The Impact of Evolving APGAS Programs on HPC Clusters

Jonas Posner

July 30, 2024

The Impact of Evolving APGAS Programs on HPC Clusters

Jonas Posner

University of Kassel, Kassel, Germany
jonas.posner@uni-kassel.de

Abstract High-performance computing (HPC) clusters are traditionally managed statically, i.e., user jobs maintain a fixed number of computing nodes for their entire execution. This approach becomes inefficient with the increasing prevalence of dynamic and irregular workloads, which have unpredictable computation patterns that result in fluctuating resource needs at runtime. For instance, nodes cannot be released when they are not needed, limiting the overall supercomputer performance. However, the realization of jobs that can grow and shrink their number of node allocations at runtime is hampered by a lack of support in both resource managers and programming environments.
This work leverages evolving programs that grow and shrink autonomously through automated decision-making, making them well-suited for dynamic and irregular workloads. The Asynchronous Many-Task (AMT) programming model has recently shown promise in this context. In AMT, computations are decomposed into many fine-grained tasks, enabling the runtime system to transparently migrate these tasks across nodes.
Our study builds on the APGAS-AMT runtime system, which supports evolving capabilities, i.e., handles process initialization and termination automatically requiring minimal additions to user code. We enable interactions between APGAS and a prototype resource manager as well as extend the Easy-Backfilling job scheduling algorithm to support evolving jobs. We conduct real-world job batch executions on 10 nodes—involving a mix of rigid, moldable, and evolving programs—to evaluate the impact of evolving APGAS programs on supercomputers. Our experimental results demonstrate a 23% reduction in job batch makespan and a 29% reduction in job turnaround time for evolving jobs.

Keywords: Resource Adaptivity · Evolving Programs · Asynchronous Many-Task

1 Introduction

Today's High-performance computing (HPC) clusters, also called *supercomputer*s, comprise thousands of interconnected computing nodes. Traditionally, supercomputers are managed using a *static* resource allocation approach. Users do not run their applications directly on the nodes; instead, they submit their applications as *jobs* to a *resource manager*, specifying the exact number of nodes and time

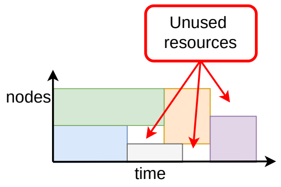

Figure 1: Possible schedule of four rigid jobs

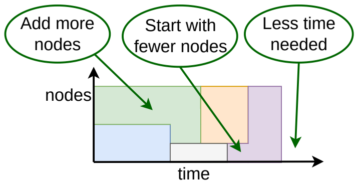

Figure 2: Improved node utilization thanks to adaptive jobs

required. The resource manager then decides the starting order of the jobs and a fixed set of nodes for each job, called *job scheduling*. Because of this static allocation approach, jobs never change their node allocations at runtime, making them *rigid*. This rigidity limits the flexibility of the resource manager in scheduling jobs, often resulting in underutilization of nodes because the job shapes submitted by users may not perfectly match the available node capacity. This mismatch results in gaps in the schedule (i.e., unused nodes) even when there are waiting jobs that cannot be started, thereby limiting overall performance, as shown in Figure 1.

Additionally, the prevalence of *dynamic* and *irregular* workloads—characterized by unpredictable computation patterns resulting in fluctuations in resource needs at runtime—is increasing. Fluctuations in resource needs mean that applications have different phases in which they require or can efficiently exploit varying numbers of nodes. However, static resource allocations prevent jobs from adjusting their allocated number of nodes at runtime to accommodate these fluctuations. Consequently, users often request more nodes than they need to ensure availability during peak demand. This leads to inefficiencies as nodes remain unused during certain job phases but cannot be reallocated to other jobs. Examples of such workloads include adaptive mesh refinement (AMR) simulations [2,22], multiscale analysis [21], fast multipole methods (FMM) [28], and graph analytics [19]. Specific examples of impacted applications are tsunami simulations [20] and flow solvers used in Computational Fluid Dynamics (CFD) [4].

Resource adaptivity offers a promising solution by allowing jobs to dynamically grow and shrink their number of node allocations at runtime. These resource changes can be initiated either by the jobs themselves (called *evolving*) or by the resource manager (called *malleable*) [10]. As shown in Figure 2, *adaptive* jobs can take advantage of gaps in the job schedule that would otherwise remain unused, thereby accelerating their completion. Thus, this flexibility holds the promise of improving both node utilization and job turnaround times [8,24].

