SN Computer Science (2024) 5:349
https://doi.org/10.1007/s42979-024-02641-7

ORIGINAL RESEARCH

On the Performance of Malleable APGAS Programs and Batch Job Schedulers

Patrick Finnerty[1] · **Jonas Posner**[2] · **Janek Bürger**[2] · **Leo Takaoka**[1] · **Takuma Kanzaki**[1]

Received: 3 November 2023 / Accepted: 18 January 2024

Abstract

Malleability—the ability for applications to dynamically adjust their resource allocations at runtime—presents great potential to enhance the efficiency and resource utilization of modern supercomputers. However, applications are rarely capable of *growing* and *shrinking* their number of nodes at runtime, and batch job schedulers provide only rudimentary support for such features. While numerous approaches have been proposed to enable application malleability, these typically focus on iterative computations and require complex code modifications. This amplifies the challenges for programmers, who already wrestle with the complexity of traditional MPI inter-node programming. *Asynchronous Many-Task* (*AMT*) programming presents a promising alternative. In AMT, computations are split into many fine-grained *task*s, which are processed by *worker*s. This makes transparent task relocation via the AMT runtime system possible, thus offering great potential for enabling efficient malleability. In this work, we propose an extension to an existing AMT system, namely *APGAS for Java*. We provide easy-to-use malleability programming abstractions, requiring only minor application code additions from programmers. Runtime adjustments, such as process initialization and termination, are automatically managed by our malleability extension. We validate our malleability extension by adapting a load balancing library handling multiple benchmarks. We show that both shrinking and growing operations cost low execution time overhead. In addition, we demonstrate compatibility with potential batch job schedulers by developing a prototype batch job scheduler that supports malleable jobs. Through extensive real-world job batches execution on up to 32 nodes, involving rigid, moldable, and malleable programs, we evaluate the impact of deploying malleable APGAS applications on supercomputers. Exploiting scheduling algorithms, such as FCFS, Backfilling, Easy-Backfilling, and one exploiting malleable jobs, the experimental results highlight a significant improvement regarding several metrics for malleable jobs. We show a 13.09% makespan reduction (the time needed to schedule and execute all jobs), a 19.86% increase in node utilization, and a 3.61% decrease in job turnaround time (the time a job takes from its submission to completion) when using 100% malleable job in combination with our prototype batch job scheduler compared to the best-performing scheduling algorithm with 100% rigid jobs.

Keywords Malleable runtime system · Malleable job scheduling · APGAS

Introduction

In the realm of modern supercomputing, the prevalence of *dynamic* and *irregular* workloads—which embodies varying computational demands and unpredictable computational patterns—is steadily rising. Compounded with the traditional *static* resource allocations on supercomputers, this leads to inefficient resource utilization and diminished overall performance.

On today's supercomputers, users do not execute their applications directly on the nodes, but submit them to the *batch job scheduler* in the form of *job*s, specifying the number of nodes and the required time. The batch job scheduler

This article is an extended version of "Patrick Finnerty, Leo Takaoka, Takuma Kanzaki, Jonas Posner: Malleable APGAS Programs and their Support in Batch Job Schedulers. International European Conference on Parallel and Distributed Computing, Workshop Asynchronous Many-Task Systems for Exascale (AMTE) 2023".

This article is part of the topical collection "Applications and Frameworks using the Asynchronous Many Task Paradigm" guest edited by Patrick Diehl, Hartmut Kaiser, Peter Thoman, Steven R. Brandt and "Ram" Ramanujam.

Extended author information available on the last page of the article

Published online: 27 March 2024

Table 1 Job classification by *Feitelson and Rudolph* [11] based on who determines the number of nodes a job runs with and when this determination is made

Decision	By job	By batch job scheduler
At job start	*Rigid*	*Moldable*
At runtime	*Evolving*	*Malleable*

then decides which jobs are started and executed, in which order, and on which nodes. Typically, nodes are used by jobs exclusively, meaning that a node is never utilized by two jobs simultaneously. Thus, the scheduler's role of allocating nodes to jobs amounts to solving a 2-dimensional knapsack problem.

This leads to under-utilization of supercomputer resources, because the job shapes (*required time* × *number of nodes*) submitted by users may not fit perfectly within the total available node capacity of a supercomputer. One approach to alleviating this issue involves introducing *elasticity* for jobs, allowing them to dynamically change their number of nodes at runtime.

As shown in Table 1, jobs can be categorized into four elasticity classes based on who determines the number of nodes a job runs with and when this determination is made [11]. In this article, we focus on *moldable* and *malleable* jobs in which the number of nodes used by the job can be adjusted by the batch job scheduler when the job starts (moldable), and during execution (malleable), respectively.

The flexibility afforded by moldable jobs allows the batch job scheduler to start them more easily than classical rigid jobs that require a specific number of nodes. Malleable jobs offer further opportunities for batch job schedulers. For instance, they make it possible to *shrink* their current node allotment to allow other jobs to be started earlier. Conversely, batch job schedulers can also *grow* the node allotments of running jobs, accelerating their completion. Thus, elasticity promises to increase resource utilization, improve job throughput, and optimize overall performance [36]. Figure 1 shows a concise but impactful scenario comparing rigid (*left*) and elastic (*right*) job executions.

However, programming malleable applications remains more challenging than programming regular rigid ones. Moreover, batch job schedulers and traditional inter-node programming models such as MPI provide only rudimentary support for malleability. While numerous approaches for this purpose have been proposed, they often require complex modifications to application codes. They also typically focus on iterative computations that provide "natural" synchronization points for resource adjustments [24].

Asynchronous Many-Task (AMT) programming is a promising approach to facilitate programmer productivity, handle dynamic and irregular workloads, and enable malleability with only minor changes to application codes. In AMT, programmers split large computations into many fine-grained *task*s, which are then dynamically mapped to processing units (e.g., CPUs), called *worker*s, by the AMT runtime system. Due to this transparent resource management, AMT offers great potential to provide flexible and efficient solutions for malleability. Malleable AMT applications could adapt to resource changes by relocating tasks and data to added resources and away from released resources. While this potential for malleability has been recognized [6, 35], a lack of AMT systems that support malleability in an efficient and simple way still remains.

This article aims to bridge this gap by proposing a malleability extension to an AMT library, namely the open-source *APGAS for Java* [42] (*APGAS* in short). *APGAS* extends the well-known *Partitioned Global Address Space (PGAS)* programming model by adding *asynchronous* task capabilities. Although the original *APGAS* supported changes in the number of processes, it did so in a very rudimentary manner [42]. This was recently improved in the context of the *Lifeline-Based Global Load Balancing* (*GLB*) library [35], but the proposed malleability technique was tightly intertwined with *GLB*. This work disentangles *APGAS* and *GLB* and makes the following main contributions:

- We propose an innovative malleability technique and implement it as an extension to *APGAS* [42]. Our malle-

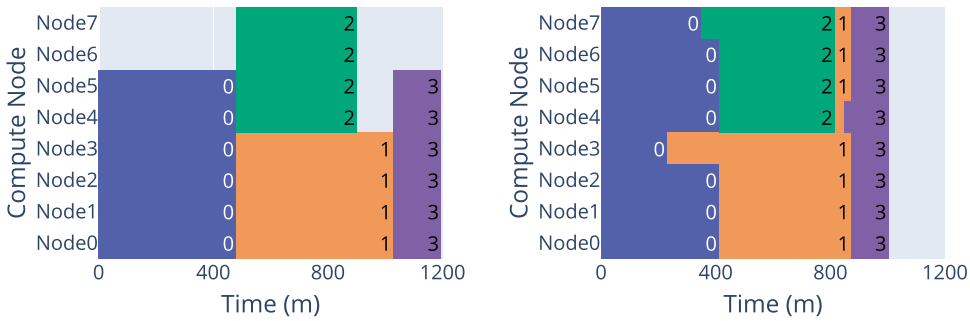

Fig. 1 Scheduling of rigid only jobs (left) and elastic only jobs (right). In the latter situation, the makespan (time needed to schedule and execute the four jobs) decreases from approximately 1200 min to approximately 1000 min thanks to the elastic nature of the jobs (adapted from [36])

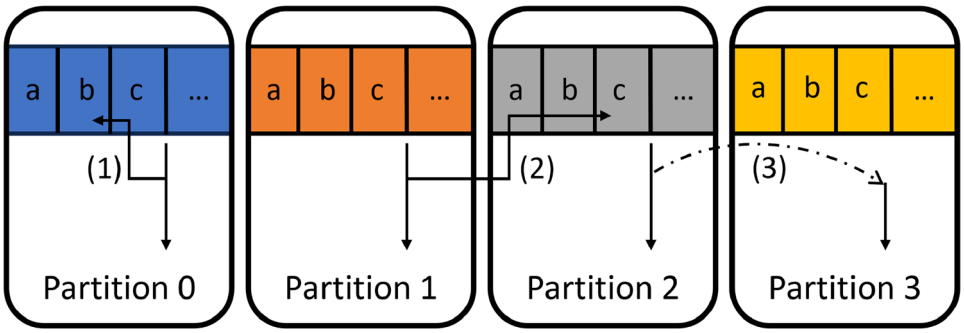

Fig. 2 *Asynchronous Partitioned Global Address Space (APGAS)* [3, 9] programming model. In the PGAS model, every process has its own memory partition, represented here by different colors. A process can access both the local memory partition (1) or a memory partition belonging to another process through the partitioned global address space (2). Local access is faster than remote access. The extended APGAS model adds *Asynchrony* to make it possible to spawn asynchronous tasks that can can spawn other new asynchronous tasks on both the local and remote processes (3) at runtime

ability extension empowers programmers to enable application malleability, requiring only small code additions thanks to our clear abstractions. Runtime adjustments, including process initiation and termination, are automatically managed by our malleability extension.

