Appendix A
Implementation details of our hybrid parallel framework

When a job transfer between different computing resources is required, the transfer is performed through main memory in the machine. If one of the source and the target computing resources is CPU, it is done by only one transaction between main memory and GPU's video memory (or global memory). For communication between two GPUs, jobs are moved to main memory first, and then are transferred to the target GPU's memory, since direct communication between GPUs is only supported under some specific conditions and GPU architectures [1].

In our implementation, we dedicate a single CPU thread, *GPU manager*, to all of GPU computing resources regardless of the number of GPUs. The GPU manager launches kernels and checks states (e.g., idle or busy) of GPU computing resources. The communication between main memory and GPU's global memory is controlled by the scheduler and the GPU manger. The GPU manager has its own communication queue, and the scheduler pushes a request on the communication queue when a data transfer between main memory and GPU memories is required. When the queue has requests, the GPU manager launches data transfer kernels asynchronously through a separate stream that is different from a stream for a job processing kernel [1]. As a result, we can overlap the data transfer and computations and it can hide the communication overhead. Instead of fetching a single job from the incoming queue of a computing resource and processing it, we dequeue multiple jobs and process them with a single invocation of the job processing routine. Specifically, we dequeue jobs that have the same job type. In each job processing routine, we divide the fetched jobs into multiple groups, and launch multiple kernels on different GPU streams for each group at once to overlap data transfer and computation. Thus, we can further minimize the data transfer overhead and optimize the utilization of GPUs.

Appendix B
An example workflow of our iterative LP solver

We show an example workflow of our scheduling method, which is based on an iterative LP solver (Sec. 4.3 in the main body of the paper).

Assume that we have only two job types (J_1, J_2) and have two times more jobs for J_1 than jobs of J_2 (i.e. n_1 = 200, n_2 = 100). Suppose also that we have three computing resources (R_1, R_2, R_3) that have identical capacities and show the same performance for both types of jobs, i.e. $T_{proc}(i,1)$ = $T_{proc}(i,2)$ = $0.01s$ where i is a computing resource index, but their setup costs are different:

$$\begin{aligned} R_1: &\quad T_{setup}(1,1) = 1s, &\quad T_{setup}(1,2) = 0s \\ R_2: &\quad T_{setup}(2,1) = 0s, &\quad T_{setup}(2,2) = 1s. \\ R_3: &\quad T_{setup}(3,1) = 0s, &\quad T_{setup}(3,2) = 4s. \end{aligned}$$

In the initial assignment step, the LP solver assumes that all the computing resources have setup costs for all the job types irrespective of the number of jobs. The LP solver, therefore, considers that the setup cost is same (i.e. one second) for two computing resource, R_1 and R_2, while the setup cost for R_3 is four seconds. Since the setup cost of R_3 (i.e. four seconds) is larger than the cost of processing all the jobs in other computing resources, the LP solver does not assign any jobs to R_3. Instead, the LP solver distributes the same number of jobs to R_1 and R_2 regardless of job types (Table 1).

Res.	n_{i1} (n_{i1}/n_1)	n_{i2} (n_{i2}/n_2)	Expected running time
R_1	100 (0.5)	50 (0.5)	2.5 sec.
R_2	100 (0.5)	50 (0.5)	2.5 sec.
R_3	0 (0.0)	0 (0.0)	0.0 sec.

TABLE 1: *An assignment result of the initial assignment step.*

However, since R_3 does not get any jobs, it actually does not take any setup cost and will be idle, even though other computing resource are busy with processing assigned jobs. Also, R_1 and R_2 will take more than two seconds, since each computing resource already takes one second for its setup cost and consumes one and half seconds for processing jobs. However, all the jobs can be done in two seconds, if we allocate all the jobs of J_2 to R_1 and J_1 to R_2 respectively.

In the first iteration of the refinement step, our iterative LP solver chooses R_3 to re-assign its jobs of J_2, since its job-to-resource ratio is zero and its setup cost (four seconds) is larger than others. We then re-run the LP solver under the constraint that no jobs of J_2 can be given to R_3, and an assignment result shown in Table 2 can be achieved. Even though R_3 does not waste its capacities, R_1 and R_2 processes jobs inefficiently, because of its incorrect calculations of setup costs. Since we achieve lower *makespan, L* (1.67 seconds) than the initial solution (2.5 seconds), we invoke one more iteration.

Res.	n_{i1} (n_{i1}/n_1)	n_{i2} (n_{i2}/n_2)	Expected running time
R_1	16 (0.08)	50 (0.5)	1.66 sec.
R_2	17 (0.085)	50 (0.5)	1.67 sec.
R_3	167 (0.685)	-	1.67 sec.

TABLE 2: *The assignment result of the first iteration.*

At the second iteration, the solver selects R_1 to re-distribute its jobs of J_1, and recompute its assignment result, which is shown in Table 3. In fact, it is the optimal solution of the example of our scheduling problem. In the next iteration, we will get even worse scheduling result and thus report the assignment result of the second iteration as the final assignment result.

Res.	n_{i1} (n_{i1}/n_1)	n_{i2} (n_{i2}/n_2)	Expected running time
R_1	-	100 (1.0)	1.0 sec.
R_2	100 (0.5)	0 (0.0)	1.0 sec.
R_3	100 (0.5)	-	1.0 sec.

TABLE 3: *The assignment result of the second iteration.*

References

[1] NVIDIA, "CUDA programming guide 4.0," 2012.