Achieving Data-Aware Load Balancing through Distributed Queues and Key/Value Stores

Ke Wang
Department of Computer Science
Illinois Institute of Technology
kwang22@hawk.iit.edu

Ioan Raicu
Department of Computer Science, Illinois Institute of Technology
Mathematics and Computer Science Division, Argonne National Laboratory
iraicu@cs.iit.edu

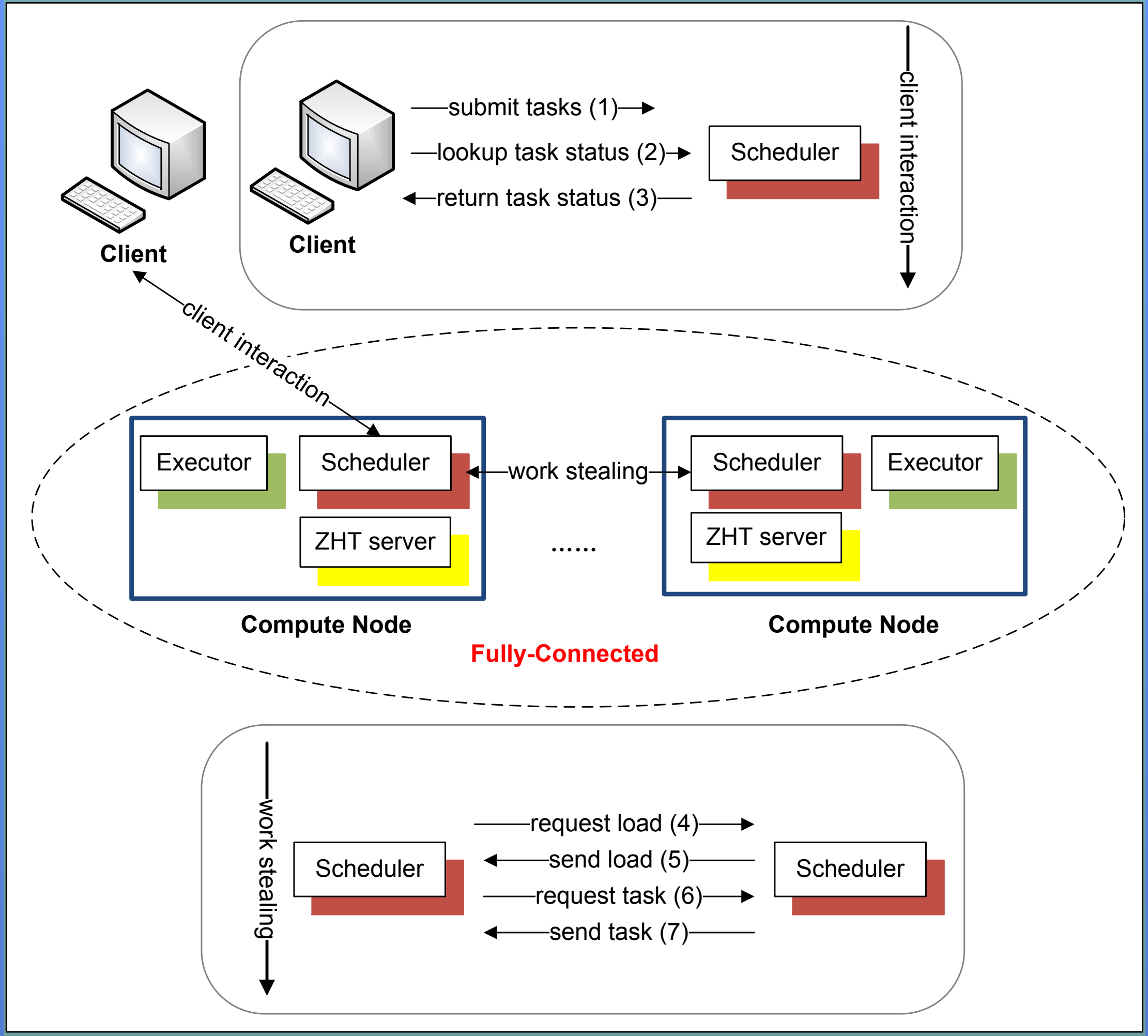
Abstract

Load balancing techniques (e.g. work stealing) are important to obtain the best performance for distributed task scheduling system. In work stealing, tasks are randomly migrated from heavy-loaded schedulers to idle ones. However, for data-intensive applications where tasks are dependent and task execution involves processing large amount of data, migrating tasks blindly would compromise the data-locality incurring significant data-transferring overhead. In this work, we propose a data-aware work stealing technique that combines key-value stores and distributed queues enabling it to achieve good load balancing, all while maximizing data-locality. We leverage a distributed key-value store, ZHT, as a meta-data service that stores task dependency and data-locality information. We implement the proposed technique in MATRIX, a distributed task execution fabric. We evaluate the work with all-pairs application structured as direct acyclic graph from biometrics, and compare with Falkon data-diffusion technique.

Contributions

1. Propose a data-aware work stealing technique that combines distributed queues and key-value stores
2. Apply a distributed key-value store as a meta-data service to store important data dependency and locality information.
3. Evaluate the proposed technique up to hundreds of nodes showing good performance using different applications under different scheduling policies.