However, realizing resource adaptivity's full potential requires significant changes across the entire HPC software stack, including resource managers, runtime systems, and applications. This challenge is evident in widely used resource managers like SLURM and the MPI standard, both of which are based on static resource allocation assumptions.

Despite recent research on resource adaptivity, no widely adopted solution exists. MPI extensions often target iterative workloads or require significant programming effort [1]. The Asynchronous Many-Task (AMT) programming model offers a promising alternative to enable resource adaptivity. In AMT, computations are decomposed into many fine-grained tasks that the runtime system can migrate across nodes. This transparent resource management enables efficient realization of both dynamic load balancing and resource adaptivity.

This study builds on the *APGAS* (Asynchronous Partitioned Global Address Space) AMT runtime system [26]. *APGAS* extends the PGAS model by enabling the spawning of asynchronous tasks at runtime. *APGAS* was recently extended to support malleability [11,12] and evolving capabilities [23]. It automatically manages processes for growing and shrinking, and provides easy-to-use abstractions. Evolving programs autonomously adjust the number of allocated nodes based on real-time resource needs. However, the prior study [23] only evaluated single program runs. This work addresses this shortcoming by evaluating the impact of evolving jobs on supercomputers, making the following contributions:

– We enable interactions between *APGAS* [23] and a prototype resource manager [11]—originally supported malleable jobs—to support evolving jobs.
– We extend the well-known Easy-Backfilling job scheduling algorithm [27] to support evolving jobs.
– We conduct real-world job batch executions on 10 nodes, involving a mix of rigid, moldable, and evolving programs. We use an existing synthetic program with an irregular and dynamic workload that is fully configurable [23].
– Our experimental results demonstrate a 23% reduction in job batch makespan and a 29% reduction in job turnaround times for evolving jobs.

The remainder of this work begins with background on *APGAS* in Section 2. Then, Section 3 outlines our developed interactions between *APGAS* and the resource manager, and Section 4 describes our experimental results. We conclude with related work in Section 5 and a summary in Section 6.

2 Background

This section briefly describes *APGAS* [26], its recently added evolving capabilities [23], and the *GLB* (*Global Load Balancing*) library [29] used as an example.

APGAS is a Java library that builds on the core concepts of IBM's X10 language [5]. It allows the creation of asynchronous tasks either locally or remotely. While *APGAS* automatically schedules tasks among worker threads within each process, programmers are responsible for the mapping of tasks to processes. For synchronization, *APGAS* provides the `finish` construct, which ensures all spawned tasks, including recursive and remote ones, complete before proceeding.

Recently, *APGAS* was extended to support malleability [11,12] and evolving capabilities [23]. These features enable programs to dynamically adjust their node allocations by growing or shrinking. Programmers only need to implement methods to specify the actions required when adjusting resources, such as before and after removing or adding a process.

APGAS handles all process management automatically, including starting and terminating processes. Evolving programs use heuristics to determine when to start a new process (if there is sufficient computational load) or terminate a process (if there are underutilized nodes). This requires no additional code modifications or interactions from the programmer or user.

In previous studies [11,12,23], *APGAS* was used to make the *Global Load Balancing (GLB)* [29] library malleable and evolving. *GLB* automatically balances dynamic and irregular workloads at runtime via work-stealing between processes. A process with no tasks sends a steal request to another process, which transfers some of its tasks or rejects the request. All *GLB* programs are automatically malleable and evolving without further code changes. The combination of *GLB* and *APGAS* allows node changes without interrupting the computations.

3 Interactions between Evolving Jobs and Resource Managers

When evolving programs are run as jobs on supercomputers, the resource manager must support evolving capabilities to allow running jobs to *request* new nodes and *release* nodes dynamically. To enable this, we extend *APGAS* [23] to interact with a prototype resource manager that can already interact with malleable *APGAS* programs [11]. All software is available as open source on GitHub [1].

