Asynchronous Many-Tasking (AMT): Load Balancing, Fault Tolerance, Resource Elasticity

Jonas Posner
University of Kassel (Germany), PLM, jonas.posner@uni-kassel.de

Motivation

- Recently, HPC applications are getting more and more diverse, including irregular ones limiting the predictability of computations.
- To enable efficient and productive programming of today's supercomputers and beyond, a variety of issues must be addressed, e.g.:
  - *Load Balancing*: utilizing all resources equally,
  - *Fault Tolerance*: coping with hardware failures, and
  - *Resource Elasticity* : allowing the addition/release of resources.
- In this work, we address above issues in the context of AMT for clusters.
- In AMT, programmers split a computation into many fine-grained execution units (called *tasks*), which are dynamically mapped to processing units (called *workers*) by a runtime system. We consider *dynamic independent tasks*, which can be generated at runtime.

Load Balancing

- We propose a coordinated work stealing technique that transparently schedules tasks to resources of the overall system, balancing the workload over all processing units.
- In this context, we introduce novel tasking constructs for spawning dynamic independent tasks and computing their results.
- Tasks can be canceled, which is useful for, e.g., search problems.
- Productivity evaluations show intuitive use compared to other programming systems such as PCJ and Spark.
- Experiments show good scalability.

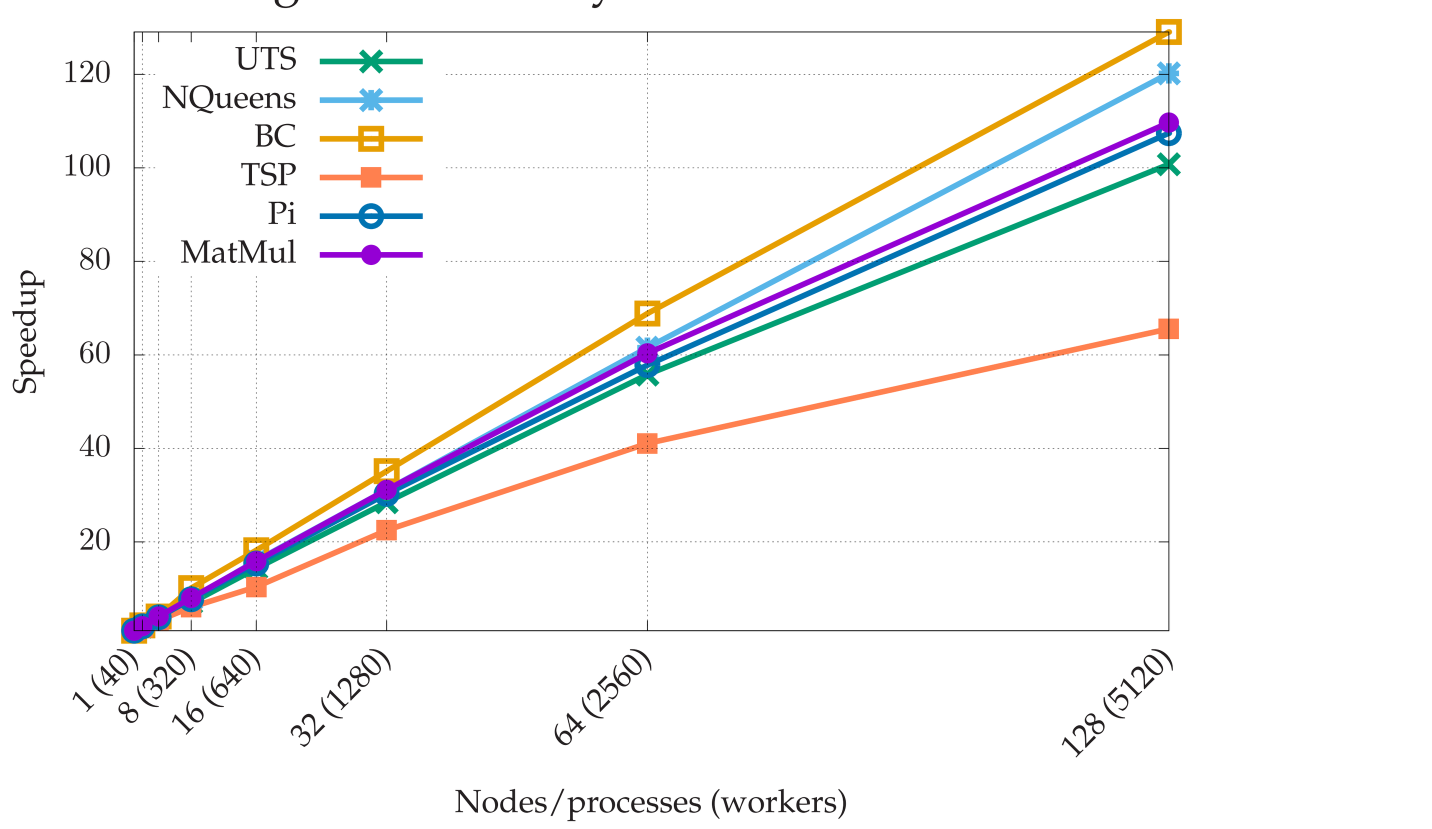