- We substantiate the usability of our malleable *APGAS* through the adaptation of *GLB* including a collection of its benchmarks.
- We propose a generic communication interface that enables running *APGAS* applications to react to *shrink* and *grow* orders from a batch job scheduler.
- We develop a prototype batch job scheduler (called *ElasticJobScheduler*) that supports malleability and is capable of communicating with malleable *APGAS* applications. In addition, we integrate four job scheduling algorithms—three of which can handle moldable jobs and one of which can handle malleable jobs.
- We perform extensive real-world experiments of job batches execution involving rigid, moldable, and malleable programs to evaluate the impact of deploying malleable APGAS applications on supercomputers. The results show that both shrinking and growing a malleable job's node allotment cause only a small execution time overhead. In addition, the results show a significant improvement of supercomputer performance regarding several metrics, including a 13.09% makespan reduction, a 19.86% increase in node utilization, and a 3.61% decrease in job turnaround time, when using 100% malleable job in combination with our scheduler for malleable jobs compared to the best-performing scheduling algorithm with 100% rigid jobs.

The remainder of this article is structured as follows. We first cover background information on *APGAS* and *GLB* before introducing our malleability extension to *APGAS*. In the "Evaluation" section, we describe our extensive real-world experiments including the derived results. We then discuss related work before concluding. All the software discussed in this article is freely available on GitHub.[1]

Background

In this section, we first provide background information about the APGAS programming model. We then discuss the Lifeline-Based Global Load Balancing scheme we extended using our new malleable programming abstractions.

APGAS Programming Model

The *Partitioned Global Address Space (PGAS)* [3, 9] programming model facilitates programmer productivity for programming inter-node parallel programs. PGAS allows programmers to see memory as a single, logically partitioned, global address space where each process maintains its local memory. It offers direct access to remote memory partitions alleviating the need for explicit message-passing operations as in, e.g., MPI [26].

The PGAS programming model has been implemented in several programming languages and runtime libraries in different ways. For example, Co-Array Fortran (CAF) [29] is a standalone programming language; OpenSHMEM [31], Unified Parallel C (UPC) [10], and UPC++ [4] are libraries for C/C++; and Titanium [45] and PCJ [28] are libraries for Java.

[1] https://github.com/ProjectWagomu.

Listing 1: Distributed parallel "Hello World" in *APGAS*

```
1  finish(()->{
2    for (Place p : places()) {
3      asyncAt(p, () -> {
4        System.out.println("Hello from " + here());
5      });
6    }
7  });
8  System.out.println("Bye");
```

Listing 2: Possible execution result of the *Distributed parallel "Hello World" in APGAS program* running with 4 places. Note that the order of apparition of the *Hello* messages is undeterministic.

```
1  Hello from place(0)
2  Hello from place(1)
3  Hello from place(3)
4  Hello from place(2)
5  Bye
```

Some languages and libraries extend PGAS into the *Asynchronous PGAS (APGAS)* programming model [38]. As illustrated in Fig. 2, this allows *task*s running on a memory partition to spawn new tasks either on the same or on a remote memory partition. In the APGAS model, a large number of asynchronous tasks can be processed by processing units (e.g., CPUs), called *worker*s. Among the programming systems that adopt the APGAS programming model, a notable one is IBM's parallel programming language X10 [7, 23]. The X10 language introduced high-level control structures that allow programmers to easily express task completion constraints.

In this work, we extend the open-source *APGAS for Java library* [42] (*APGAS* for short), which ports the key language constructs of X10 to Java. In *APGAS*, a memory partition is called a *place* and is typically mapped to a physical node or processor in a distributed system.

Places are sequentially numbered from 0 to $n-1$ for an n-process execution. The `asyncAt` construct allows asynchronous task spawn at a remote place. Task termination is managed with the `finish` construct, which only completes once all transitively spawned asynchronous tasks have been executed. This principle is illustrated in Listing 1 along with a possible execution result shown in Listing 2. The `Bye` message in Line 8 of Listing 1 is printed only after each place has completed printing its `Hello` message in Line 4 of Listing 1.

Lifeline-Based Global Load Balancing

Lifeline-Based Global Load Balancing (GLB) [39] is a fully distributed work-stealing scheme that was first implemented as a library in X10 [44, 47] and later adapted in *APGAS* [14, 34]. *GLB* combines random victim selection for stealing work with pre-determined stealing channels (the so-called *lifelines*) through which places that run out of work signal their *lifeline buddys* and passively for work to reach them.

In this work-stealing scheme, the key work abstraction that needs to be implemented by programmers is called a *bag*. It can generally be understood as a queue of tasks, although there are no requirements for the programmer to actually implemented it as such.

The computation starts from a single bag given to the first place. It is then successively split by the stealing network, spreading the work to the other places in the execution. As the computation progresses, each place keeps track of the their contribution to the result. The computation completes when all the places run out of work, which is elegantly implemented using a single `finish` provided by the *APGAS* runtime. The final result is then obtained by performing a reduction across all places.

In the *APGAS* variant, the bag is implemented using two generic types, `B` and `R`, where `B` is the implementing class itself, and `R` is the type of the result produced by the computation. The final overall result is obtained by performing

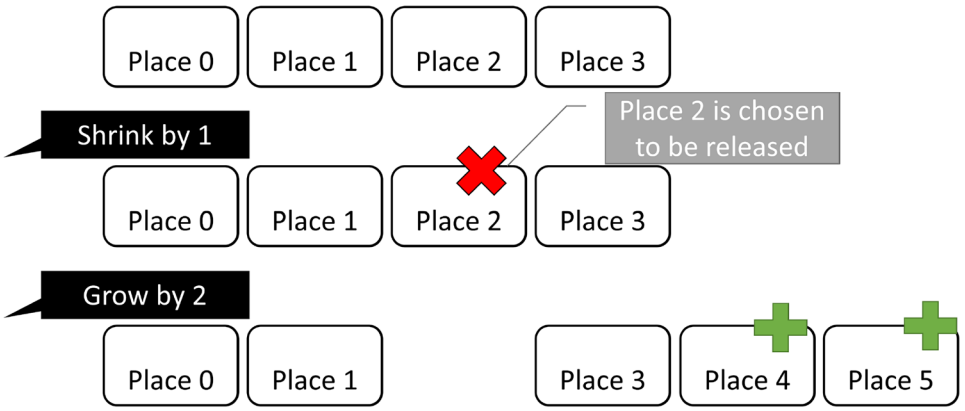

Fig. 3 Malleable *APGAS* program reacting to received shrink/grow orders

a reduction on all the nodes that participated in the computation. It is assumed that each task can be computed by any worker and that they are independent of each other. In addition, tasks cannot communicate with each other, except for passing parameters when creating new tasks. The key operations that a bag implementation needs to perform are:

- `boolean process(int)`: processes a certain amount of work (passed as parameter) contained in the bag
- `B split(boolean)`: returns a new bag instance containing a splitted part (generally half) of the work contained in the original bag; the `boolean` parameter is used to take away all of the contents of the bag in case the work it contains cannot be split
- `void merge(B)`: takes a passed bag and merges its work into the current bag
- `boolean isEmpty()`: returns whether the bag is empty, i.e., whether it contains work or not
- `boolean isSplittable()`: returns whether splitting the bag is possible, i.e., if it contains enough work to be split into two bags by calling `split(false)` or if calling `split(true)` is necessary to obtain work from it
- `void submit(R)`: puts the contribution of this bag into the passed instance of the result `R`

In this work, we build on a *GLB* variant enabling multiple workers per place, where each worker has its own bag instance [13]. Load balancing within a place is performed by the means of a shared bag in which the workers take up more work when they empty their bag, and collaboratively put work back into when it is emptied.

When all the workers of a place run out of work, the place enters a stealing phase. It first attempts to steal work from the shared bag of a randomly selected remote place. If successful in this *random steal*, new workers are spawned and the load balancing within a place starts again. If not, the place establishes its *lifelines* on pre-determined places and passively waits for work to reach it. If able, the places on which the lifelines were established, called *lifeline buddies*, will send back some work to the place that ran out and start new workers using an asynchronous task.

When the computation begins, the bag containing the totality of the work is given to `place(0)`, with the other places stealing from it (either directly or indirectly) through their lifelines. The computation completes when all the places run out of work, which is elegantly implemented using a single `finish` provided by the *APGAS* runtime.

The network created between the places by the lifelines describes a directed graph between the places. It is necessary for this graph to be connected; otherwise, some places may never obtain new work once they run out. In addition, the graph should have a bounded out-degree, meaning the number of lifelines that individual nodes can establish should be limited. This prevents nodes from spending excessive time addressing lifeline thieves instead of focusing on computation. Moreover, the graph should possess a low diameter, i.e., any place should be reachable within just a few hops from another place. This ensures that when some places run out of work, they can quickly receive tasks from the few places that still have available work. A class of graph which fulfills these properties are the *cyclic hypercubes* which establish a lifeline between two places if their number (written in a certain base z) is within a distance $+1$ in the Manhattan distance in modulo z arithmetic [39].