Workflow of Evolving Jobs. Evolving jobs are submitted to the resource manager with a minimum and maximum number of nodes, rather than a fixed number. The resource manager decides the initial number of nodes to be allocated to the job within the specified range. At startup, *APGAS* runs a daemon thread that monitors the computational load on all nodes every second and decides whether new nodes are needed or whether nodes can be released. Currently, only one node at a time is requested or released [23].

Releasing Nodes. *APGAS* automatically initiates a shrinkage *without* requesting feedback from the resource manager. As soon as *APGAS* has completed the actions defined by the programmer, the corresponding process is terminated, and *APGAS* sends a `release`-message with the name of the node to the resource manager. The resource manager then removes the node from the job and can assign it to another job in the next scheduling-interval. Since an evolving *APGAS* program has a more detailed insight into its dynamic computing load, it can shrink to one node, regardless of the originally specified minimum.

Requesting Nodes. If a new node is needed, *APGAS* sends a `request`-message to the resource manager. The resource manager queues all incoming requests and processes them in the next scheduling-interval. Only if the resource manager grants the request, i.e., assigns a new node to the job and sends the name of the node to the running program, *APGAS* starts a new process on the new node and calls the actions defined by the programmer. Jobs does not send a new `request` until the previous one has been answered. If a job releases a

[1] https://github.com/ProjectWagomu/APGAS 10.5281/zenodo.12634598
https://github.com/ProjectWagomu/LifelineGLB 10.5281/zenodo.12634602
https://github.com/ProjectWagomu/ElasticJobScheduler 10.5281/zenodo.12634497

node after it has made a request, the pending request is canceled by the resource manager. After a release, the job can send a new request.

(Evolving) Easy-Backfilling. The prototype resource manager already supports the well-known *EasyBackfilling* algorithm for scheduling rigid jobs [27]. *EasyBackfilling* starts jobs in order of their arrival time. It attempts to fill gaps in the schedule caused by waiting jobs with earlier start times but larger node requirements, with jobs arriving later with smaller node requirements. However, only jobs that do not delay the start of the leading job are backfilled. Moldable jobs, for which the number of nodes can be determined at the start, are started with the maximum possible number of available nodes.

We implement a new job scheduling algorithm, called *EvolvEasyBack*, to support evolving jobs. As mentioned above, all incoming node requests are queued. At the next scheduling-interval, *EvolvEasyBack* processes these requests first, in the order they were received. As long as there are idle nodes, *EvolvEasyBack* assigns nodes to the requesting jobs. If a requesting job is no longer running or has already been assigned its maximum number of nodes, the request is canceled. If there are no requests left and there are idle nodes, the job scheduling algorithm of *EasyBackfilling* is executed.

4 Evaluation

In this section, we experimentally evaluate the impact of evolving *APGAS* programs on supercomputers. We conduct job batch executions consisting of rigid, moldable, and evolving programs on 10 nodes, varying the proportion of moldable/evolving jobs from 0% to 100%.

First, we describe the experimental setting in Section 4.1. We then analyze the results in Section 4.2 and discuss the findings in Section 4.3.

4.1 Experimental Setting

Environment. Experiments were conducted on the cluster at the University of Kassel[2]. Each node consists of two 6-core Intel Xeon E5-2643 CPUs and 128 GB of main memory. We encapsulated each job batch into a SLURM job having 11 nodes. One node runs our resource manager, while the remaining 10 nodes serve as computing nodes for executing jobs. Java was used in version 19.0.2.

Jobs. For all jobs, we use the *EvoTree* benchmark [23], a synthetic benchmark designed to simulate highly irregular and dynamic workloads. As shown in Figure 3, *EvoTree* starts with a single task (1) that dynamically creates a perfect m-ary task tree. It then creates (2) a sequential branch (3) that ends in another tree (4). An evolving program could grow for the first tree, shrink to a single node for the sequential branch, and then grow for the second tree. *EvoTree* is implemented with *APGAS* + *GLB*, thus automatically balancing the load as well as releasing nodes and requesting new ones as needed.

[2] https://www.hkhlr.de/en/clusters/linux-cluster-kassel

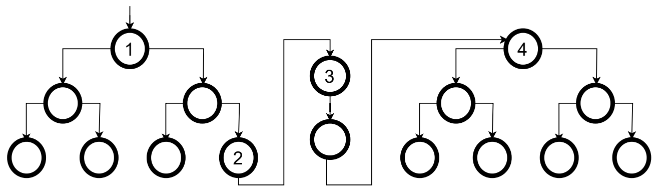

Figure 3: *EvoTree* example run: two perfect m-ary task trees, each with two children and a depth of three, connected by a sequential branch of two [23].