MATRIX Architecture Overview



ZHT as Meta-Data Service

Data dependency and locality information of each task is represented as(Key, Value) pair: key is the task id, value is the TaskMetaData

```
typedef   TaskMetaData
{
    /* number of waiting parents */
    int    num_wait_parent;

    /* schedulers that run each parent task */
    vector<string>    parent_list;

    /* data object name produced by each parent */
    vector<string>    data_object;

    /* data object size (byte) produced by each parent */
    vector<long>    data_size;

    /* all data object size (byte) produced by all parents */
    long    all_data_size;

    /* children of this tasks */
    vector<string>    children;
}  TMD;
```

Distributed Queues

Wait Queue (WaitQ): holds tasks that are waiting for parents to complete

Dedicated Local Ready Queue (LReadyQ): holds ready tasks that can only be executed on local node

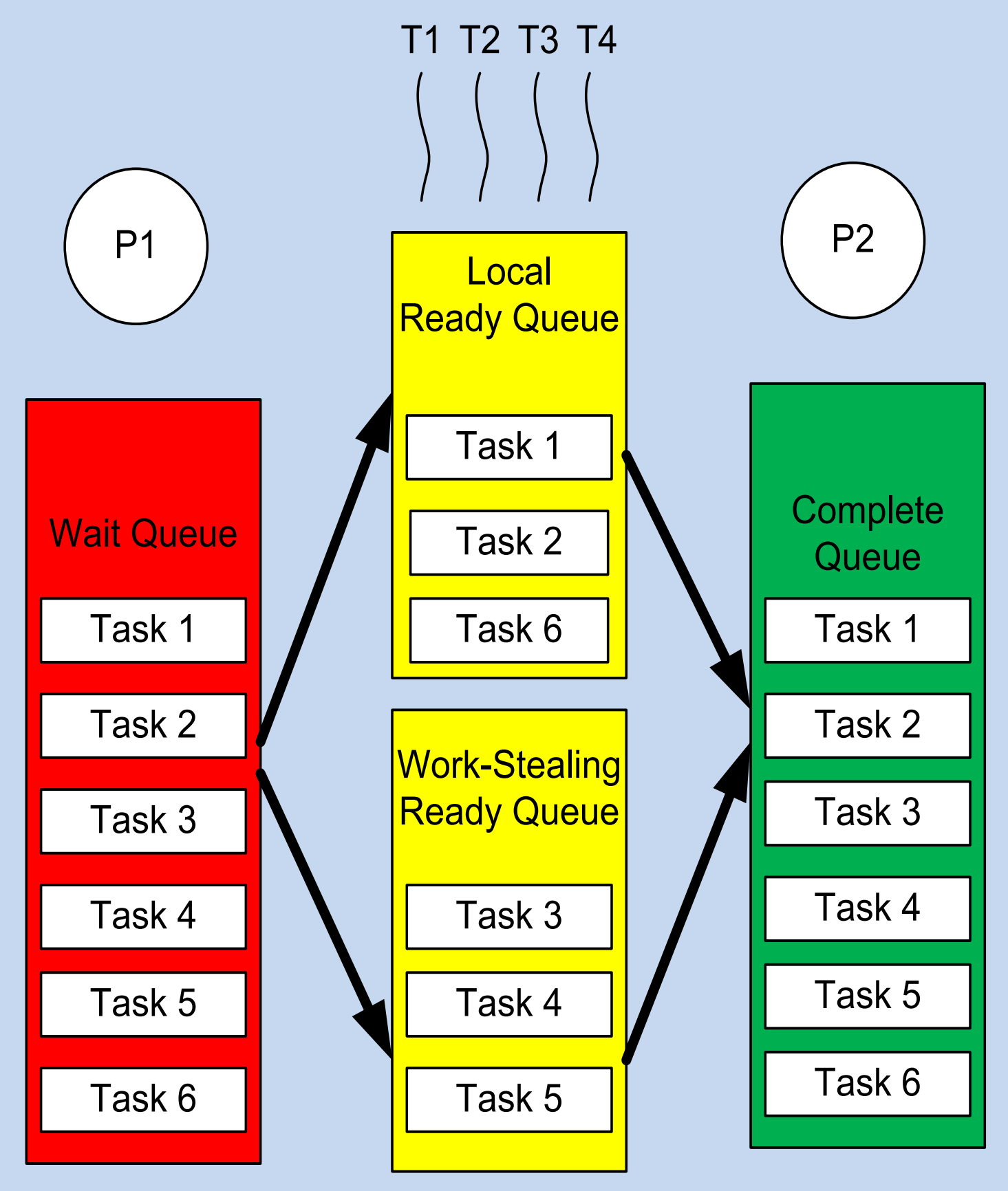
Shared Work-Stealing Ready Queue (SReadyQ): holds ready tasks that can be shared through work stealing

Complete Queue (CompleteQ): holds tasks that are completed

P1: a program that checks if a task is ready to run, and moves ready tasks to either ready queue according to the decision making algorithm

P2: a program that updates the task metadata for each child of a completed task

T1 to T4: executor has 4 (configurable) executing threads that executes tasks in the ready queues and move a task to complete queue when it is done



