Resource Elasticity at Task-Level

Jonas Posner
Advisor: Prof. Dr. Claudia Fohry
University of Kassel, PLM, Germany
{jonas.posner | fohry}@uni-kassel.de

Abstract—Adaptive resource management of supercomputers enables the resources of running jobs to be changed dynamically, which highly improves the throughput of supercomputers compared to conventional static resource management. However, adaptivity must be addressed by both global job schedulers and user applications, for which a non-negligible additional development effort arises.

This work proposes a novel resource elasticity scheme at the intermediate level of a task-based runtime system. Applications using our runtime system automatically adapt to the addition and release of multiple compute nodes and dynamically balance the load without requiring additional development effort. Experiments using up to 128 nodes demonstrate low costs and great scalability for both adding and releasing multiple compute nodes on-the-fly.

Keywords—Resource Elasticity, Malleability, Task-based Parallel Programming

1 Introduction

Adaptive resource management of supercomputers offers several benefits compared to conventional static resource management, including highly improved global throughput and decreased energy consumption [1, 2]. However, adaptivity must be backed by at least three major layers: (1) global job schedulers, (2) programming systems, and (3) algorithms/applications. Moreover, a communication API between (1) and (2) must be provided. Recent research addresses these layers, but no comprehensive solution has yet been established.

Elastic algorithms cause a non-negligible additional development effort, and often focus on iterative approaches that automatically provide explicit synchronization points. This work, in contrast, proposes a novel resource elasticity scheme at the intermediate level of a task-based runtime system. Applications using our runtime automatically adapt to the addition and release of multiple compute nodes and dynamically balance the load without requiring explicit synchronization points or additional programming effort.

We build on the dynamic independent tasks pattern (*DIT*), which is well suited for tree-based algorithms solving search, optimization, and approximation problems [3, 4]. In DIT, tasks encapsulate sub-computations that can be executed in parallel to other tasks. Tasks can not communicate with other tasks, except for parameter passing when generating child tasks at runtime. Typically, DIT is coupled with *work stealing* to balance the load dynamically at runtime. This work considers a multi-threaded lifeline-based variant [5], in which each compute node runs a single process that maintains multiple worker threads. Local workers share tasks with other local workers, and only when the entire process runs out of tasks, the process attempts to steal tasks from random processes, followed by lifeline buddies. The latter are predetermined by a graph, and record unsuccessful steal requests and possibly answer them later.

Section 2 sketches our scheme, Section 3 reports experiments, and Section 4 finishes with conclusions.

2 Resource Elasticity Scheme

Addition and release of compute nodes is controlled by process 0 (denoted as $\mathtt{P}_0$), which can not be released. Resource changes can be triggered any time, but multiple requests are handled sequential. Added processes are denoted as `P(add)`, extracted ones as `P(rel)`, and staying ones as `P(stay)`. Following actions are performed concurrently to task processing, but *not* to work stealing.

Release: $\mathtt{P}_0$ broadcasts a list of all `P(rel)`. Each `P(rel)` will no longer send steal requests to other processes. Each `P(stay)` recalculates the lifeline graph avoiding a dissection of it and deletes all open steal requests from/to any `P(rel)` to avoid sending/receiving tasks to/from any `P(rel)`. Each process notifies $\mathtt{P}_0$ of the completion of these actions. When $\mathtt{P}_0$ has received all acknowledgments, it is guaranteed that no steal requests related to any `P(rel)` are open and no more will be sent. Nevertheless, any `P(rel)` may still process tasks. Thus, $\mathtt{P}_0$ causes each `P(rel)` to stop task processing, and to send all open tasks and interim results to a lifeline buddy (if no lifeline buddy stays, a random `P(stay)` is chosen). After system-wide success, all `P(rel)` have no tasks left, and will not be included into any steal request. Thus, $\mathtt{P}_0$ shuts down all `P(rel)`.

Add: $\mathtt{P}_0$ starts all `P(add)`, which automatically connect to the running application. Afterwards, $\mathtt{P}_0$ causes all processes to recalculate the lifeline graph, and all new lifeline buddies of each `P(add)` immediately send tasks to them.