Table 1: Job configurations of *EvoTree*

Rigid: Number of Nodes	**Moldable/Evolving: Range of Nodes**
1	1-2
2	1-4
4	2-8
8	4-10

For rigid jobs, we configured each tree with a computation time of 50 seconds and the sequential branch with a computation time of 100 seconds. These times remain constant regardless of the number of initial processes, due to *EvoTree*'s smooth weak scaling.

Job Batches. We pseudo-randomly generated five job batches, each containing 20 rigid jobs. For each of these job batches, we created additional job batches where 20%, 40%, up to 100% of the jobs were made evolving. Our job configurations are listed in Table 1. The job submission times are generated stochastically using a uniform distribution between 0 and 10 minutes. We scheduled each job batch with both *EasyBackfilling* and *EvolvEasyBack*. Evolving jobs submitted to *EasyBackfilling* are treated as moldable.

4.2 Experimental Results

In the following, we evaluate our averaged experimental results.

Evolving Runtime Performance. The running time costs of the programs were as follows: 4.82 seconds for starting, 3.94 seconds for growing and 3.89 seconds for shrinking. These times only pertain to the durations logged in the respective methods; the actual task computations and work-stealing by unaffected workers continue uninterrupted. These findings are consistent with the results of previous studies [11,23].

Figure 4 shows the number of shrinking and growing events per evolving job. The number of shrinking events decreases as the proportion of evolving jobs increases. With 20% evolving jobs, there are 3.5 shrinking events per job; while with 100% evolving jobs, there are 2.2 shrinking events per job. In contrast, the number of growing events fluctuates less, ranging between 1.9 and 2.2 per job.

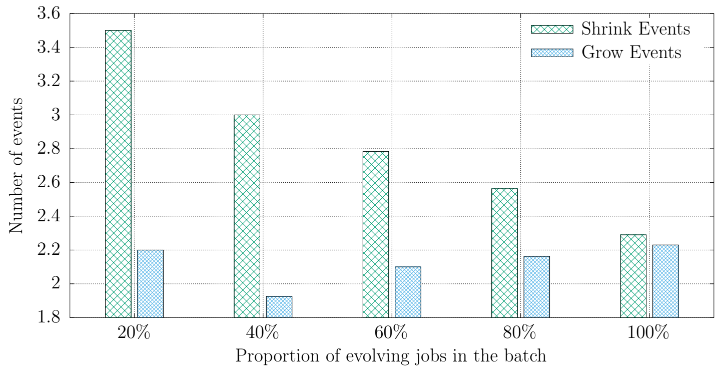

Figure 4: Average number of shrinking and growing events per evolving job for *EvolvEasyBack*

The higher frequency of shrinking events is because, unlike growing events, they are executed without needing a response from the resource manager. A previous study evaluating malleable *APGAS* programs on supercomputers showed always less than one shrinking and one growing event per job [11]. The higher number of events in this study is due to the significantly more dynamic nature of the jobs.

Supercomputer Performance. Supercomputer operators and individual users may have different perspectives on performance metrics. Operators focus on the time needed to compute a job batch (job batch makespan) and node utilization. In contrast, users are more concerned with the time it takes to get their job results (job turnaround time). We will first address the operators' perspective and then the users' perspective.

Makespan. Figure 5 shows the job batch makespans. With 0% moldable/evolving jobs, *EasyBackfilling* and *EvolvEasyBack* perform similarly, resulting in makespans of 1507 and 1510 seconds, respectively. This is expected, as both *EasyBackfilling* and *EvolvEasyBack* execute the same algorithm for rigid jobs. The makespan decreases with an increasing proportion of moldable/evolving jobs, as expected. *EasyBackfilling* achieves a makespan of 1385 seconds with 100% moldable jobs, due to increased flexibility for job scheduling. However, between 20% and 80% moldable jobs, *EasyBackfilling* shows no improvement or even a degradation, as the flexibility of moldable jobs is insufficient to fill scheduling gaps.