Decision Making Algorithm

ALGORITHM 1. Decision Making to Put a Task in the Right Ready Queue

```
Input: a ready task (task), TMD (tm), a threshold (t), current scheduler id (id),
LReadyQ, SReadyQ, estimated length of the task in second (est_task_length)
Output: void.
if (tm.all_data_size / est_task_length <= t) then
        SReadyQ.push(task);
else
        long max_data_size = tm.data_size.at(0);
        int max_data_scheduler_idx = 0;
        for each i in 1 to tm.data_size.size(); do
                if tm.data_size.at(i) > max_data_size; then
                        max_data_size = tm.data_size.at(i);
                        max_data_scheduler_idx = i;
                end
        end
        if (max_data_size / est_task_length <= t); then
                SReadyQ.push(task);
        else if tm.parent_list.at(max_data_scheduler_idx) == id; then
                LReadyQ.push(task);
        else
                send task to: tm.parent_list.at(max_data_scheduler_idx)
        end
end
return;
```

Four Scheduling Policies

(1) **MLB (Maximize Load Balancing):** considers only the load balancing, and all the ready tasks are put in the SReadyQ that are allowed to be migrated, no matter how big the data is.

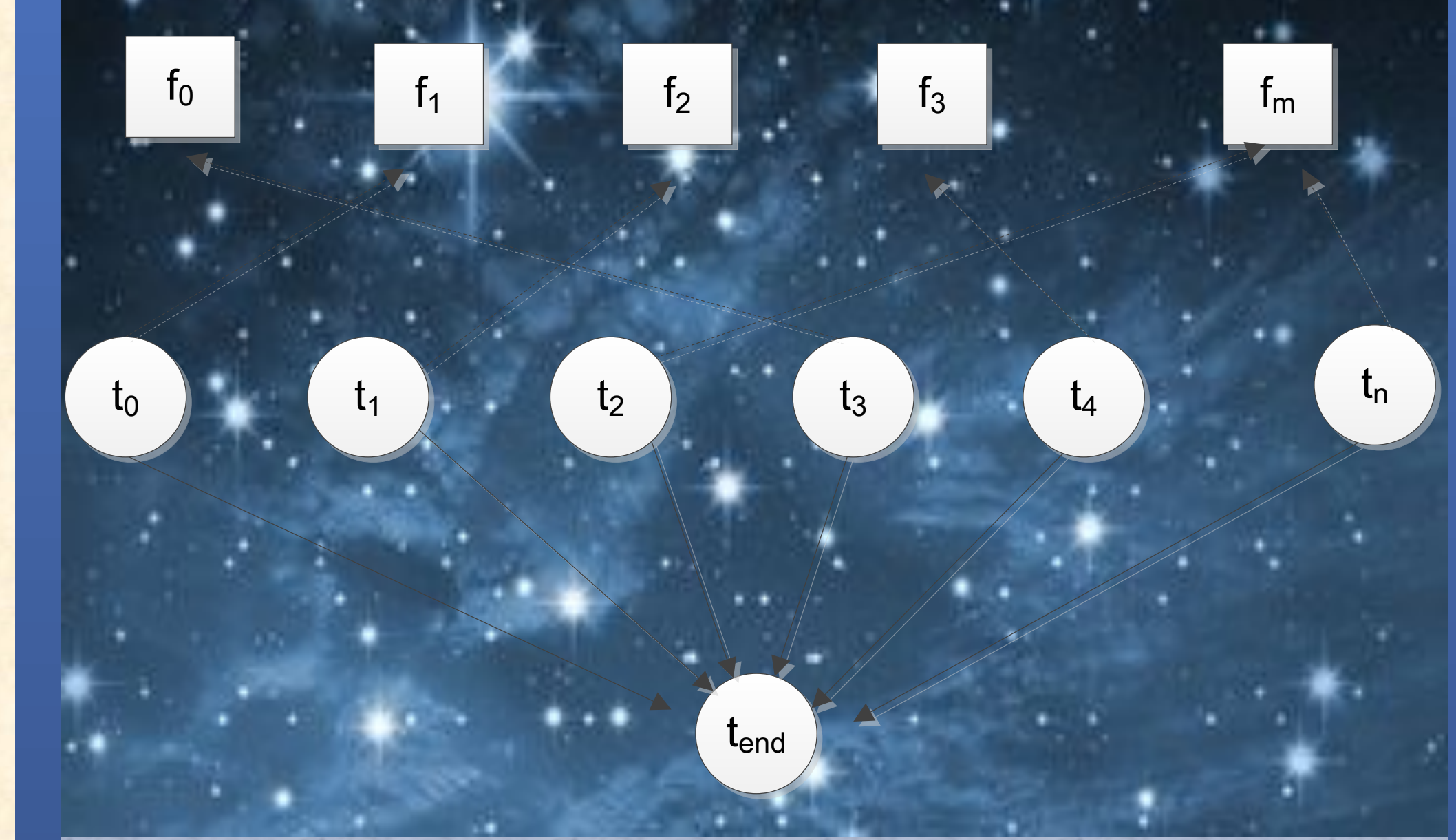
(2) **MDL (Maximize Data-Locality):** considers only data-locality, and all the ready tasks that require input data would be put in LReadyQ, no matter how big the data is.

