Transparent Resource Elasticity for Task-Based Cluster Environments with Work Stealing

P2S2 – ICPP

Jonas Posner and Claudia Fohry
jonas.posner@uni-kassel.de

U N I K A S S E L
V E R S I T Ä T
University of Kassel, Germany
Programming Languages/Methodologies

August 9th, 2021

Motivation

- Today's jobs on supercomputers typically retain the same set of resources throughout their lifetime, called *rigid*
- Rigidity limits the flexibility of job schedulers
- Adaptive resource management enables on-demand resource changes at runtime
- Benefits include highly improved global throughput and decreased energy consumption

Introduction

- Adaptivity must be backed by at least three major layers:
  - Job scheduler
  - Programming system
  - Algorithm / Application
- Resource change requests may come from the application itself or from the job scheduler
- We address the second case, called *malleability*
- **Malleable programs require additional programmer effort and are rarely used in practice so far**

Contributions

- A novel malleability scheme for a work-stealing task environment that enables a transparent adaptation to the addition or release of multiple nodes
  - *Programmers do not need to modify their programs*
- Experimental overhead evaluation of resizing operations
  - Formulas to estimate overhead-free running times
- Simulating execution of job sets on supercomputers to quantify the impact of malleable jobs
  - A heuristic to determine malleability job parameters

Asynchronous Many-Task (AMT)

- AMT programs are naturally structured into many small execution units, called *tasks*
- A runtime systems maps these tasks to computing resources (e.g., processes, threads), called *workers*
- Since AMT resource management is transparent and performed at runtime, AMT is well suited for providing malleability

Multi-Worker Global Load Balancing Library[1]

- Combines *work sharing* for *intra*-process load balancing with *work stealing* for *inter*-process load balancing
- On each node runs one process with multiple workers
- Each *worker* maintains a private local task queue
- Each *process* maintains additional shared task queues
- Work sharing: Workers with surplus tasks provide them to others by inserting them into a shared queue

[1] P. Finnerty et al., Self-Adjusting Task Granularity for Global Load Balancer Library on Clusters of Many-Core Processors, PMAM, 2020.

Multi-Worker Global Load Balancing

- Work stealing: When a whole process runs empty, it tries to steal tasks from other processes
  - Processes are arranged in a *lifeline-graph* for victim selection and termination detection
- Dynamic independent tasks:
  - Tasks are free of side effects
  - Tasks generate results and possibly new tasks
  - Task results are locally accumulated
  - The final result is reduced over all worker results
  - Well suited for tree-based algorithms solving search, optimization, and approximation problems

Malleability Scheme

- We implemented the scheme by extending MultiGLB
- The scheme is performed concurrently to task processing
- Process 0 takes some responsibilities and can not be released
- Shrinking:
  - All steals related to the processes to be released are stopped (including recalculation of lifeline-graph)
  - All processes to be released stop task processing, and send their tasks and results to co-processes
- Expanding:
  - Recalculation of the lifeline graph whereby new processes automatically will receive tasks via work stealing

Overhead Analysis

- We experimentally quantified the costs for adding and releasing processes on-the-fly
- We used up to 128 nodes, each with 40 cores
- Two synthetic benchmarks (called `StatSyn` and `DynSyn`)
    - Perform placeholder computations
    - Are configurable for smooth weak scaling
    - Allow an accurate analysis with derived formulas

Overhead Analysis

- We executed the benchmarks in three ways:
  - *Rigid:* no resize operation is triggered
  - *Shrinking:* after half of the running time one shrink operation is triggered to release half of the processes
  - *Expanding:* after half of the running time one expand operation is triggered to double the processes

Estimation of Overhead-Free Running Times

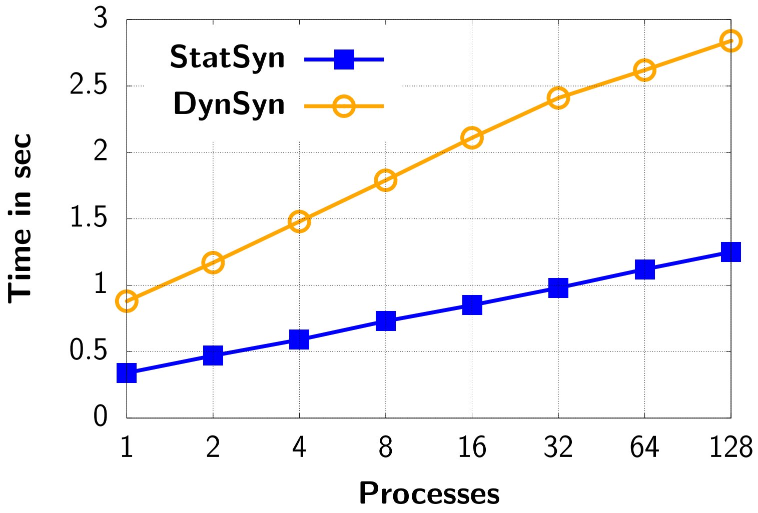
- **Shrinking**: the program is initialized with P_i processes and P_s processes are released at T_{op}:

$$T_{Est}(P_i - P_s) = T_{op} + \frac{\left(\widehat{T}(P_i) - \frac{T_{op}}{1+L(P_i)}\right) \cdot P_i \cdot (1 + L(P_i - P_s))}{P_i - P_s} \tag{1}$$