In contrast, *EvolvEasyBack* consistently decreases the makespan with an increasing proportion of evolving jobs. With 100% evolving jobs, *EvolvEasyBack* achieves a makespan of 1157 seconds, representing a 23% reduction compared to 0% evolving jobs and a 16% reduction compared to *EasyBackfilling* with 100% moldable jobs.

Node Utilization. Figure 6 shows the node utilization. With 0% moldable/evolving jobs, *EasyBackfilling* and *EvolvEasyBack* have similar node utilizations of 81%. With increasing proportion of moldable/evolving jobs, node utilization generally increases.

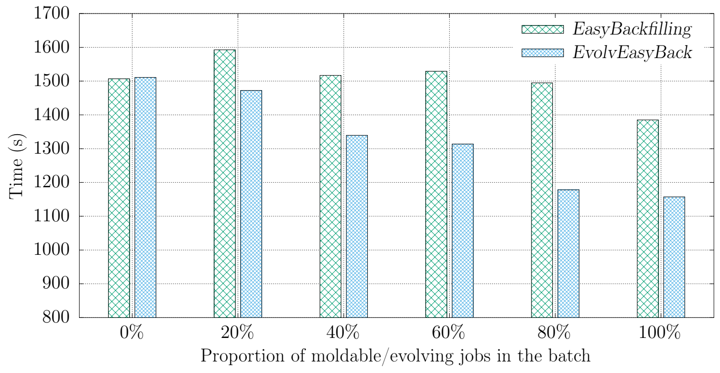

Figure 5: Average job batch makespan depending on the proportion of moldable/evolving jobs and the job scheduling algorithm used

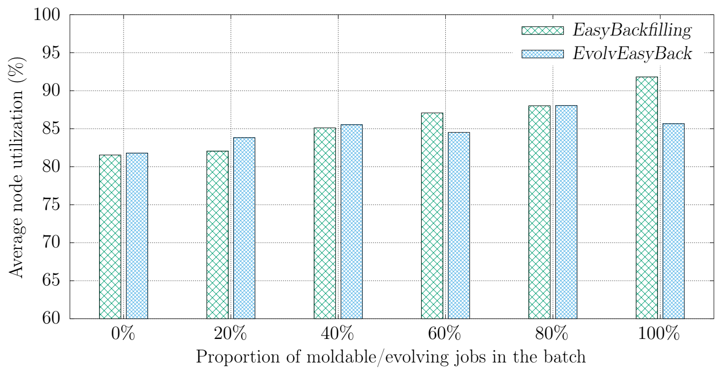

Figure 6: Average node utilization depending on the proportion of moldable/evolving jobs and the job scheduling algorithm used

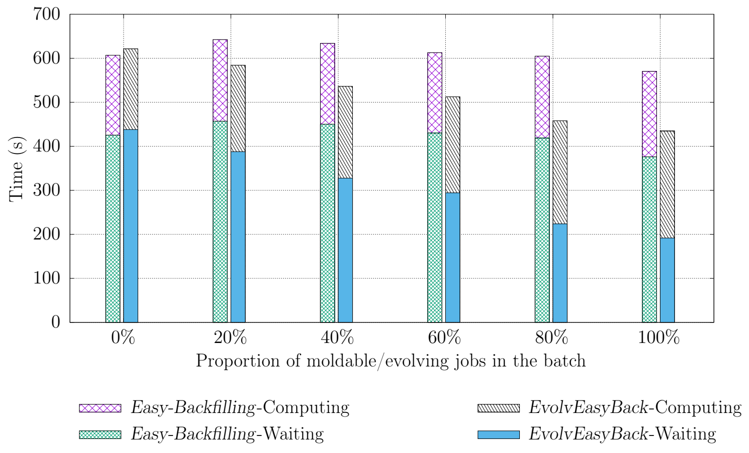

Figure 7: Average job turnaround times depending on the proportion of moldable/evolving jobs and the job scheduling algorithm used

EasyBackfilling has a node utilization of 91% with 100% moldable jobs. *EvolvEasyBack* has a node utilization of 85% with 100% evolving jobs, although it reaches 88% utilization with 80% evolving jobs. However, it is crucial to note that while *EasyBackfilling* appears to have higher node utilization compared to *EvolvEasyBack*, this is not necessarily beneficial. When considering the previous makespan values, we see that *EasyBackfilling* allocates nodes to jobs that cannot fully utilize them due to their evolving nature, particularly during phases where they need only one node.