Malleable Programs in APGAS

At its most fundamental level, a malleable program needs to react to *shrink* and *grow* orders sent by the batch job scheduler. In the programming model implemented by *APGAS*, this finds an intuitive translation consisting of releasing and adding places to the partitioned global address space, as illustrated in Fig. 3.

This requires *APGAS* to be able to communicate with the batch job scheduler following a certain protocol.

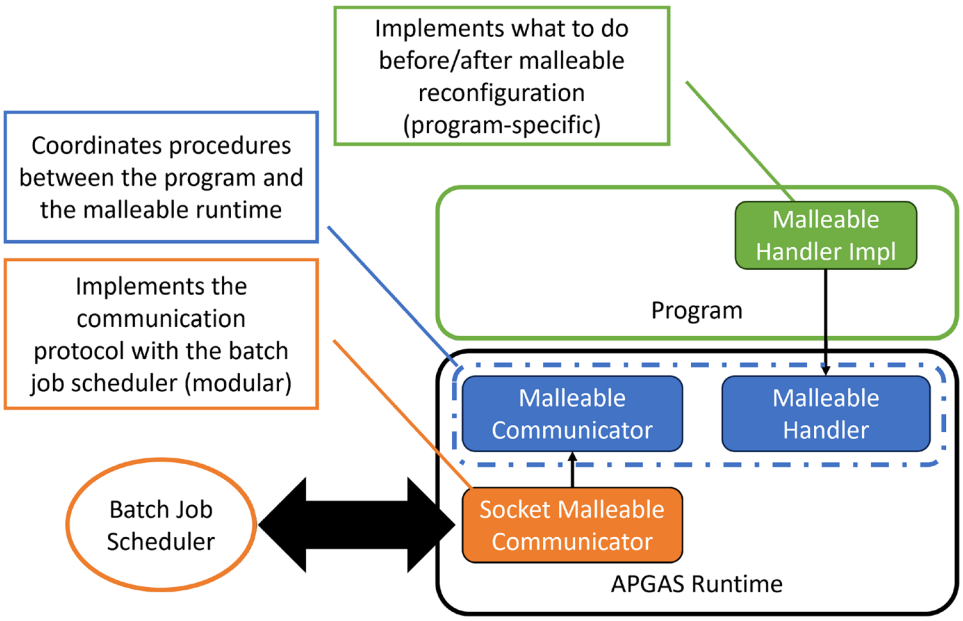

Fig. 4 Malleable *APGAS* architecture

Additionally, *APGAS* must be capable of initiate new processes and terminate processes at runtime. Finally, some level of preparation in response to changes in resources must be performed by the running user program.

To enable malleability in *APGAS*, we designed a generic framework composed of several components, as illustrated in Fig. 4. First, the `MalleableCommunicator` is responsible for communicating with the batch job scheduler. Second, the `MalleableHandler` defines the actions that need to be performed upon receiving a *shrink* or *grow* order from the batch job scheduler. As this is program-specific, programmers are expected to implement this `MalleableHandler` to specify these actions.

In the following section, we introduce our new programming abstractions accompanied by an application example (*GLB*). We then explain how *APGAS* interacts with the batch job scheduler, followed by the consequences on the programming model. The source code of our modified *APGAS* library[2] and the example application[3] are both freely available on GitHub.

Programmer Abstractions

We provide an interface for the programmers to implement the `MalleableHandler`. A single instance of this handler will be prepared on `place(0)` (this choice will be justified in the "Consequences on the Programming Model" section) and its methods automatically called by *APGAS* when the corresponding orders are received. This interface presents the following four methods for programmers to implement:

- `List<Place> preShrink(int nbPlaces)`
- `void postShrink(int nbPlaces, List<Place> removedPlaces)`
- `void preGrow(int nbPlaces)`
- `void postGrow(int nbPlaces, List<Place> continuedPlaces, List<Place> newPlaces)`

Shrinking

In the implementation of the method `preShrink`, programmers choose the places to be released and perform any preparatory steps prior to effectively terminating their corresponding processes. Typically, data and tasks must be relocated from the places to be released to the remaining ones.

APGAS only terminates the corresponding processes when the `preShrink` method has returned. This allows programmers to use all *APGAS* constructs in a stable runtime, thereby guaranteeing that all necessary preparations have completed. Upon successful process termination, the method `postShrink` is called to signal the end of the transition period. Here, programmers can perform the steps necessary for their program to resume normal execution, thus logically completing the shrink.

[2] https://github.com/ProjectWagomu/APGAS.

[3] https://github.com/ProjectWagomu/LifelineGLB.

Growing

Similar to shrinking, *APGAS* automatically calls the methods `preGrow` and `postGrow` before and after an increase in the number of places. In the method `preGrow`, the number of places to be added is passed. After new places have joined the runtime, the method `postGrow` is called. For convenience, the places that were continued/added are indicated in two distinct lists as parameters this method.

Handler Registration and De-registration

To enable malleability in a running program, programmers need to register their `MalleableHandler` implementation with the *APGAS* runtime. This is done using a new construct called `defineMalleableHandler`. Until this registration occurs, the *APGAS* program ignores any orders sent by the batch job scheduler and is effectively rigid.

We justify this design by the fact that no arbitrary program can be malleable until some level of initialization has been performed. This is true of the `MalleableHandler` in particular, as it may need specific data structures pertaining to the running program to be initialized before it can be created. Hence, we consider that an *APGAS* program is malleable from the moment the `MalleableHandler` is registered with the *APGAS* runtime. In our implementation, registering the handler causes the *APGAS* runtime to contact the batch job scheduler to inform it of the fact it can now receive and handle malleable orders (shrink and grow).

Moreover, we also account for cases in which the program is in its shutdown phase and can no longer handle malleable orders. Therefore, we provide method `disableMalleableHandler` as a new construct to allow the user to prevent the arrival of further malleable orders. Similar to the first notification that signals to the batch job scheduler the capability of the program to receive malleable orders, a de-registration notification signals the batch job scheduler that it should no longer send malleable orders to that programs. From the moment the program sends this notification, it is effectively rigid.

We do not allow programs to re-register the `MalleableHandler` after it has been de-registered. If a malleable program presents phases during which a malleable change is not possible immediately, methods `preShrink` and `preGrow` should be used to wait until the program progresses into a phase where malleable changes are possible.

Example: GLB Library

As introduced in the "Lifeline-Based Global Load Balancing" section, the Lifeline-Based Global Load

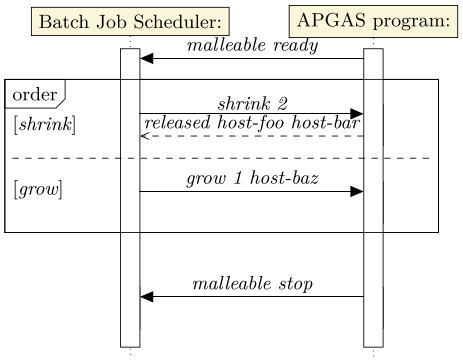

Fig. 5 Lifecycle of a malleable *APGAS* program. In this example, the program receives a shrink and a grow order. In general, a malleable program may receive multiple such orders (or none at all) depending on the decisions made by the batch job scheduler

Balancing is a fully distributed work-stealing scheme. We converted it to a malleable implementation in which places can be added or released at runtime. We build on an existing malleable *GLB* implementation [35], but adapt it to our new malleable *APGAS*, thus significantly reducing its complexity and increasing the general usability. Indeed, the `MalleableHandler` implementation for our *GLB* involves the following:

- `preShrink`: disconnect the places to be released from the lifeline work-stealing network and relocate tasks and intermediate results from the places to be released to continuing places
- `postShrink`: no operation to be performed
- `preGrow`: no operation to be performed
- `postGrow`: new places are initialized and join the lifeline work-stealing network

The `MalleableHandler` is registered using `defineMalleableHandler` when the computation begins (right before the first task is created) and the program remains malleable until the it reaches the final result reduction phase, at which point the `MalleableHandler` is de-registered using using `disableMalleableHandler`. This guarantees that the parts of the result held by each node are kept during the reduction that is occurring at that moment. Moreover, as the result reduction is usually brief, there is not merit in changing the allotment at this phase in the execution, since the program will terminate anyway, thus releasing all the nodes it was using up.

While other malleability approaches often rely on synchronization points in the user program—most commonly provided by iterative computations [24]—our combination of *GLB* and *APGAS* does not require any interruption of the computation to react to malleable orders.

Batch Job Scheduler Interactions

As explained in the "Programmer Abstractions" section, malleable programs and the batch job scheduler managing the resources of a supercomputer need to interact to successfully perform malleable changes. The lifecycle of a malleable *APGAS* program and its possible interactions with the batch job scheduler are shown in Fig. 5.

Our malleable *APGAS* architecture is illustrated in Fig. 4. Here, the `MalleableCommunicator` communicates with the batch job scheduler. While the abstract `MalleableCommunicator` we provide implements the general procedures to handle *shrink* and *grow* orders, a child class has to implement the details of the communication protocol with the batch job scheduler.