(3) **RLDS (Rigid Load balancing and Data-locality Segregation):** ready tasks are put in either queue according to Algorithm 1. Once a task is put in the LReadyQ of a scheduler, it is confined to be executed locally.

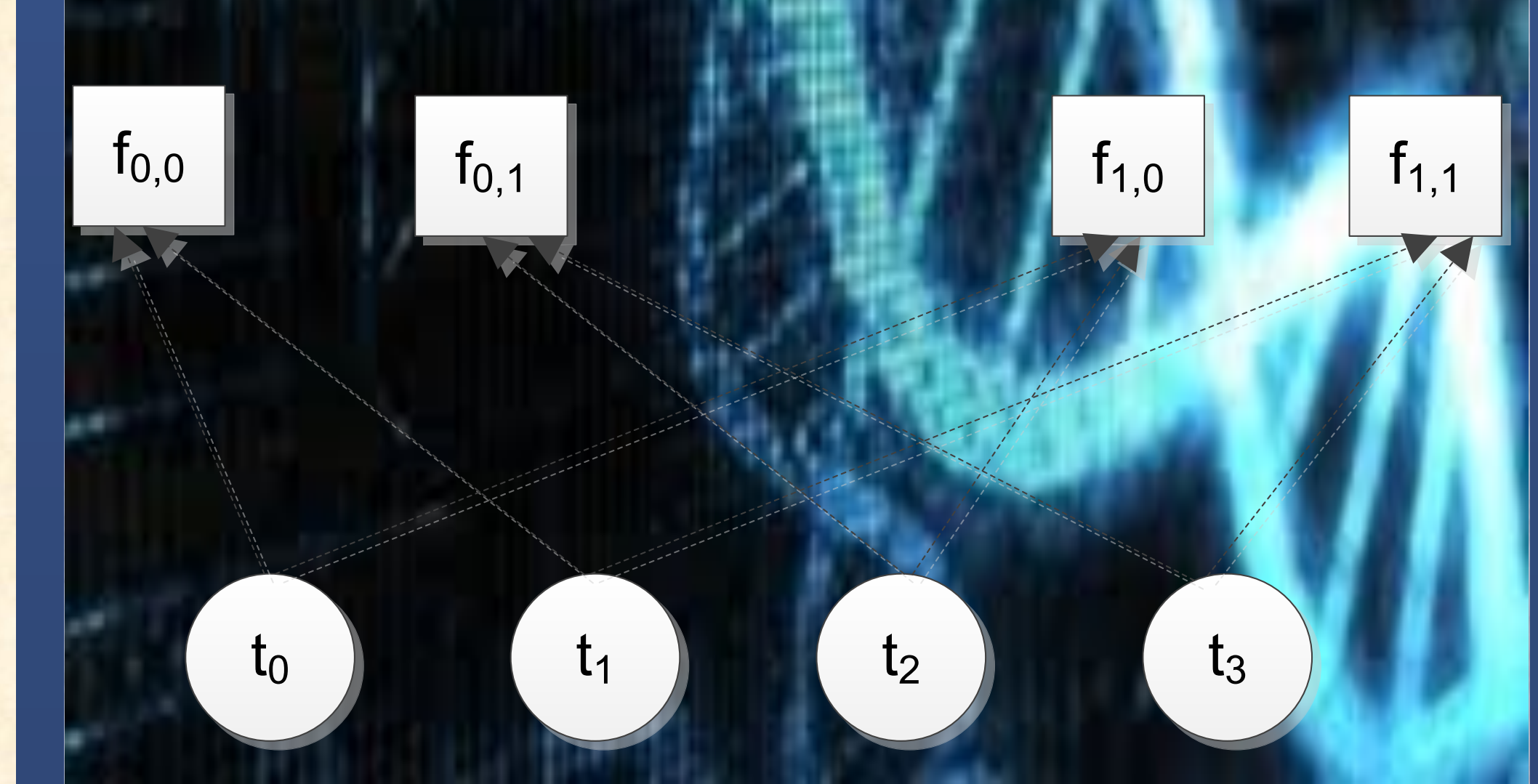
(4) **FLDS (Flexible Load balancing and Data-locality Segregation):** ready tasks are put in either queue according to Algorithm 1. A task in the LReadyQ of a scheduler may be moved to SReadyQ to avoid hotspot problem.