In contrast, *EvolvEasyBack* slightly increases node utilization (from 81% to 85%) but significantly reduces makespan (from 1510 seconds to 1157 seconds). This improvement is because evolving jobs can release unused nodes, which can then be reallocated to other jobs, optimizing overall performance.

Job Turnaround Time. Figure 7 shows the job turnaround times, which include both job waiting time and job computing time. With 0% moldable/evolving jobs, both *EasyBackfilling* and *EvolvEasyBack* have similar turnaround times, with waiting times of 425-438 seconds, and computation times of 181 seconds.

As the proportion of evolving jobs increases, *EvolvEasyBack* consistently reduces job waiting times. For instance, with 100% evolving jobs, the waiting time is significantly reduced to 191 seconds, compared to 376 seconds with *EasyBackfilling*. Although the computation time increases slightly with *EvolvEasyBack*, the overall turnaround time is still greatly improved due to the reduction in waiting time. Specifically, *EvolvEasyBack* achieves a 29% reduction (from 605 seconds to 428 seconds) when compared to *EasyBackfilling* with 0% evolving jobs, and a 23% reduction (from 564 seconds to 428 seconds) when compared to *EasyBackfilling* with 100% evolving jobs.

4.3 Discussion

The experimental results confirm our expectations that evolving jobs increase flexibility, leading to a positive impact on the supercomputer's performance. The most efficient setting was *EvolvEasyBack* with 100% evolving jobs, resulting in a 23% reduction of the makespan and a 29% reduction in job turnaround times. Node utilization was only slightly improved by *EvolvEasyBack*, suggesting that further improvements could be possible with different job batches. The results from *EasyBackfilling* indicate that merely analyzing node utilization is not sufficient, as jobs may not use their allocated nodes efficiently.

This study has some limitations due to the experimental setup. Future work should include larger-scale experiments and the use of real-world applications rather than synthetic ones. Additionally, *EvolvEasyBack* is currently simplistic, as node requests are always fulfilled when possible. Consideration should be given to fairness, such as limiting or prioritizing node requests. This could also involve enabling a mix of malleable and evolving jobs. *APGAS* is classified as a prototype, and implementing the techniques with, e.g., C and MPI could significantly reduce the running time cost of resource allocation changes [15].

5 Related Work

In practice, only a few applications can change the number of their processes at runtime due to the complexity of such mechanisms [3,18]. However, resource adaptivity has recently gained research interest, leading to various techniques and prototypes [1,13].

While MPI supports process spawning at runtime since version 2, it has several limitations [14]. For instance, spawn operations are synchronous and collective, blocking calling processes during the creation and initialization of new processes [7]. MPI extensions such as Elastic MPI [7] and DMRlib [17] have been proposed to address issues like this. Recently, MPI Sessions formed the basis for a novel adaptive MPI runtime environment that conceptualizes resource changes as transformations between process sets, overcoming previous limitations such as global synchronization [16]. However, most MPI studies focus on iterative workloads and not on irregular and dynamic ones like this work.

Programming environments outside MPI that provide resource adaptivity are still rare. Notable examples include X10 [5] and Charm++ [25]. X10 and *APGAS* initially had similarities, including basic support for removing and starting processes. Charm++ also supports resource adaptivity by decomposing computations into independent objects that can migrate between processes.

Research on adaptive job scheduling algorithms is comparatively sparse compared to rigid ones, but they are known to improve performance on various metrics, such as node utilization. Some proposed adaptive scheduling algorithms have been implemented directly in practical settings either by extending existing resource managers such as SLURM [7,17] and Torque/Maui [25], or by developing custom prototype resource managers [11,12,15]. These implementations are tailored to specific runtime environments, allowing for real-world experiments.

Other adaptive job scheduling algorithms are evaluated via simulations [6,9,24], offering benefits such as speed, reproducibility, and resource efficiency over real-world experiments. Simulations enable thorough algorithm evaluations under diverse conditions, but ensuring accurate input modeling, especially for job traces, application scaling, and adaptivity, remains challenging for reliable outcomes.