We provide the implementation `SocketMalleableCommunicator` inspired by *Prabhakaran et al.* [37]. True to its name, it uses a sockets to communicate with the batch job scheduler. The socket is opened to receive connections upon the programmer registering the `MalleableHandler` (i.e., `defineMalleableHandler` has been called). The `SocketMalleableCommunicator` then notifies the batch job scheduler that the application is now capable of receiving *shrink* and *grow* orders. The socket remains open until either the `main` method completes or the `MalleableHandler` is de-registered (i.e., `disableMalleableHandler` has been called), at which point a notification signaling the end of the malleable phase of the application is sent to the scheduler before the socket is closed.

The design of our `MalleableCommunicator` is modular, so that possible future expansion to other batch job schedulers that may use a different protocol than the one we implemented with our custom batch job scheduler. Indeed, adapting the *APGAS* runtime to work with a different batch job scheduler would only involve implementing the communication between the two; existing malleable applications and the procedures used to perform the malleable changes (adding and releasing places) need not be modified.

Shrink Orders

When the batch job scheduler sends a *shrink* order, the socket expects to receive the string `shrink` followed by the number of nodes to release. The order is then transmitted to the method `preShrink`. When the places to be released are identified, the corresponding nodes to be released are identified with an internal mapping of *APGAS*. After *APGAS* is called to shut down the corresponding processes, the nodes on which they were running are returned to the batch job scheduler through the socket connection, indicating that the may now be used by another job. The shrinking procedure is then completed by calling the method `postShrink`.

Grow Orders

Similar to *shrink* orders, for *grow* orders the batch job scheduler sends the string `grow` followed by the number of nodes added to the allotment, and the names of the nodes to spawn new processes on. The order is transmitted to the method `preGrow`. When `preGrow` returns, new processes are started on the designated nodes by the *APGAS* runtime. When all new places have successfully started, the method `postGrow` is called. Unlike shrink orders, no notification needs to be sent back to the batch job scheduler in this case.

Consequences on the Programming Model

As discussed above, malleable *APGAS* programs can add and release places at runtime. This straightforward concept has significant consequences on the APGAS programming model, challenging several seemingly intuitive assumptions.

In a regular, rigid, *APGAS* program, places are sequentially numbered from 0 to $n-1$ for an n-process execution. For a *shrink* order, the `MalleableHandler` implementation decides which places to release, potentially leading to gaps in the numbering. Also, as the ids of the released places are not reused, gaps will remain when new places are later added, as illustrated in Fig. 3.

Another consequence of changing the number of places is a potential disruption to ongoing `for loops` on participating places. Consider the `for loop` in Line 2 of Listing 1. If a change of the number of places is ongoing, such a `for loop` may try to spawn new asynchronous tasks using `asyncAt` on a place that is currently leaving the computation, resulting in an error; or places that just joined the runtime may be left out of the `for loop`.

This is the reason why we introduce four methods for programmers to specify the actions to perform before and after any malleability order. This way, programmers can prevent any ongoing `for loop` from spawning a task that would fail immediately or, more likely, ensure that any such ongoing `for loop` is allowed to complete before the methods `preShrink` or `preGrow` return and the actual malleable change occurs. In the case of *GLB*, this involves temporarily pausing inter-place stealing to ensure that places to be removed are no longer considered.

The existing *APGAS* implementation has certain limitations, notably the inability to remove `place(0)`. This is due to its unique role in runtime setup and its responsibility for executing the `main` method. This motivates our decision to assign the `MalleableCommunicator` and `MalleableHandler` roles to `place(0)`, given that this place is an immutable component of the runtime.

Another limitation is the inability for the `finish` construct to be relocated to another place. As this construct controls task termination, it is not possible to immediately

release a place containing a `finish` with pending tasks. These pending tasks need to complete for the `finish` to return, allowing the task that spawned the finish to continue and complete its execution. Only then will the place be shut down. In practice, this should be not an obstacle for applications unless they rely on many nested `finish` constructs opened on different places.

In the case of *GLB*, there are only two `finish` layers. At the top level, a single `finish` on `place(0)` controls the global completion of the computation and remains present throughout. Within this `finish`, one `finish` per place is spawned to control the completion of workers on that place. This `finish` returns when all tasks on that place has been processed; a new `finish` is opened in its place when work is received through the stealing channels. As this second `finish` only spawns local tasks, when a shrink operation occurs, simply causing the workers on that place to terminate causes this `finish` to also terminate, thus preserving the integrity of the global termination mechanism.

Evaluation

In this section, we evaluate our malleable *APGAS* implementation through real-world experiments. We assess the overhead introduced by handling shrink and grow orders (i.e., initiating and terminating processes). We then evaluate the impact of deploying malleable *APGAS* programs on supercomputers with regard to multiple metrics. For that, we execute job batches composed of real rigid/moldable/malleable programs on a 32-node cluster, adjusting the proportion of moldable/malleable jobs in the job batch from 0 to 100%.

First, we present a job scheduler prototype designed to support malleability, including communication with malleable jobs. We then detail our experimental settings. The findings and analysis related to malleable runtime performance are presented in the "Malleable Runtime Performance" section, while section "Scheduler Performance" focuses on the scheduling performance metrics of supercomputers. The experimental results are further discussed in section "Discussion".

Job Scheduler

Existing batch job schedulers currently used in production, such as Slurm [46] or Torque [41], offer no or only limited support for malleability. Thus, we developed a modular prototype of a batch job scheduler, called *ElasticJobScheduler*, that provides support for both moldable and malleable jobs. Its source code is freely available on GitHub.[4]

ElasticJobScheduler is implemented in a modular design to allow for the evaluation of job scheduling algorithms that can handle both moldable and malleable jobs. These algorithms can be easily implemented and need be selected when starting *ElasticJobScheduler*.

ElasticJobScheduler resembles most of the well-known concepts of existing batch job schedulers. Consequently, programs to be executed are submitted as jobs. While rigid jobs are submitted with an exact number of required nodes, malleable jobs are submitted with a *minimum* and a *maximum* number of nodes. For malleable jobs, the number of nodes to be used to execute this job is decided by the scheduler at job start. *ElasticJobScheduler* may further adjust the number of nodes allocated to them between their specified *minimum* and *maximum* during their execution. If the selected job scheduling algorithm only supports moldable jobs, malleable jobs are treated as moldable, i.e., even if these jobs are capable of changing the number of nodes they are running on during their execution, they use the same number of nodes throughout.

To allow for dynamic changes in the allotment of malleable jobs, communication is established between *ElasticJobScheduler* and the malleable jobs. We implemented the workflow depicted in Fig. 5 between *APGAS* and *ElasticJobScheduler*. Communication is implemented using sockets, as detailed in section "Batch Job Scheduler Interactions". Given *ElasticJobScheduler* 's modular design, it allows for future expansion in communication methods and, e.g., supporting evolving jobs.

We implemented the following four job scheduling algorithms. While the first three job scheduling algorithms—*FCFS*, *Backfilling*, and *Easy-Backfilling*—are well known in the context of rigid job scheduling, we extended them to handle malleable jobs as well. As such, malleable jobs, when scheduled by these job scheduling algorithms, are considered as moldable. *ElasticJobScheduler* executes the selected job scheduling algorithm at 5-s intervals. In addition, the job scheduling algorithm is also triggered by specific events, such as the submission or completion of jobs.

First Come First Served (FCFS)

The First Come First Served (*FCFS*) job scheduling algorithm sorts submitted jobs in a queue determined by their submission timestamp. Jobs in this queue are started sequentially as soon as their required number of nodes is available in the supercomputer. For moldable jobs, the *FCFS* starts the jobs with the maximum possible number of nodes, taking both into account the specified maximum of the job and the currently idle nodes.

[4] https://github.com/ProjectWagomu/ElasticJobScheduler.

Backfilling

Similar to *FCFS*, the *Backfilling* job scheduling algorithm sorts jobs in a queue based on submission timestamps. If the required number of nodes for the foremost job in the queue is available, the foremost job is started. If not, *Backfilling* scans the queue for smaller jobs that may start with the currently available number of nodes in the supercomputer. We applied the same moldable job extension to *Backfilling* as we did for *FCFS*.

Easy-Backfilling

The job scheduling algorithm *Easy-Backfilling* enhances *Backfilling* by incorporating a fairness mechanism. Specifically, *Easy-Backfilling* identifies smaller jobs that can run on the available nodes *without* impeding the start of the foremost job in the queue. To make this decision, the estimated amount of time needed by the job to complete execution (as specified by the user at job submission) is taken into account. The *FCFS* moldable extension was similarly applied to *Easy-Backfilling*.

Malleable-Algorithm

The job scheduling algorithm *Malleable-Algorithm* adopts the so-called *minAgree* algorithm from [36], which operates in following three steps:

1. Waiting jobs are started using *Easy-Backfilling*. Malleable jobs are always started with their minimum number of nodes.
2. If there are jobs in the queue and but not enough idle nodes to start them, nodes are extracted from running malleable jobs, allowing the start of a waiting job. The malleable jobs running with the largest number of nodes see their allotment reduced.
3. Any remaining idle nodes are assigned to running malleable jobs, with preference given to malleable jobs running with the smallest number of nodes.