Figure 1: Inter-process speedups over running time with 1 process with 40 workers

References

[1] Jonas Posner. *Load Balancing, Fault Tolerance, and Resource Elasticity for Asynchronous Many-Task Systems*. PhD thesis, University of Kassel (Germany), 12/2021. submitted.

Fault Tolerance

- We propose four techniques to protect programs transparently.
- All perform localized recovery and continue the program execution with fewer resources after failures.
  - *Task-level Checkpointing (TC)*: Writes uncoordinated checkpoints comprising descriptors of all open tasks in a resilient store.
  - *Incremental and Selective Task-level Checkpointing (IncTC)*: Saves only parts of open tasks.
  - *Supervision with Steal Tracking (SST)*: Writes *no* checkpoints at all, but exploits natural task duplication of work stealing.
  - *Combination of TC and SST (LogTC)*: Logs stealing events to reduce the number of checkpoints.
- Experiments show no clear winner between the techniques.
- Compared to the well-known checkpoint/restart library DMTCP, our techniques clearly pay off and have significantly less overhead.
- For instance, TC has a failure-free running time overhead below 1% and a recovery overhead below 0.5 seconds, both for smooth weak scaling.
- We derive formulas predicting running times including failure handling.

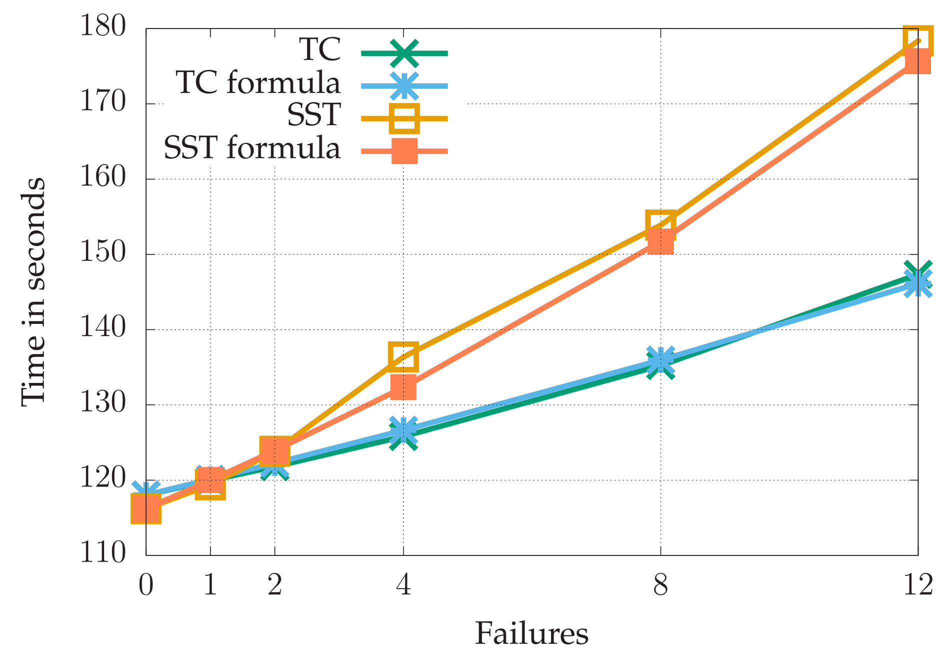

Figure 2: Total running times for failures

- Simulations of job set executions show that the makespan can be reduced by up to 97%.