6 Conclusion

This study demonstrated the benefits of evolving programs on supercomputers, especially for handling dynamic and irregular workloads. We leveraged *APGAS*, which provides evolving capabilities, allowing programs to autonomously adjust their allocated nodes. We enabled interactions between *APGAS* and a prototype resource manager, and extended Easy-Backfilling to support evolving jobs.

Real-world job executions on a 10-node cluster showed a 23% reduction in job batch makespans and a 29% reduction in job turnaround times for evolving jobs.

Future work should explore mixing malleable and evolving jobs to further validate these benefits.

Acknowledgements I thank Patrick Finnerty for all the fruitful discussions we have at our meetings.

References

1. Aliaga, J.I., Castillo, M., Iserte, S., Martín-Álvarez, I., Mayo, R.: A Survey on Malleability Solutions for High-Performance Distributed Computing. Applied Sciences **12**(10) (2022). https://doi.org/10.3390/app12105231
2. Berger, M.J., Oliger, J.: Adaptive mesh refinement for hyperbolic partial differential equations. Journal of Computational Physics **53**(3), 484–512 (1984). https://doi.org/10.1016/0021-9991(84)90073-1
3. Bernholdt, D.E., Boehm, S., Bosilca, G., Venkata, M.G., Grant, R.E., Naughton, T., Pritchard, H.P., Schulz, M., Vallee, G.R.: A survey of MPI usage in the US Exascale Computing Project. Concurrency and Computation: Practice and Experience (CCPE) **32**(3) (2020). https://doi.org/10.1002/cpe.4851
4. Bramkamp, F., Bramkamp, F., Bramkamp, F., Lamby, P., Lamby, P., Müller, S., Müller, S.: An adaptive multiscale finite volume solver for unsteady and steady state flow computations. Journal of Computational Physics (2004). https://doi.org/10.1016/j.jcp.2003.12.005
5. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. SIGPLAN Notices **40**(10), 519–538 (2005). https://doi.org/10.1145/1103845.1094852
6. Comprés, I., Arima, E., Schulz, M., Rotaru, T., Machado, R.: Probabilistic Job History Conversion and Performance Model Generation for Malleable Scheduling Simulations. In: Proceedings International Conference on High Performance Computing (ISC) Workshops (HPCMALL). pp. 82–94. Springer (2023). https://doi.org/10.1007/978-3-031-40843-4_7
7. Comprés, I., Mo-Hellenbrand, A., Gerndt, M., Bungartz, H.J.: Infrastructure and API Extensions for Elastic Execution of MPI Applications. In: European MPI Users' Group Meeting. pp. 82–97. ACM (2016). https://doi.org/10.1145/2966884.2966917
8. Dongarra, J. *et al.*: The International Exascale Software Project roadmap. The International Journal of High Performance Computing Applications **25**(1), 3–60 (2011). https://doi.org/10.1177/1094342010391989
9. Eberius, D., Wasi-Ur-Rahman, M., Ozog, D.: Evaluating the Potential of Elastic Jobs in HPC Systems. In: Proceedings International Conference on High Performance Computing, Networking, Storage and Analysis (SC) Workshops (PMBS). pp. 1324–1333. ACM (2023). https://doi.org/10.1145/3624062.3624199
10. Feitelson, D.G., Rudolph, L.: Toward Convergence in Job Schedulers for Parallel Supercomputers. In: Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP). pp. 1–26. Springer (1996). https://doi.org/10.1007/BFb0022284
11. Finnerty, P., Posner, J., Bürger, J., Takaoka, L., Kanzaki, T.: On the Performance of Malleable APGAS Programs and Batch Job Schedulers. Springer Nature Computer Science (2024). https://doi.org/10.1007/s42979-024-02641-7
12. Finnerty, P., Takaoka, R., Kanzaki, T., Posner, J.: Malleable APGAS Programs and their Support in Batch Job Schedulers. In: Proceedings Euro-Par Workshops (AMTE). Springer (2023). https://doi.org/10.1007/978-3-031-48803-0_8
13. Galante, G., da Rosa Righi, R.: Adaptive parallel applications: from shared memory architectures to fog computing (2002-2022). Cluster Computing (2022). https://doi.org/10.1007/s10586-022-03692-2
14. Gropp, W.: MPICH2: A New Start for MPI Implementations. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer (2002). https://doi.org/10.1007/3-540-45825-5_5