Experimental Setting

Environment

To run the experiments, we used the cluster of the University of Kassel [8]. It consists of Infiniband-connected nodes, each equipped with two 24-core AMD EPYC 7443 CPUs and 256 GB of main memory. As this cluster deploys Slurm, we encapsulate each of our job batches into a Slurm job that allocates 33 nodes. One node is used to run *ElasticJobScheduler*, and 32 nodes are used as compute nodes for executing jobs. All jobs start one process per allocated node with 48 worker threads each. We used Java in version 19.0.2.

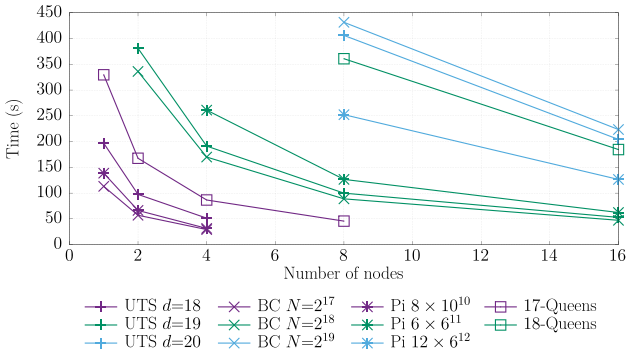

Fig. 6 Running time of the *GLB* programs when run in rigid configuration with strong-scaling from one to 16 nodes

Jobs

As described in section "Example: GLB Library", we made *GLB* malleable by deploying our proposed malleability features of *APGAS*. Consequently, we have deployed the following *GLB* programs as jobs. These programs did not need to be adapted; they are automatically malleable thanks to our added malleability in *GLB*.

- Unbalanced Tree Search (*UTS*) [30] dynamically generates a highly irregular tree at runtime. Each tree node represents a task. The result is the number of generated tree nodes.
- *N-Queens* [17] calculates the number of placements of N queens on an $N \times N$ chessboard, so that no two queens threaten one-another.
- Betweenness Centrality (*BC*) [15] calculates a centrality score for each node of a given static graph.
- *Pi* calculates the value of π based on a number of random samples using a Monte Carlo algorithm.

BC and *Pi* deploy static tasks, i.e., all tasks are known from the beginning and are therefore evenly distributed at the beginning, and no new tasks are generated at runtime. In contrast, *UTS* and *N-Queens* deploy dynamic tasks and start the computation with a single task. This single task generates new tasks at runtime, and these new tasks can in turn generate new tasks, etc. Consequently, those generated tasks are shared between workers and nodes.

In both *UTS* and *N-Queens*, the result is a single `long` value. In *Pi*, the result is a single `double` value. In contrast, in *BC*, the result is a `long` array.

To facilitate the creation of well-structured job batches, we executed each program in multiple configurations to create various job sizes.

Table 2 Job configurations and relative selection probability in the job batches

Program	Rigid: number of nodes	Relative probability	Moldable/ malleable: range of nodes
UTS $d = 18$	2	1.5	1–4
	4	0.5	1–4
UTS $d = 19$	2	1.5	1–8
	4	1	1–8
	8	1	1–8
UTS $d = 20$	8	1	8–16
	16	0.5	8–16
N-Queens $N = 17$	2	1.5	1–4
	4	1	1–4
N-Queens $N = 18$	8	1	8–16
	16	0.5	8–16
BC $N = 2^{17}$	2	1.5	1–4
	4	1	1–4
BC $N = 2^{18}$	2	1.5	1–8
	4	1	1–8
	8	0.5	1–8
BC $N = 2^{19}$	8	1	8–16
	16	0.5	8–16
Pi 8×10^{10}	2	1.5	1–4
	4	1	1–4
Pi 6×10^{11}	4	1.5	4–16
	8	1	4–16
	16	0.5	4–16
Pi 12×10^{12}	8	1	8–16
	16	0.5	8–16

- *UTS*: geometric tree shape, branching factor 4, random seed 19, tree depth d = 18, 19, 20
- *N-Queens*: $N = 17, 18$
- *BC*: random seed 2, number of graph nodes $2^{17}, 2^{18}, 2^{19}$
- *Pi*: samples $= 8 \times 10^{10}, 6 \times 10^{11}, 1.2 \times 10^{12}$.

We measured the execution times of each program in strong-scaling from 1 node up to 16 nodes for the above configurations. The measured running times in rigid configuration are reported in Fig. 6, reflecting the average of three executions per configuration.

Batches

Our objective is to construct representative job batches designed for an approximate execution duration of 15 min using 32 nodes, comprising 25 distinct jobs. Consequently, the aggregate computational demand for one job batch is defined as $32 \times 0.25 = 8$ node hours.

To distribute this 8 node hours among the 25 jobs, we have allowed flexibility in job sizes, with some jobs demanding more nodes than others. The specific configurations for these jobs, encompassing both their parameters and the allowed node range, are summarized in Table 2.

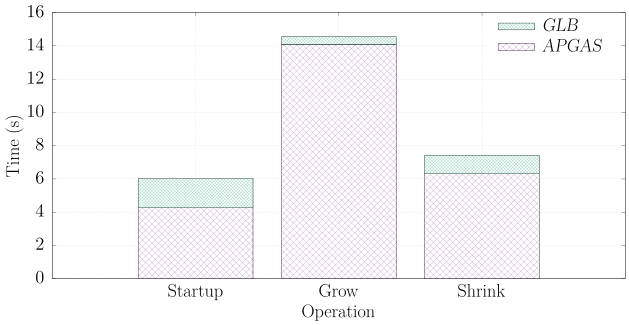

Fig. 7 Running time costs for adaption to resource changes

Relative selection probabilities were determined to facilitate the construction of job batches that closely match with the desired 8 node hours goal. To construct a job batch containing 100% rigid jobs, 25 rigid jobs are selected according to the distribution specified in Table 2, resulting in an approximate total of 8 node hours. However, due to inherent variations—particularly in cases where a job batch may contain a preponderance of larger jobs—the computational demand (in node hours) may vary between job batches.

For our study, we generated 10 distinct job batches containing 100% rigid jobs, leveraging a pseudo-random number generator initialized with 10 unique seeds.

Job batches containing malleable jobs are constructed based on a corresponding rigid job batch. To instantiate a job batch comprising, for instance, 20% (or 40%, and so on) malleable jobs, a subset—equivalent to 5 out of 25 (or 10 out of 25, respectively)—of the rigid jobs are made malleable. The batch job schedulers can choose to execute these jobs with any number of nodes within the range indicated in Table 2.

We assume that all jobs in a job batch are submitted to *ElasticJobScheduler* within the first 10 min, and the first five jobs are submitted immediately at time step 0. The timestamps for each job submission are determined stochastically, following a uniform distribution [0, 10).

In total, we conducted 240 runs: 10 job batches, each having six proportions of malleable jobs (0%, 20%, ⋯, 100%), scheduled by four job scheduling algorithms.

Malleable Runtime Performance

In this section, we examine the experimental results of our evaluation as it pertains to the performance of *APGAS* programs. We analyze the overhead caused by resource changes in malleable *GLB* programs (i.e., initiating and terminating

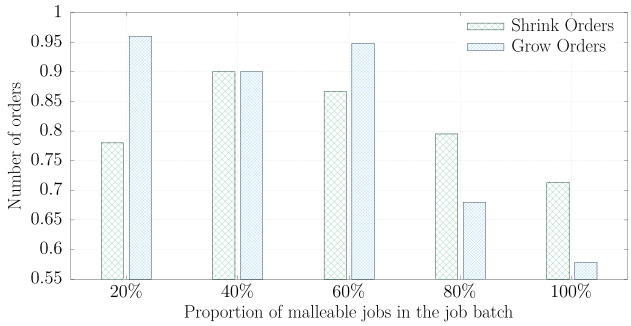

Fig. 8 Average number of shrink and grow orders

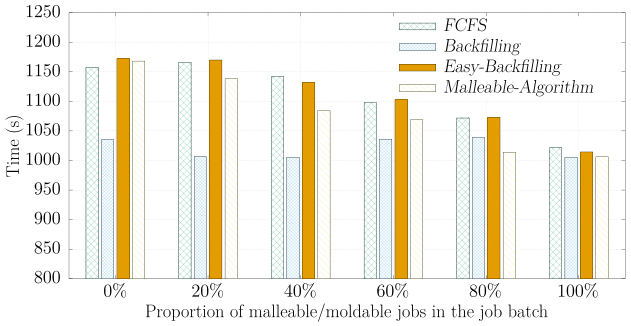

Fig. 9 Average job batch makespan depending on the proportion of malleable/moldable jobs and the job scheduling algorithm used

processes). Figure 7 presents the measured times in seconds, averaged over all malleable job executions of all job batches.

The startup of a malleable *GLB* program takes about 6 s for the program to be able to receive malleable orders. This time includes the startup time of *APGAS* (unrelated to malleability) which takes about 4.27 s. This is primarily caused by the distributed initiation of all processes/places and their inter-connection. The initialization of all *GLB* constructs takes 0.32 s. In addition, registering the `MalleableHandler` and the connection establishment with the *ElasticJobScheduler* account for the remaining 1.45 s.

Grow orders add an average of 3.8 places to the running program, which takes an average of 14.55 s. Of which, *APGAS* requires 14.09 s to initiate and connect the new processes, following which *GLB* takes only 0.46 s to logically initialize the new workers and incorporate the new places into the work-stealing network. As expected, the `preGrow` method takes 0 s, since no action need be performed, while the `postGrow` method accounts for the entire 0.46 s.