Applications

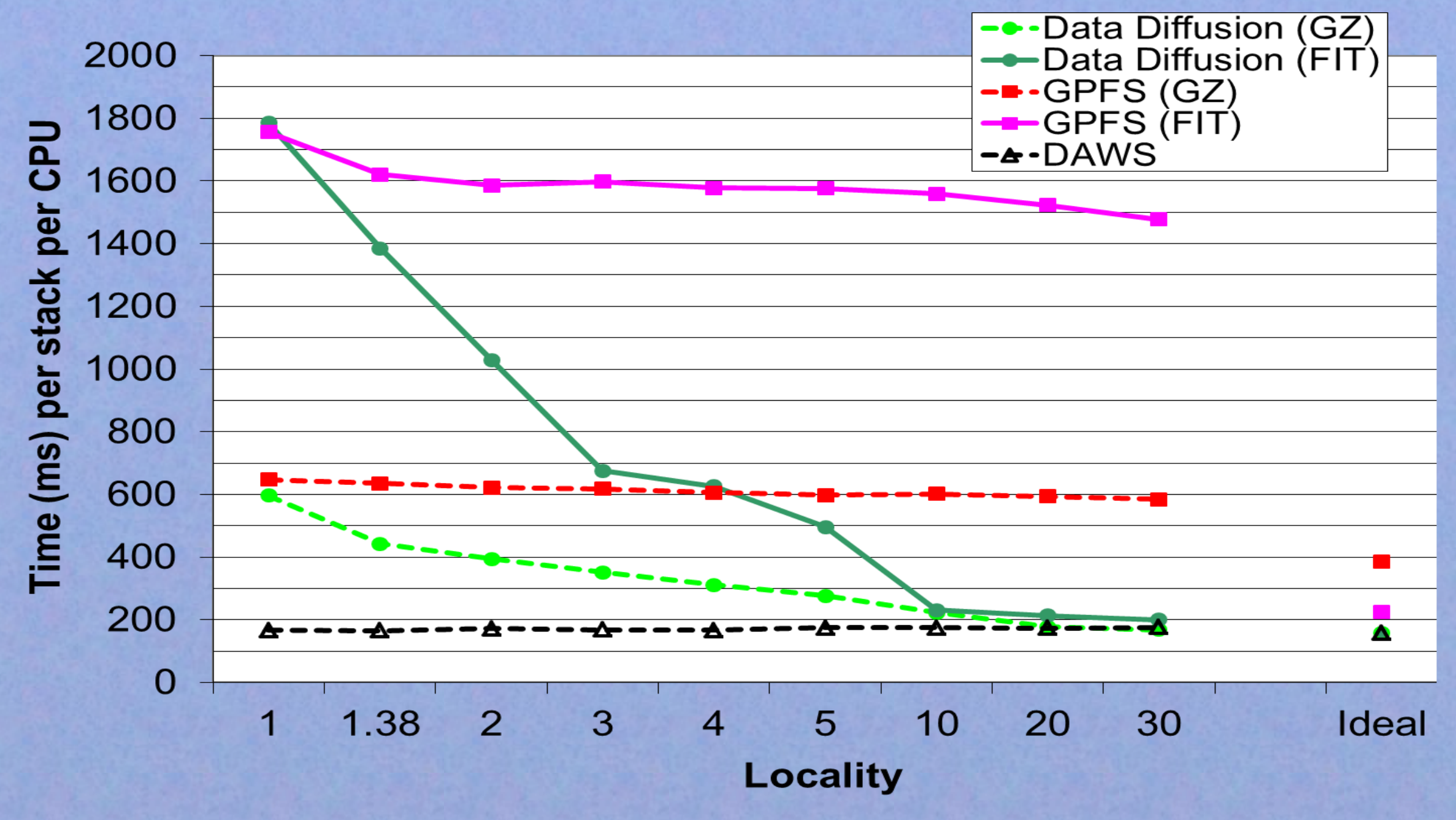
Image Stacking in Astronomy: conducts the "stacking" of image cutouts from different parts of the sky. The stacking procedure involves re-projecting each image to a common set of pixel planes, then co-adding many images to obtain a detectable signal that can measure their average brightness/shape.