15. Huber, D., Schreiber, M., Schulz, M.: A Case Study on PMIx-Usage for Dynamic Resource Management. In: Proceedings International Conference on High Performance Computing (ISC) Workshops (HPCMALL). pp. 42–55. Springer (2023). https://doi.org/10.1007/978-3-031-40843-4_4
16. Huber, D., Streubel, M., Comprés, I., Schulz, M., Schreiber, M., Pritchard, H.: Towards Dynamic Resource Management with MPI Sessions and PMIx. In: European MPI Users' Group Meeting. ACM (2022). https://doi.org/10.1145/3555819.3555856
17. Iserte, S., Mayo, R., Quintana-Ortí, E.S., Peña, A.J.: DMRlib: Easy-Coding and Efficient Resource Management for Job Malleability. Transactions on Computers (TC) (2020). https://doi.org/10.1109/TC.2020.3022933
18. Laguna, I., Marshall, R., Mohror, K., Ruefenacht, M., Skjellum, A., Sultana, N.: A Large-Scale Study of MPI Usage in Open-Source HPC Applications. In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC). ACM (2019). https://doi.org/10.1145/3295500.3356176
19. Lumsdaine, A., Gregor, D., Hendrickson, B., Berry, J.: Challenges in Parallel Graph Processing. Parallel Processing Letters **17**(01), 5–20 (2007). https://doi.org/10.1142/S0129626407002843
20. Mo-Hellenbrand, A., Comprés, I., Meister, O., Bungartz, H., Gerndt, M., Bäder, M., Bader, M.: A Large-Scale Malleable Tsunami Simulation Realized on an Elastic MPI Infrastructure. Proceedings Conference Computing Frontiers (CF) (2017). https://doi.org/10.1145/3075564.3075585
21. Müller, S.: Adaptive Multiscale Schemes for Conservation Laws (2003). https://doi.org/10.1007/978-3-642-18164-1
22. Plewa, T., Linde, T., Weirs, G.: Adaptive Mesh Refinement—Theory and Applications (2008). https://doi.org/10.1007/b138538
23. Posner, J., Goebel, R., Finnerty, P.: Evolving APGAS Programs: Automatic and Transparent Resource Adjustments at Runtime. In: Proceedings Workshop on Asynchronous Many-Task Systems and Applications (WAMTA) (2024). https://doi.org/10.1007/978-3-031-61763-8_15
24. Posner, J., Hupfeld, F., Finnerty, P.: Enhancing Supercomputer Performance with Malleable Job Scheduling Strategies. In: Proceedings Euro-Par Workshops (PECS). Springer (2023). https://doi.org/10.1007/978-3-031-48803-0_14
25. Prabhakaran, S., Neumann, M., Rinke, S., Wolf, F., Gupta, A., Kale, L.V.: A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications. In: Proceedings International Parallel and Distributed Processing Symposium (IPDPS). pp. 429–438. IEEE (2015). https://doi.org/10.1109/IPDPS.2015.34
26. Tardieu, O.: The APGAS Library: Resilient Parallel and Distributed Programming in Java 8. In: Proceedings SIGPLAN Workshops on X10. pp. 25–26. ACM (2015). https://doi.org/10.1145/2771774.2771780
27. Wong, A.K., Goscinski, A.M.: Evaluating the EASY-backfill job scheduling of static workloads on clusters. In: Proceedings International Conference on Cluster Computing (CLUSTER) (2007). https://doi.org/10.1109/CLUSTR.2007.4629218
28. Yokota, R., Barba, L., Narumi, T., Yasuoka, K.: Petascale turbulence simulation using a highly parallel fast multipole method on GPUs. Computer Physics Communications **184**(3), 445–455 (2013). https://doi.org/10.1016/j.cpc.2012.09.011
29. Zhang, W., Tardieu, O., Grove, D., Herta, B., Kamada, T., Saraswat, V., Takeuchi, M.: GLB: Lifeline-based Global Load Balancing Library in X10. In: Proceedings Workshops on Parallel Programming for Analytics Applications (PPAA). pp. 31–40. ACM (2014). https://doi.org/10.1145/2567634.2567639