Shrink orders release an average of 2.55 places, consuming a total of 7.41 s. Of this time, *APGAS* requires 6.31 s to release and terminate the corresponding processes. *GLB* requires a mere of 1.1 s to relocate all tasks and intermediate results from the places to be released to the remaining places, and to subsequently disconnect the places to be released from the work-stealing network. As expected, shrinking is more time-consuming for *GLB* compared to growing, since the former entails the movement of tasks and intermediate results—a step absent in the latter. The method `preShrink` accounts for the totality of the 1.1 s as the method `postShrink` does not carry out any operation.

Notably, both growing and shrinking are efficiently executed at the *GLB* level. Moreover, the times indicated reflect only the times registered in the respective methods; the actual computation of the tasks and the work-stealing of the unaffected workers remain undisturbed.

The time required by the *APGAS* runtime is relatively high, primarily due to the Hazelcast [18] library it relies on to manage the communication between the processes. Thus, there is potential for enhancement in *APGAS* 's process management in future research. More efficient deployments could be achieved using tools such as PMIx [21].

Figure 8 presents the average number of shrink and grow orders received by malleable jobs averaged across all executions of all job batches for the *Malleable-Algorithm*. As the proportion of malleable jobs increases, there is a discernible shift in *shrink* and *grow* orders. For 20% malleable jobs, there are 0.78 shrink orders per malleable job and 0.96 grow orders per malleable job. However, for 100% malleable jobs, these values decline to 0.71 shrink orders per malleable job and 0.58 grow orders per malleable job. This indicates that as the system adapts to a greater number of malleable jobs it stabilizes and requires fewer malleable orders per job. Overall, the values remain low for both shrink and grow orders in all situations, settling any concern about loss of performance due to too frequent allotment changes.

Scheduler Performance

When comparing the performance of the job scheduling algorithms, there are mostly two points of view that can be adopted: that of the operator of the supercomputer and that of individual users. The former may be more concerned with the overall usage of the entire supercomputer and its capability to process jobs without focusing on the jobs of a single user. The latter will be more concerned with how long it takes to obtain the results from their jobs. We therefore split this section into two subsections to reflect these different perspectives in sections "Makespan and Node Utilization" and "Average Job Turnaround Time."

Makespan and Node Utilization

The metric *makespan* represents the overall completion time required to complete an entire job batch, including scheduling, starting, and finishing all jobs. A shorter makespan indicates more efficient job scheduling by the batch job

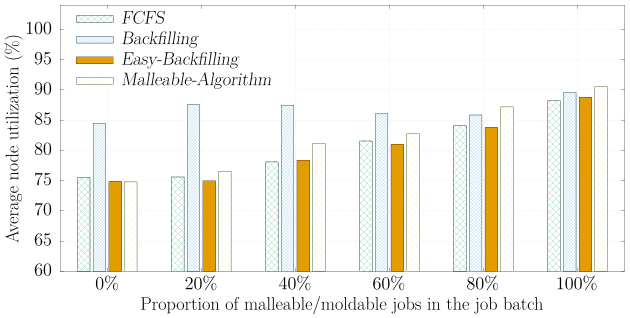

Fig. 10 Average node utilization depending on the proportion of malleable/moldable jobs and the job scheduling algorithm used

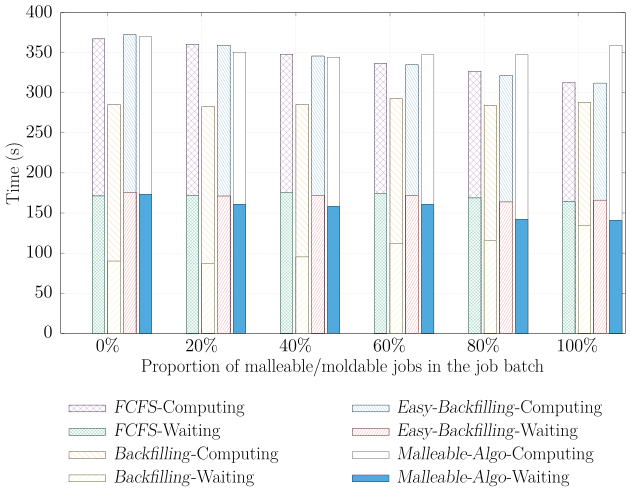

Fig. 11 Average job turnaround times

scheduler. Recall that we constructed the job batches to complete in around 15 min (or 900 s) on 32 nodes. In practice, since the jobs do not fit perfectly, there are times when nodes remain idle, resulting in longer makespans than 15 min as shown in Fig. 9.

As can be seen in Fig. 9, the job scheduling algorithm *Backfilling* shows the lowest makespan of all job scheduling algorithms. However, *Backfilling* exhibits only a marginal time decrease as the proportion of moldable jobs increases, with times of 1035.23 s for 0% moldable jobs and 1005.2 s for 100%, i.e., only a 3% improvement. Small anomalies appear for 60% moldable jobs and 80% moldable jobs where the makespan increases compared to cases where fewer jobs were moldable. This can be explained by unfortunate resource allocation or consolidation of nodes by moldable jobs. While *Backfilling* exhibits commendable performance for the metric makespan, it does not incorporate fairness considerations for jobs or users, potentially limiting its practical applicability.

The other job scheduling algorithms—*FCFS*, *Easy-Backfilling*, and *Malleable-Algorithm*—demonstrate a more pronounced makespan reduction as the proportion of malleable/moldable jobs increases. *FCFS*, being a fundamental job scheduling algorithm, improves from 1157.46 s (for 0% moldable jobs) to 1021.68 s (for 100% moldable jobs), marking a 13.3% improvement. *Easy-Backfilling*, an extension of *Backfilling* with fairness considerations, understandably lags behind *Backfilling* in terms of this metric. However, *Easy-Backfilling* improves from 1171.95 s (for 0% moldable jobs) to 1014.40 s (for 100% moldable jobs), marking a 15.5% improvement

Malleable-Algorithm is particularly interesting, since it is the only job scheduling algorithm leveraging malleable (not just moldable) jobs. For 0% malleable jobs, performance of *Malleable-Algorithm* is in line with *Easy-Backfilling* as expected (in the absence of moldable/malleable jobs, these two job scheduling algorithms are identical; minor deviations result from natural variations of real program executions). As expected, *Malleable-Algorithm* shows improved performance as the proportion of malleable jobs increases, and for 100% malleable jobs, *Malleable-Algorithm* shows the (almost) best performance even accounting for fairness across all job scheduling algorithms. *Malleable-Algorithm* starts at 1168 s for 0% malleable jobs and decreases to 1006.13 s for 100% malleable jobs, an improvement of 16.1%.

The metric *average node utilization*, depicted in Fig. 10, represents the fraction of time during which nodes are actively executing jobs. A higher utilization signifies that the nodes are being used more, potentially processing more jobs faster. This is confirmed when comparing the makespan against the average node utilization: the node utilization is clearly inversely correlated to the makespan—the shorter the makespan, the higher the node utilization.

Backfilling shows with 84.45% average node utilization the highest value for 0% moldable jobs. This superior value can again be attributed to its disregard for fairness in job scheduling. Even as the proportions of moldable jobs increase, *Backfilling* 's utilization remains commendably high, inching toward 89.54% for 100% moldable jobs. Such consistent efficiency underscores *Backfilling* 's ability to fill gaps, regardless of malleable job.

The other job scheduling algorithms—*FCFS*, *Easy-Backfilling*, and *Malleable-Algorithm*—show a consistent upward trend in average node utilization with an increasing proportion of malleable/moldable jobs. *FCFS* scales from an initial 75.52% for 0% moldable jobs to 88.18% for 100% moldable jobs, successfully leveraging the flexibility of moldable jobs, optimizing node utilization and reducing idle time of nodes *Easy-Backfilling* 's trajectory closely parallels *FCFS*. It embarks with a 74.83% average node utilization, culminating at 88.75% as the job batches become predominantly malleable/moldable. The average node utilization of *Easy-Backfilling* runs largely parallel to *FCFS*.

For 0% moldable jobs, *Easy-Backfilling* has a average node utilization of 74.83%, and for 100% moldable Jobs, it has an average node utilization of 88.75%. The increase in node utilization for *Easy-Backfilling* and *Malleable-Algorithm* confirms that the fairness mechanisms embedded into these algorithms do not unduly hamper their ability to utilize nodes efficiently.

Among all job scheduling algorithms, *Malleable-Algorithm* has the highest increase in utilization. From an average node utilization of 74.78% for 0% malleable jobs, it reaches 90.52% average node utilization for 100% moldable jobs. This is consistent with the nature of the algorithm as it is capable of of both selecting the number of nodes that moldable/malleable jobs use when starting their execution (just as both *FCFS* and *Easy-Backfilling*), and change the number of nodes of running jobs. Among the candidates we have evaluated so far, this makes *Malleable-Algorithm* the most promising overall.

Average Job Turnaround Time

The metric *average job turnaround time* refers to the time a job takes from its submission to completion, as visualized in Fig. 11. This metric is a composite of the *job waiting time*—the time the job spends in the queue waiting to be started—and the *job computation time*—the time the job needs to execute to completion—offering a comprehensive perspective on job processing efficiency from the users' perspective.