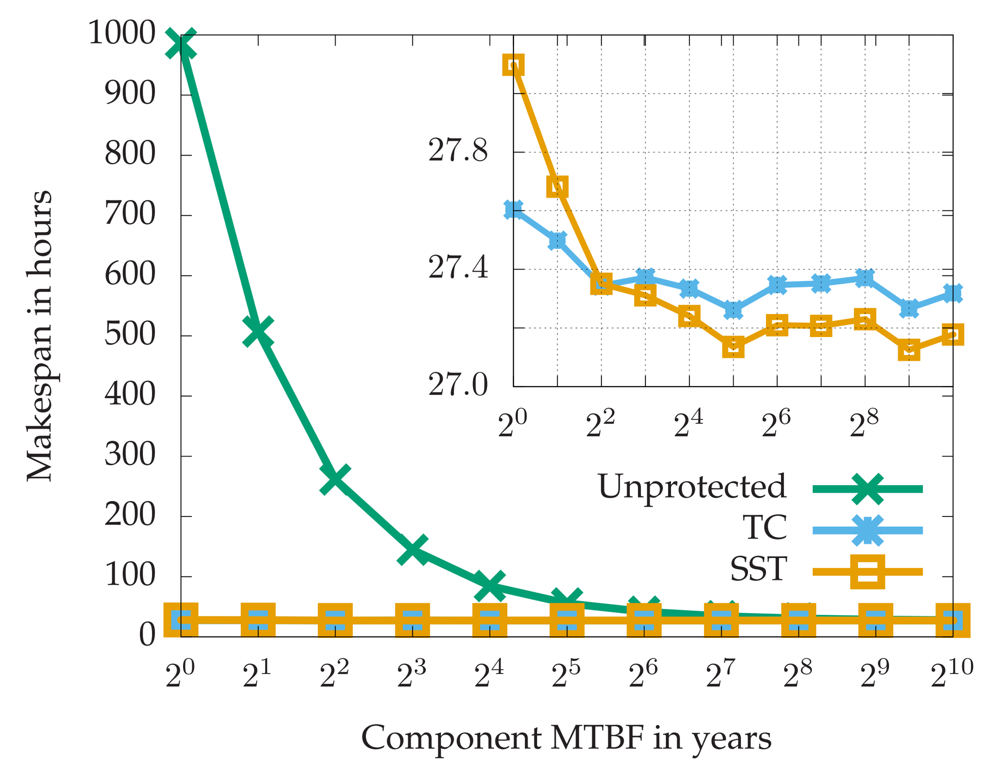

Figure 3: Makespan simulations of unprotected jobs and protected jobs

Resource Elasticity

- We propose a technique to enable the addition and release of nodes at runtime by transparently relocating tasks accordingly.
- We derive formulas that estimate the overhead-free running time of work stealing programs with a changing number of resources.
- Analyses show costs for adding and releasing nodes below 0.5 seconds.