3 Experiments

We implemented the elasticity scheme by extending the multi-threaded Global Load Balancing library [5] and used the elastic runtime "APGAS for Java" [6] for parallelization. Experiments were run on Goethe-HLR [7] using up to 128 nodes, each providing two 20-core CPUs. We started one process per node; each with 40 worker threads. We deployed two synthetic benchmarks (called *StatSyn* and *DynSyn*), which perform some placeholder computations to provide smooth weak scaling enabling an accurate analysis of running times. All runs were configured with a base computation time of $100s$, so that run without any resource changes takes this time plus some time for work stealing. Adding and releasing compute nodes is triggered after $50s$ running time, and affects the total running time.

StatSyn generates $6K$ tasks per worker and distributes them evenly at the beginning. *DynSyn* starts the computation with one task, and generates a perfect m-ary task tree dynamically at runtime, resulting in an average of $10M$ tasks per worker. For both benchmarks, tasks have a 20% variation in their duration.

Figures 1 and 2 report the running time overheads for halving and doubling the number of compute nodes, respectively. Releasing costs vary between $0.01s$ and $0.26s$ for StatSyn and $0.19s$ and $0.45s$ for DynSyn, and are mainly caused by sending tasks and results from `P(rel)` to `P(stay)`. Adding costs vary between $1.40s$ and $1.84s$ for StatSyn and $0.37s$ and $0.76s$ for DynSyn, and are mainly caused by the delay of new workers until they can start task processing. Thus, adding nodes incurs slightly higher costs than releasing nodes. Adding costs are about $1s$ higher for StatSyn than for DynSyn, because proportionally more additional work stealing is caused for StatSyn than for DynSyn. Since both releasing and adding new compute nodes is performed in a distributed way and asynchronously to task processing, the costs increase only gently with the number of nodes, resulting in good scalability.

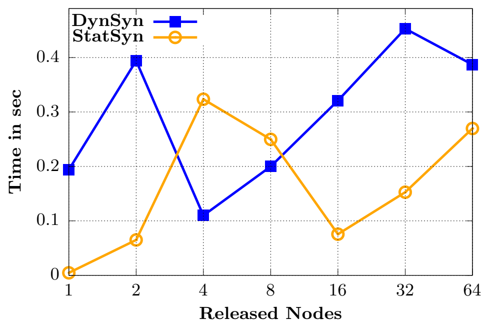

Figure 1: Costs for releasing nodes

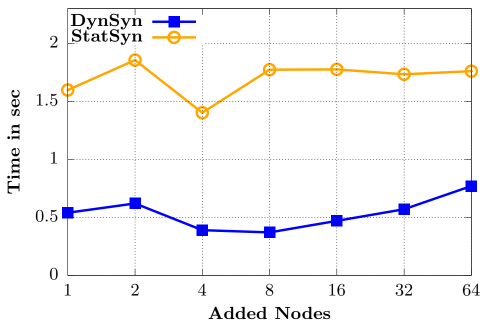

Figure 2: Costs for adding nodes

4 Conclusions

In this work, we have proposed a novel task-level resource elasticity scheme that enables the addition and release of compute nodes on-the-fly, without requiring explicit synchronization points or additional programming effort. We have conducted experiments with our implementation of the scheme, which have shown low costs and good scalability for both adding and releasing compute nodes.

Future work should consider more benchmarks and a deeper cost analysis. In addition, the resource elasticity scheme should be extended to be able to cope with unexpected resource changes, such as fail-stop failures.

References

[1] S. Iserte, R. Mayo *et al.*, "DMRlib: Easy-coding and efficient resource management for job malleability," *TC*, 2020.

[2] M. Chadha, J. John *et al.*, "Extending SLURM for dynamic resource-aware adaptive batch scheduling," in *HiPC*, 2020.

[3] B. Archibald, P. Maier *et al.*, "YewPar: Skeletons for exact combinatorial search," in *PPoPP*, 2020.

[4] R. Harrison, G. Beylkin *et al.*, "MADNESS: A multiresolution, adaptive numerical environment for scientific simulation," *SIAM*, 2016.

[5] P. Finnerty, T. Kamada *et al.*, "Self-adjusting task granularity for global load balancer library on clusters of many-core processors," in *PMAM*, 2020.

[6] O. Tardieu, "The APGAS library: resilient parallel and distributed programming in Java 8," in *SIGPLAN*, 2015.

[7] Goethe-HLR, 2021. [Online]. Available: https://www.top500.org/system/179588