Backfilling shows a relatively constant job turnaround time, as it is 284.41 s for 0% moldable jobs and 281.86 s for 100% moldable jobs. However, as the proportion of moldable jobs increases, the job waiting times increase, because the moldable jobs are often started with more nodes than their rigid counterparts. This is also reflected in the fact that as the proportion of malleable jobs increases, the job computation time decreases.

FCFS shows consistent decrease of the average job turnaround time with increasing proportion of moldable jobs. For 0% moldable jobs, *FCFS* has a turnaround time of 366.20 s and for 100% malleable jobs of 306.31 s. This is due to the fact that the flexible node numbers of the jobs allow for better utilization of nodes, and thus, both the job waiting time and the job computation time decrease as the proportion of moldable jobs increases. *Easy-Backfilling* corresponds quite closely to the behavior of *FCFS*.

Malleable-Algorithm 's job turnaround time varies only slightly, with 368.87 s for 0% malleable jobs and 352.99 s for 100% malleable jobs. While its job waiting time decreases from 173.51 to 140.67 s, its job computation time increases from 195.36 to 212.31 s. Both are caused by the fact that malleable jobs do not only run with more nodes than their rigid counterparts but also with fewer nodes. Moreover, the times needed by the programs to react malleable orders are also taken into account.

Discussion

The experimental results reinforce the anticipated correlation between the proportion of moldable/malleable jobs and job scheduling flexibility, revealing a positive impact on supercomputer performance across various metrics.

Notably, while *Backfilling* frequently yielded commendable outcomes, its practical relevance is diminished by its disregard for fairness. Consequently, it has been excluded from subsequent discussions. The synergy between *Malleable-Algorithm* and 100% malleable *GLB* programs utilizing our malleable *APGAS* runtime led to significant enhancements in regard to following metrics:

- *Makespan*: 13.09% decrease compared to the top-performing algorithm for 0% malleable jobs (*FCFS*)
- *Node utilization*: increase of 19.86% compared to the top-performing algorithm for 0% malleable jobs (*FCFS*)
- *Job turnaround time*: decrease of 3.61% compared to the top-performing algorithm for 0% malleable jobs (*FCFS*)

We should, however, introduce a word of caution concerning node utilization. Indeed, in the context of malleable job scheduling, this metric should not be seen as a primary objective to maximize, but rather as a reflection of the reduced makespan and job turnaround time. This is because there are limits to how much additional nodes can benefit the performance of parallel programs—as modeled by Amdhal's law [20] for instance. Increasing the number of nodes of running jobs beyond reason would increase node utilization without actually bringing about any benefit. In our evaluation, the range of allowable nodes for each job were all chosen within a reasonable range, i.e., within the range where the program scales strongly (see Fig. 6 and Table 2). Under these conditions, the node utilization is a reasonable indicator of the job scheduling algorithm performance.

Overall, our results emphasize the potential benefits of adopting malleable job scheduling algorithms to better harness the processing power of supercomputers. In this work, we have only implemented one relatively simple existing malleable job scheduling algorithm. In future work, we expect to draw further benefits by adopting more complex malleable job scheduling algorithms.

As discussed in section "Batch Job Scheduler Interactions," our current implementation of communication between *APGAS* and *ElasticJobScheduler* is based on sockets. To enable compatibility of *APGAS* with other batch job schedulers that support malleability, only the `MalleableCommunicator` component would need to be modified to allow existing applications to be ported to this

new batch job scheduler. The converse is also true; to enable compatibility of *ElasticJobScheduler* with other parallel programming runtimes that support malleability, only a new communication method for *ElasticJobScheduler* needs to be implemented. Until evolving and malleable jobs are natively supported in batch job schedulers, encapsulating these jobs inside a larger one managed by *ElasticJobScheduler* appears to be a viable option for experimentation.

Related Work

While malleability has not yet been established in everyday supercomputing, several approaches have been proposed, e.g., [2, 16]. Each technique has distinct characteristics and may vary in effectiveness and programmer productivity impact. Typically, malleability solutions tailored to specific applications or fields prove more efficient.

The well-known *checkpoint/restart* technique can enable malleability for applications. Regularly storing the running application's state as a checkpoint allows for resource adjustments due to both unforeseen hardware failures or malleability orders [19, 40]. However, since it is necessary to write checkpoints as well as termination and restart the application, this results in a significant reconfiguration penalty. In-memory checkpoint writing can mitigate this penalty [48] by bypassing the distributed file system. While checkpoint/restart can be semi-transparently implemented at the user level, it still requires adaptations of user codes [25, 27]. In contrast, our proposed malleable *APGAS* does not require any checkpointing nor restarting, and facilitates programmer productivity through clear and easy-to-use programming abstractions.

Programming systems supporting user-level malleability are still rare, with a few exceptions such as ULFM [5] and X10 [22, 43]. In ULFM, procedures render the use of a communicator impossible upon the discovery that a process has failed. Procedures to then reconstruct a new communicator containing one less process are then undertaken before the program can resume execution. These programming constructs are intentionally rather low-level and should be considered as anchor points for fault-tolerant libraries to to plug into. While *APGAS* initially supported changing the number of places at runtime, it did so in a rudimentary manner [42] before it was improved only in the context of *GLB* [35]. In this work, we disentangles the intertwined malleability of *APGAS* and *GLB*, further refined the malleability concept and improved the usability significantly.

Malleable algorithm research has mainly focused on iterative computations offering natural synchronization points for application adaptation [24]. While our malleable *APGAS* requires few user code additions, it is not limited to iterative computations, but is ideally suitable for dynamic and irregular workloads.

Another malleability approach rather close to ours is that of parallel programming system Charm++ [1]. Contrary to the PGAS programming model, which exposes the distributed nature of the computation to the programmer, the programming model adopted by Charm++ hides this nature to the programmer. Instead, a Charm++ program is composed of multiple independent objects called *chares*, on which methods can be called by sending an asynchronous *message* from one chare to another. The messages received by a chare are processed sequentially by the processing element which hosts this chare. New chares can be created dynamically at runtime, and relocated between processing elements. As a result, converting existing application codes so that they become malleable can be done without significant code modifications, simply by activating the internal load balancing strategy [37].

DMRlib [24] proposes a common interface for MPI to support malleability by automating data redistribution and hide the reconfiguration internals. Both the DMRlib approach and that of Charm++ were combined with a customized version of a batch job scheduler, Slurm [46] and Torque [41], respectively, whereas we resorted to implementing a prototype batch job scheduler for the purpose of the experimental evaluation presented in this work. One next significant hurdle to overcome in providing support for malleable jobs will consist in integrating support in batch job schedulers used in current production systems, such as Slurm, Torque, or others. As mentioned above, some attempts have been made and some efforts are ongoing, but, understandably, they often focus on a single distributed runtime systems and batch job scheduler pair at a time.

Recent work by *Huber et. al.* made significant progress in this direction, adding new programming constructs to MPI [21] by extending the concept of MPI sessions as introduced as part of MPI 4.0 [26]. Requests for additional nodes coming from a running evolving program are transmitted to a slightly extended version of PMIx [33]. Our malleable *APGAS* runtime should be capable of accessing these new interfaces of PMIx thanks to the modular design of our `MalleableCommunicator`. Also, as current batch job systems already rely on PMIx to control some aspects of running jobs, they would only need to be modified to interact with running elastic jobs through the PMIx interface.

Once the challenge of communication between jobs and batch job schedulers is solved, the challenge of finding a common interaction protocol between the batch job schedulers and jobs will have to be resolved. One main challenge is that a multitude of distributed runtime systems that support elasticity now exist, and such a hypothetical standard will have to be sufficiently generic to guarantee program portability from a batch system to another. While

the 3-phase *Startup/Malleable/Termination* we propose is sufficient for the application we presented here, it lacks consideration for cases of program or hardware failures.

AMT's malleability potential remains largely untapped. Prior research has explored worker addition in a resilient *GLB* variant in X10 [6], but lacked a shrinking protocol. A multi-worker *GLB* variant in *APGAS* enabled both shrinking and growing [14, 35]. We adapted this *GLB* variant and several benchmarks to our new malleable *APGAS* to demonstrate its ease of use. However, while we used *GLB* as an application, our malleability is not limited to *GLB* but extends to a more generic parallel programming system, *APGAS*. To the best of our knowledge, no research has proposed such an easy-to-use malleable AMT.

While we were able to run our experiments using a real supercomputer and real programs for the purposes of our evaluation, this approach has its limits. First and foremost, the job submission patterns on supercomputers are known to follow certain patterns and vary at multiple time scales [12], i.e., depending on the hour of the day, the day of week, etc. Studying the behavior of job scheduling algorithms should, therefore, be done on such time scales. Moreover, a large number of job batches generated with different seeds should be run to be able to draw general trends. This makes actually running malleable workloads on larger systems with larger number of nodes completely impractical as it would require a long time and significant computing resources, that would be better used by other "real" jobs. The only viable option is to resort to simulation.

The recent release of *Elastisim* [32] makes this possible. This simulator allows users to describe a supercomputer and job submission patterns using simple JSON files. Jobs are assigned configurable models that describe the behavior of a program such as its different computation and communication phases, the amount of computation to perform, and amount of data to exchange. The job scheduling algorithm is a modular component of the simulator and can be implemented in Python, allowing easy comparison between different job scheduling algorithms [36]. Clearly, the reliability of the results obtained through simulations depends in part on the accuracy of the job models used for the simulations. The construction of a model of our *GLB* program for use in Elastisim simulations is a prospect for future work.