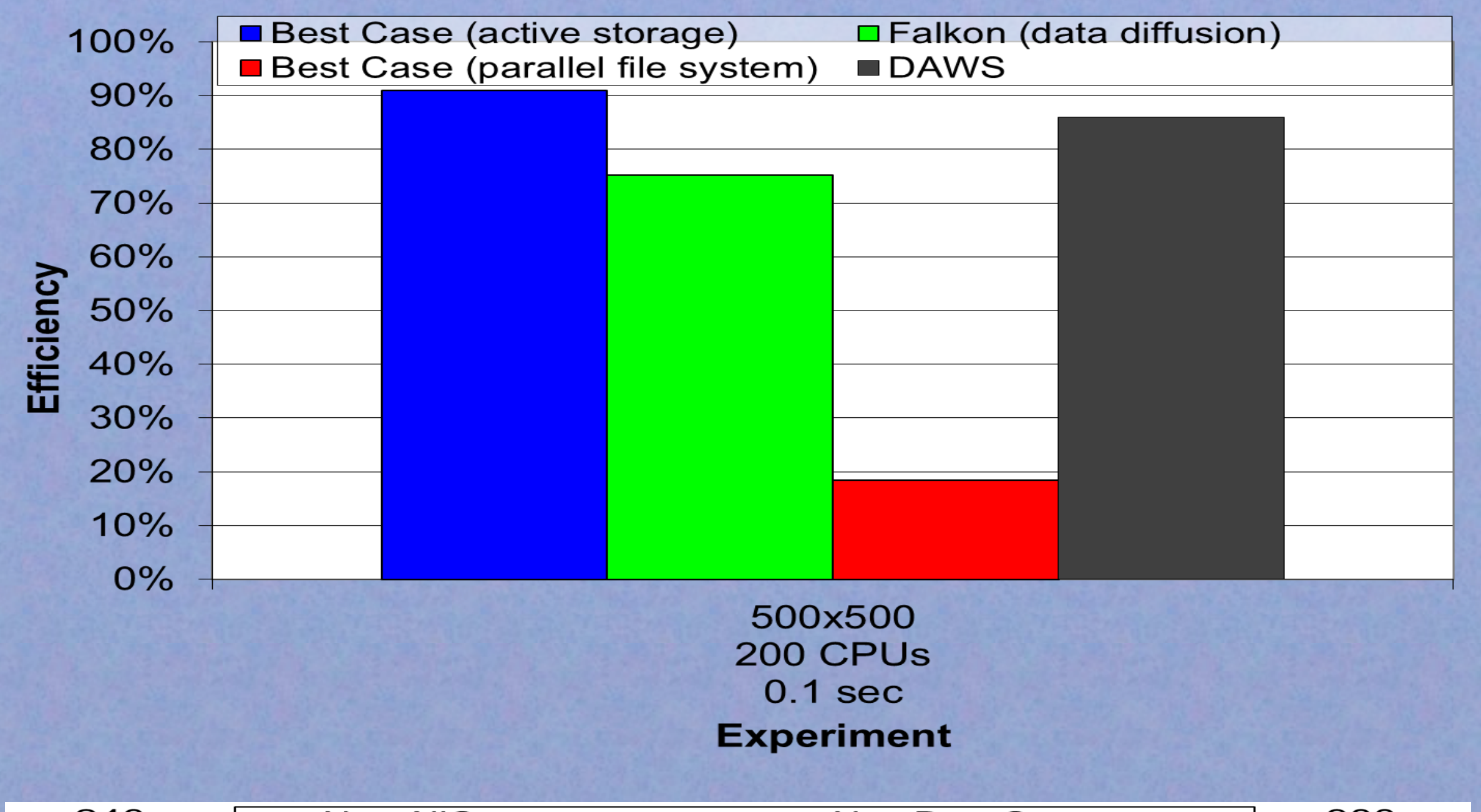


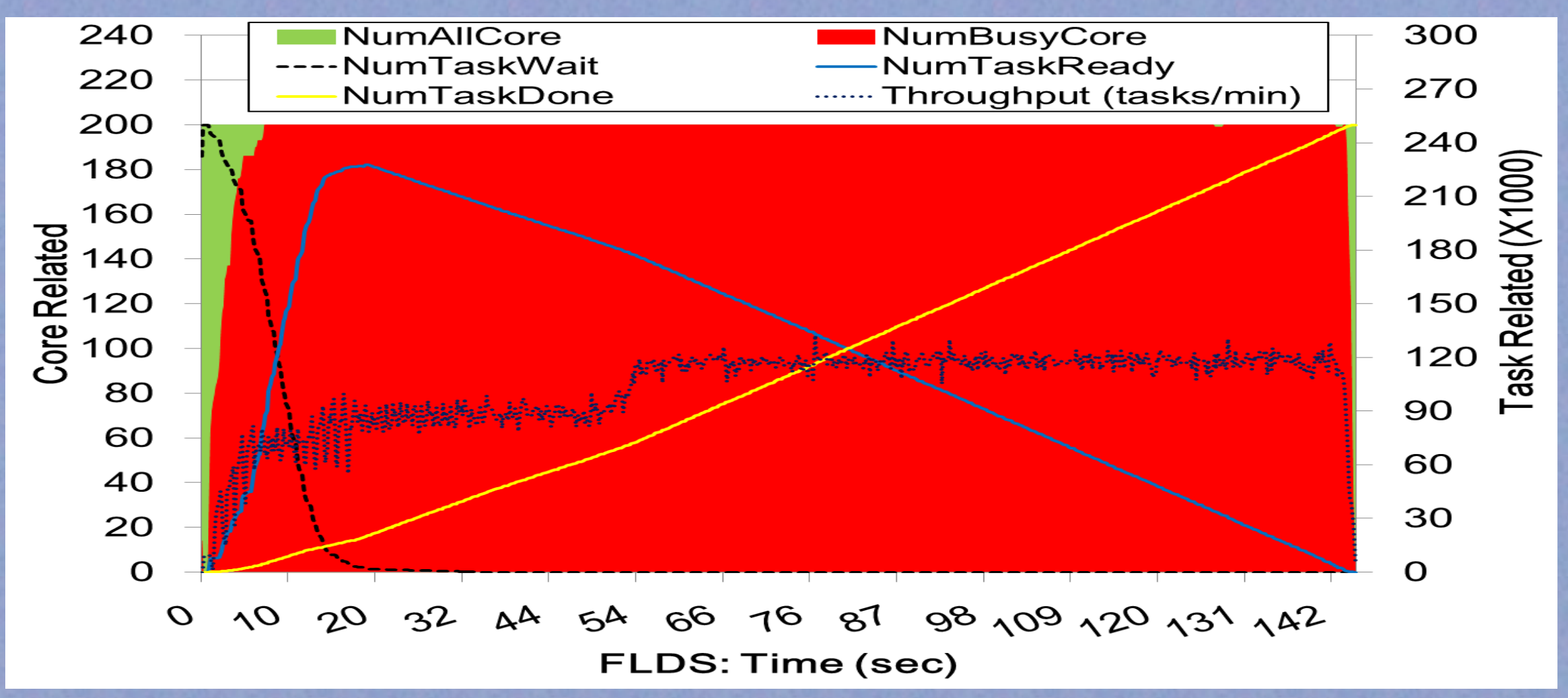
All-Pairs in Biometrics: All-Pairs is a common benchmark for data-intensive applications that describes the behavior of a new function on sets A and sets B. For example, in Biometrics, it is very important to find out the covariance of two sequences of gene codes. In this workload, all the tasks are independent, and each task execute for 100 ms to compare two 12MB files with one from each set