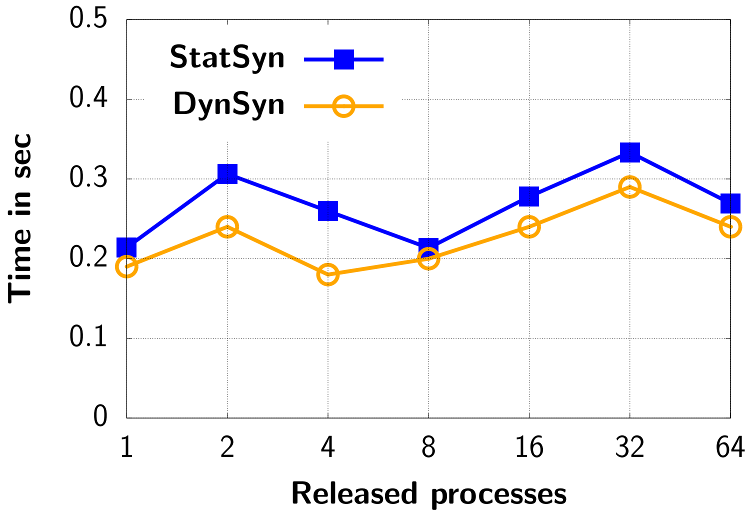
- **Growing**: the program is initialized with P_i processes and P_e processes are added at T_{op}:

$$T_{Est}(P_i + P_e) = T_{op} + \frac{\left(\widehat{T}(P_i) - \frac{T_{op}}{1+L(P_i)}\right) \cdot P_i \cdot (1 + L(P_i + P_e))}{P_i + P_e} \tag{2}$$

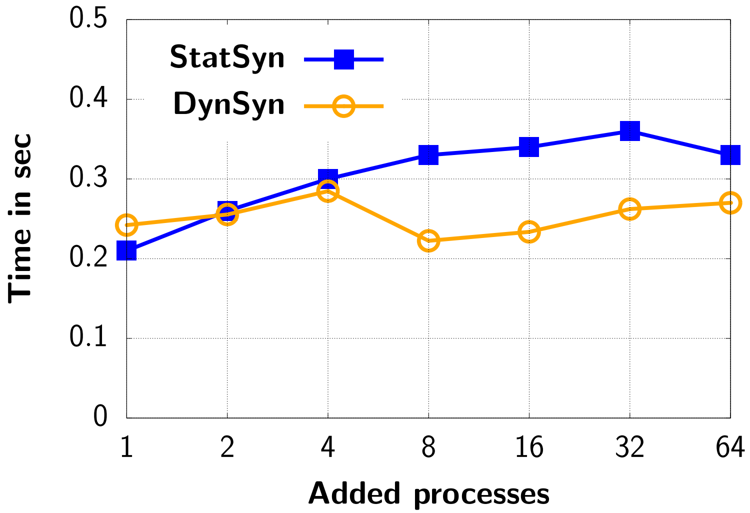
Costs of Load Balancing (Rigid Runs)