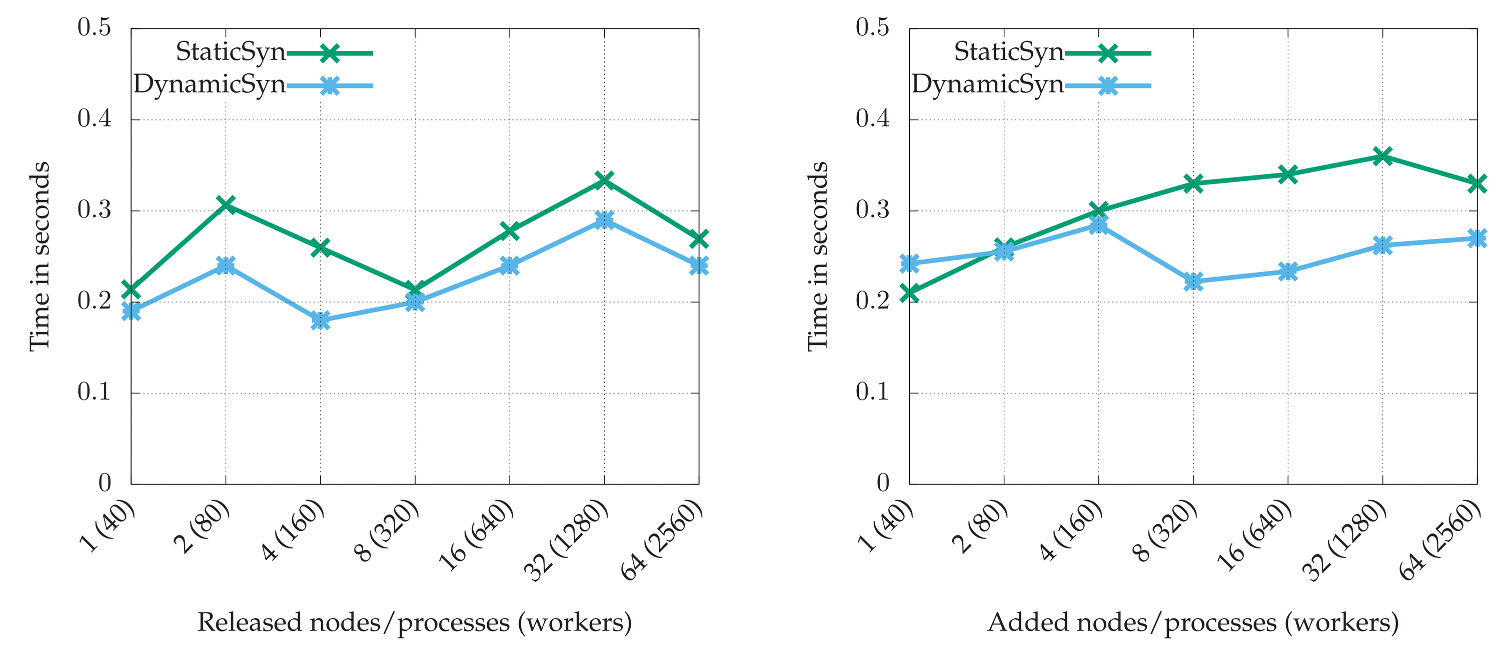

Figure 4: Costs for adding and releasing nodes

- Simulations of job set executions with several heuristics show that the makespan can be reduced by up to 20%.

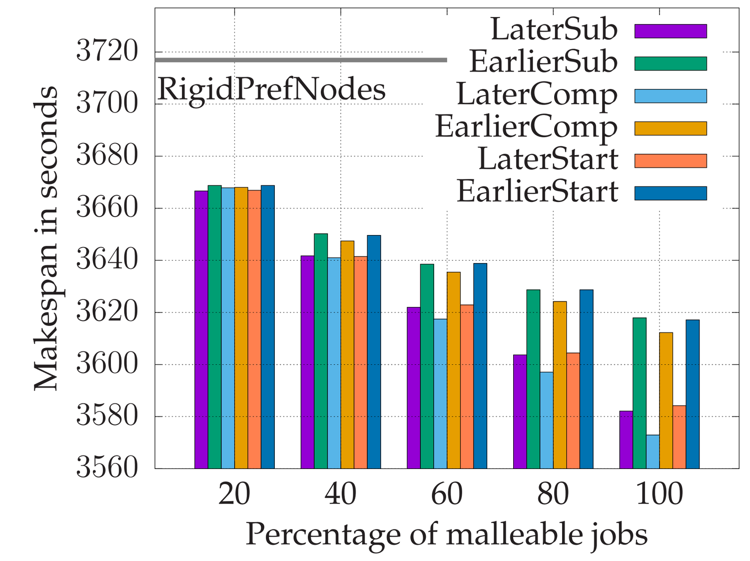

Figure 5: Makespan simulations of a varying number of malleable jobs

Conclusions

- We have proposed
  - a novel coordinated work stealing technique that achieves both intra- and inter-process load balancing,
  - four novel fault tolerance techniques to protect programs transparently while incurring negligible overhead, and
  - a novel resource elasticity technique that enables programs to transparently adapt to the addition or release of multiple nodes while incurring negligible overhead.
- AMT enables efficient programming, scalability, and can provide load balancing, fault tolerance, and resource elasticity in an efficient way.
- Future work should adapt our techniques to heterogenous architectures such as GPUs or FPGAs.