to process the most task (resp. the least task)

Results:
- Dynamic outer achieve performance similar to HFP with a smallest complexity

Conclusion

- **Our contibutions:**
- We provided a formal model of the optimization problem of ordering independent tasks sharing input data in order to minimize the
- amount of data transfers
- We adapted two heuristic from the literature and designed a new heuristic based on gathering tasks with similar data into packages
- We implemented all three heuristics into the StarPU runtime
- We implemented a new scheduler : Dynamic Outer

Results:
- HFP allows on average a 8.3% (resp. 11% and 11.6 %) improvement for a 2D matrix multiplication (resp. 3D matrix multiplication and Cholesky).
- HFP is good in distributing data transfer over time to increase transfer/computation overlap
- HFP is well suited to multi GPU