Costs of Shrinking



Costs of Expanding



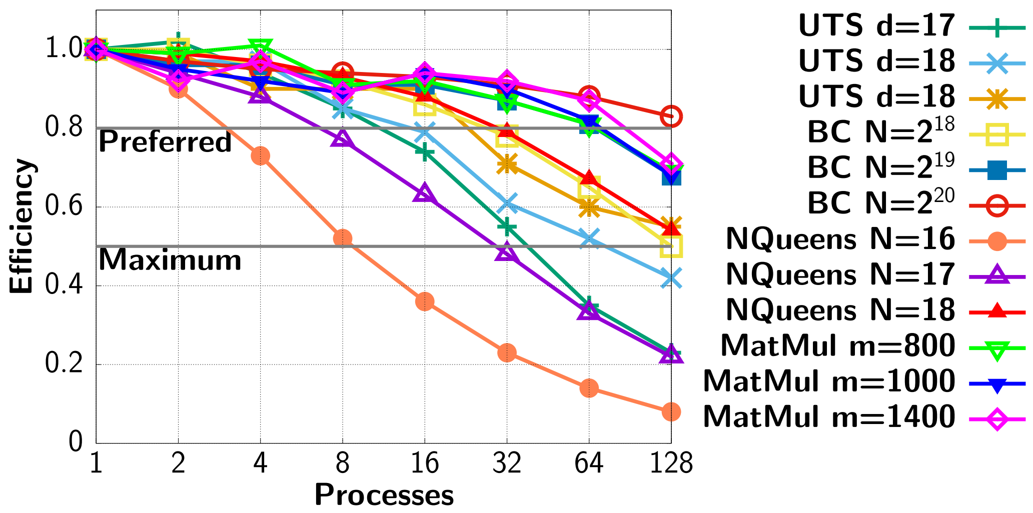
Simulation

- We quantified the impact of malleable workloads compared to rigid ones on supercomputers
- We simulated the execution of sets of independent parallel jobs
- The job sets are generated with benchmarks performance data
- Simulated supercomputer consists of 2048 nodes
- Simulator starts jobs sorted by submission time, coupled with back-filling adapted from Slurm

A Heuristic for Malleable Job Parameters

- Usually, only a certain range of nodes makes sense for applications from a performance point of view
- Our heuristic builds on the well-known concept of program $\textit{efficiency} = T(1)/(p \cdot T(p))$
- The *minimum* number of nodes is always 1
- The *preferred* number of nodes is defined as the largest p, for which the efficiency is ≥ 0.8
- The *maximum* number of nodes is defined as the largest p, for which the efficiency is ≥ 0.5

Program Efficiencies

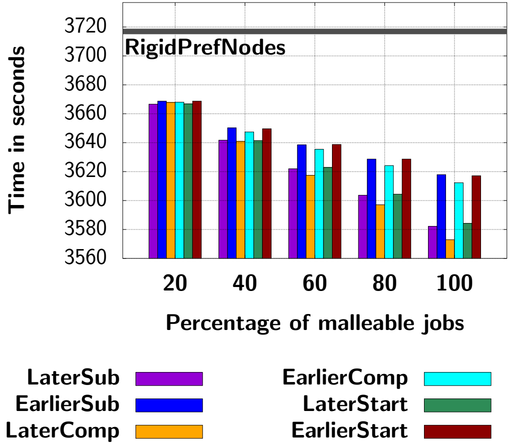


Elastic Job Scheduling Strategy

- Job scheduler strategy to decide when and with how many resources to shrink or grow jobs
- We adapted an existing strategy[2] designed to improve the global throughput:
    - Jobs that run with less than the preferred number of nodes are expanded to their preferred number
    - If there are pending jobs waiting to be started, running jobs are shrunk to do so, if possible. However, running jobs are never shrunk to less than their preferred number of nodes. Pending jobs are started with any number of nodes
    - Jobs that run with less than the maximum number of processes are expanded

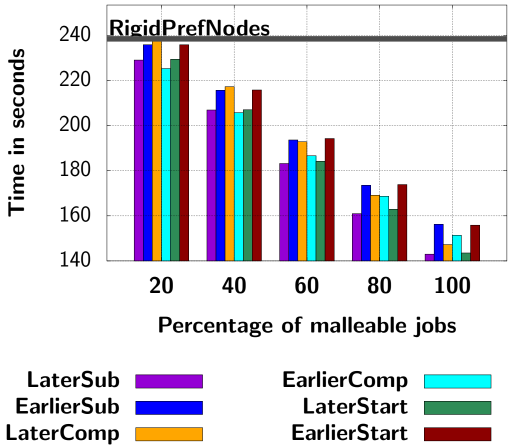
[2] S. Iserte, R. Mayo, et al., DMRlib: Easy-coding and Efficient Resource Management for Job Malleability, IEEE Trans., 2020.

Simulation: Makespan



- The makespan decreases with an increasing percentage of malleable job
- For 100% malleable jobs, the makespan decreases by up to 20%
- No clear winner between the six considered priorities

Simulation: Job Waiting Time



- The average job waiting time decreases with an increasing percentage of malleable jobs
- For 100% malleable jobs, the average job waiting time decreases by up to 80%
- No clear winner between the six considered priorities

Conclusions

- We have proposed a novel malleability scheme at the intermediate level of an AMT runtime system
- Multiple nodes can be added or released on-the-fly
- No explicit synchronization points, additional programming effort, or human input is required
- Adding and releasing nodes has little overhead and scales well
- Simulation results show that adaptive resource management of supercomputers pays off
- In future work, our malleability scheme should be evaluated with other task models and work stealing schemes

Thank you for your attention!

Please feel free to ask questions