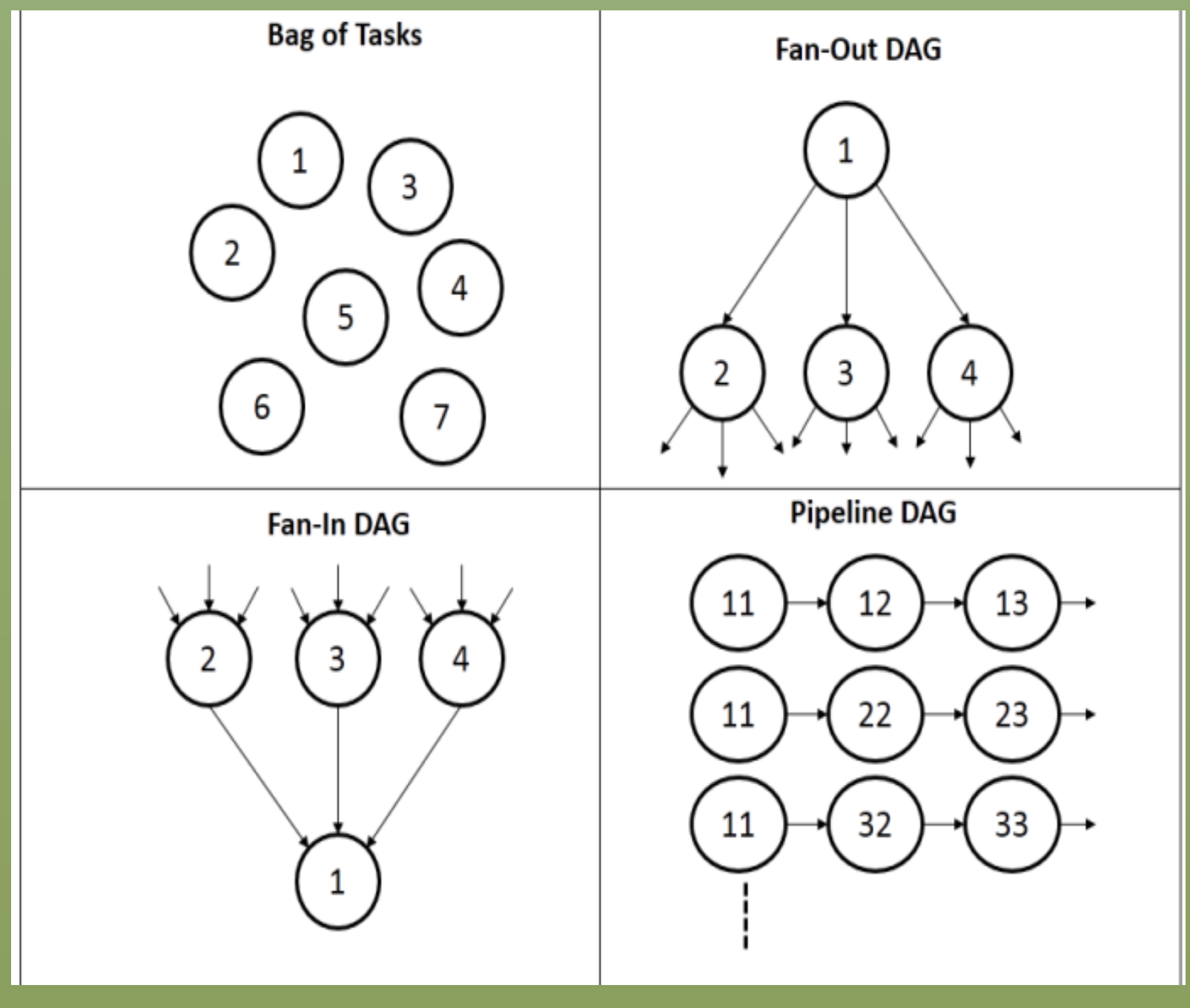
Application Results

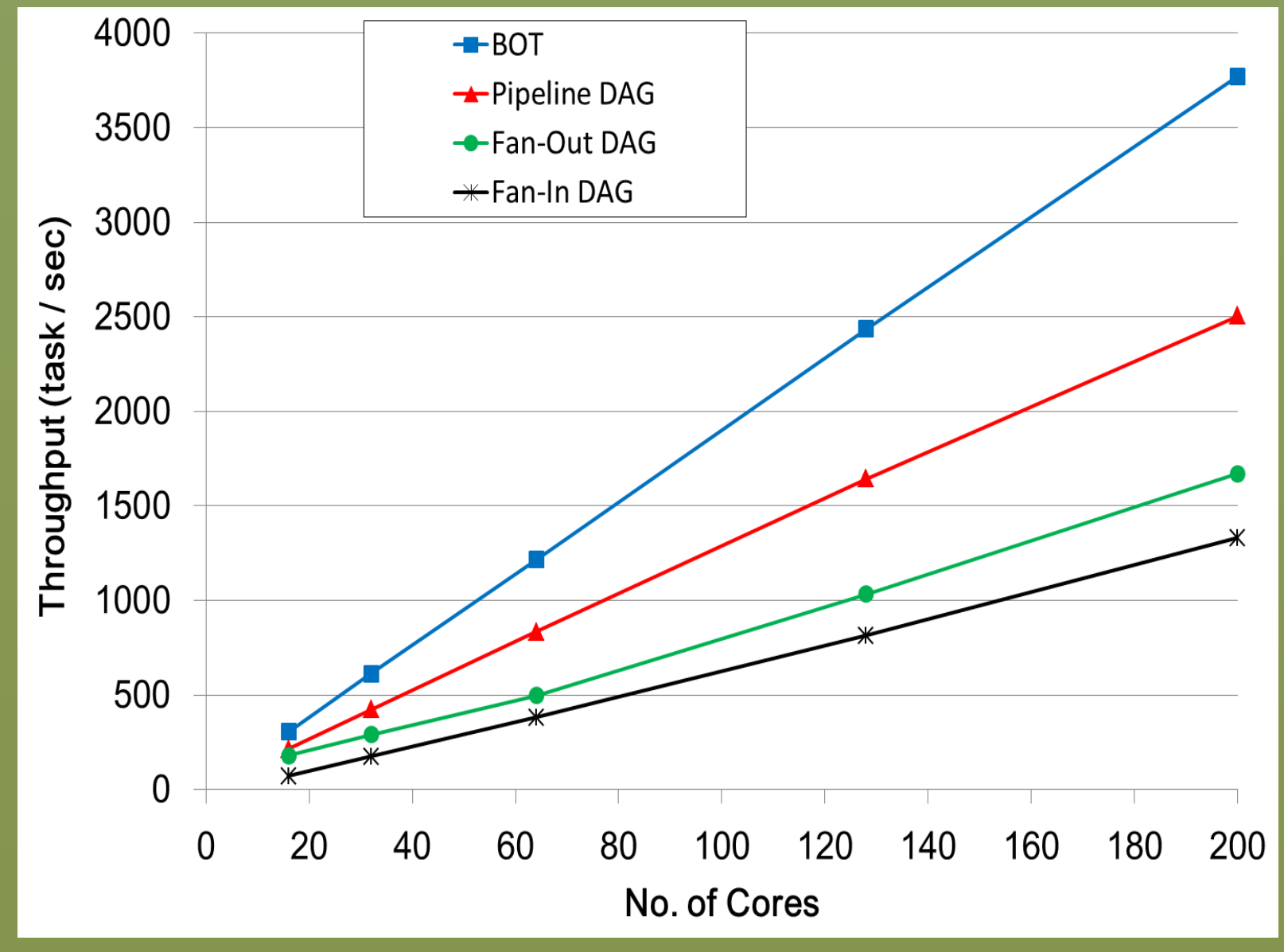






Micro-Benchmarks





Conclusions

Applications for extreme-scales are becoming more data-intensive and fine-grained in both task size and duration. Task schedulers for data-intensive applications at extreme-scales need to be scalable to deliver the highest system utilization, which poses urgent demands for both load balancing and data-aware scheduling. This work combined distributed load balancing with data-aware scheduling through a data-aware work stealing technique. We implement the technique in MATRIX, and apply a DKVS, as a transparent meta-data service. We evaluated our technique under four different scheduling policies with different workloads, and compared our technique with the Falkon data diffusion approach. Results showed that our technique is scalable to achieve both good load balancing and high location-hit rate. We have planned much work in the future, such as larger scales, HPC support, workflow integration, and MapReduce framework support.

References

[1] K. Wang, A. Kulkarni, M. Lang, D. Arnold, I. Raicu. "Using Simulation to Explore Distributed Key-Value Stores for Extreme-Scale Systems Services," IEEE/ACM Supercomputing/SC 2013.
[2] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, I. Raicu. "ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table", IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2013.
[3] K. Wang, A. Rajendran, I. Raicu. "MATRIX: MAny-Task computing execution fabRIc at eXascale," tech report, IIT, 2013.
[4] I. Raicu, Y. Zhao, I. Foster, A. Szalay. "Accelerating Large-scale Data Exploration through Data Diffusion", International Workshop on Data-Aware Distributed Computing 2008, co-locate with ACM/IEEE HPDC 2008.
[5] I. Raicu, I. Foster, Y. Zhao, P. Little, C. Moretti, A. Chaudhary, D. Thain. "The Quest for Scalable Support of Data Intensive Workloads in Distributed Systems", ACM HPDC 2009.