Conclusion

In this work, we have proposed a malleability extension for an existing AMT system, namely *APGAS*, which allows applications to easily adapt the node allotments by minimal additions to the user code. Runtime adjustments, such as process initiation and termination, are automatically managed by our malleable *APGAS*. The practical usability of our malleable *APGAS* was validated by adapting the *GLB* library including a collection of its benchmarks.

Furthermore, we showed the seamless integration potential of our malleable *APGAS* with potential future batch job schedulers, demonstrated by our newly developed prototype batch job scheduler, i.e., *ElasticJobScheduler*. For *ElasticJobScheduler*, we implemented four distinct job scheduling algorithms, with three handling moldable jobs and one specifically designed for handling malleable jobs.

For evaluations, we conducted comprehensive real-world experiments involving executing batches of rigid, moldable, and malleable jobs. The experimental results show that both shrinking and growing cause only a small running time overhead. Notably, we observed significant improvements in supercomputer performance metrics, including a 13.09% makespan reduction, a 19.86% increase in node utilization, and a 3.61% decrease in job turnaround time.

Future work could explore the development of *evolving* applications that autonomously initiate shrink/grow requests, rather than relying on the orders initiated by the batch job scheduler. However, determining the optimal timing and conditions for such requests, as well as how batch job schedulers should respond to these requests (given that it may not be possible to grant all grow requests) remains an open question.

The question of fairness between users of elastic jobs and rigid jobs also remains open. As demonstrated in numerous articles about the integration of elastic jobs in a batch system, elastic jobs bring various benefits for the entire supercomputer in terms of node utilization, or job turnaround times, etc. Nevertheless, the development of elastic programs requires more programming effort compared to the conventional rigid programs. Once the question of supporting elastic jobs is solved, the question of how to create the appropriate incentive for users to create elastic workloads will become prevalent. Without the adequate incentive, users of supercomputers may not make the effort to create elastic programs.

Author Contributions The authors are listed in order of contribution to this article.

Funding Open Access funding enabled and organized by Projekt DEAL. This research received no funding.

Data availability The entirety of the source code of the programs used in this paper is available online:

APGAS: https://github.com/ProjectWagomu/APGAS/releases/tag/v0.0.2, https://doi.org/10.5281/zenodo.10495541.

GLB: https://github.com/ProjectWagomu/LifelineGLB/releases/tag/v0.0.2, https://doi.org/10.5281/zenodo.10495547.

ElasticJobScheduler: https://github.com/ProjectWagomu/ElasticJobScheduler/releases/tag/v0.1, https://doi.org/10.5281/zenodo.10495534.

The experimental data supporting the evaluation are publicly available online:

https://github.com/ProjectWagomu/ArtefactSNCS24/releases/tag/v0.1, https://doi.org/10.5281/zenodo.10495532.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Ethics Approval This research does not contain any studies with human participants or animals performed by any of the authors.

Informed Consent Informed consent was obtained from all individual participants included in this research.

References

1. Acun B, Gupta A, Jain N, et al. Parallel programming with migratable objects: Charm++ in practice. In: International conference for high performance computing, networking, storage and analysis (SC). IEEE; 2014. p. 647–658. https://doi.org/10.1109/SC.2014.58.
2. Aliaga JI, Castillo M, Iserte S, et al. A survey on malleability solutions for high-performance distributed computing. Appl Sci. 2022. https://doi.org/10.3390/app12105231.
3. Almasi G. PGAS (Partitioned global address space) languages. Berlin: Springer; 2011. p. 1539–45. https://doi.org/10.1007/978-0-387-09766-4_210.
4. Bachan J, Baden SB, Hofmeyr S, et al. UPC++: a high-performance communication framework for asynchronous computation. In: International parallel and distributed processing symposium (IPDPS). IEEE; 2019. p. 963–973. https://doi.org/10.1109/IPDPS.2019.00104.
5. Bland W, Bouteiller A, Herault T, et al. Post-failure recovery of MPI communication capability: design and rationale. Int J High Perform Comput Appl. 2013;27(3):244–54. https://doi.org/10.1177/1094342013488238.
6. Bungart M, Fohry C. A malleable and fault-tolerant task pool framework for X10. In: Proceedings international conference on cluster computing. IEEE; 2017. https://doi.org/10.1109/cluster.2017.27.
7. Charles P, Grothoff C, Saraswat V, et al. X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Notices. 2005;40(10):519–38. https://doi.org/10.1145/1103845.1094852.
8. Competence Center for High Performance Computing in Hessen (HKHLR). Linux Cluster Kassel. 2023. https://www.hkhlr.de/en/clusters/linux-cluster-kassel.
9. De Wael M, Marr S, De Fraine B, et al. Partitioned global address space languages. Comput Surv. 2015. https://doi.org/10.1145/2716320.
10. El-Ghazawi T, Smith L. UPC: unified parallel C. In: Proceedings international conference on high performance computing, networking, storage and analysis (SC). ACM; 2006. https://doi.org/10.1145/1188455.1188483.
11. Feitelson DG, Rudolph L . Toward convergence in job schedulers for parallel supercomputers. In: Job scheduling strategies for parallel processing. Springer, p. 1–26. https://doi.org/10.1007/bfb0022284.
12. Feitelson DG, Tsafrir D, Krakov D. Experience with using the parallel workloads archive. J Parallel Distrib Comput. 2014;74(10):2967–82. https://doi.org/10.1016/j.jpdc.2014.06.013.
13. Finnerty P, Kamada T, Ohta C. Self-adjusting task granularity for global load balancer library on clusters of many-core processors. In: Proceedings international workshop on programming models and applications for multicores and manycores. ACM; 2020. p. 1–10. https://doi.org/10.1145/3380536.3380539.
14. Finnerty P, Kamada T, Ohta C. A self-adjusting task granularity mechanism for the Java lifeline-based global load balancer library on many-core clusters. Concurr Comput Pract Exp. 2021. https://doi.org/10.1002/cpe.6224.
15. Freeman LC. A set of measures of centrality based on betweenness. Sociometry. 1977;40(1):35. https://doi.org/10.2307/3033543.
16. Galante G, da Rosa Righi R. Adaptive parallel applications: from shared memory architectures to fog computing. Clust Comput. 2022;25(6):4439–61. https://doi.org/10.1007/s10586-022-03692-2.
17. Gik EJ (1987) Schach und Mathematik. 1st ed. Thun.
18. Hazelcast Unified Real-Time Data Platform for Instant Action. 2023. http://hazelcast.org.
19. Herault T, Robert Y. Fault-tolerance techniques for high-performance computing. Berlin: Springer; 2015. https://doi.org/10.1007/978-3-319-20943-2.
20. Hill MD, Marty MR. Amdahl's law in the multicore era. Computer. 2008;41(7):33–8. https://doi.org/10.1109/MC.2008.209.
21. Huber D, Streubel M, Comprés I, et al. Towards dynamic resource management with MPI sessions and PMIx. In: European MPI users' group meeting. ACM; 2022. https://doi.org/10.1145/3555819.3555856.
22. IBM. Elastic X10. 2014. http://x10-lang.org/documentation/practical-x10-programming/elastic-x10.html.
23. IBM The X10 Programming Language. 2021. https://github.com/x10-lang.
24. Iserte S, Mayo R, Quintana-Ortí ES, et al. DMRlib: easy-coding and efficient resource management for job malleability. Trans Comput. 2021;70(9):1443–57. https://doi.org/10.1109/tc.2020.3022933.
25. Maghraoui KE, Desell TJ, Szymanski BK, et al. Dynamic malleability in iterative MPI applications. In: International symposium on cluster computing and the grid. IEEE; 2007. https://doi.org/10.1109/ccgrid.2007.45.
26. Message Passing Interface Forum. MPI: a message-passing interface standard Version 4.0. 2021. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.
27. Moody A, Bronevetsky G, Mohror K, et al. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: International conference for high performance computing, networking, storage and analysis (SC). IEEE; 2010. https://doi.org/10.1109/sc.2010.18.
28. Nowicki M, Bała P. Parallel computations in Java with PCJ library. In: Proceedings international conference on high performance computing simulation (HPCS). IEEE; 2012. p. 381–387. https://doi.org/10.1109/HPCSim.2012.6266941.
29. Numrich RW, Reid J. Co-Arrays in the next Fortran Standard. SIGPLAN Fortran Forum. 2005;24(2):4–17. https://doi.org/10.1145/1080399.1080400.

30. Olivier S, Huan J, Liu J, et al. UTS: an unbalanced tree search benchmark. In: Languages and compilers for parallel computing (LCPC). Springer; 2006. p. 235–250. https://doi.org/10.1007/978-3-540-72521-3_18.
31. OpenSHMEM Application Programming Interface. 2020. http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.5.pdf.
32. Özden T, Beringer T, Mazaheri A, et al. ElastiSim: a batch-system simulator for malleable workloads. In: Proceedings of the international conference on parallel processing (ICPP). ACM; 2023. https://doi.org/10.1145/3545008.3545046.
33. PMIx Administrative Steering Committee. Process management interface for exascale (PMIx) Standard 4.0. 2020. https://pmix.github.io/uploads/2020/12/pmix-standard-v4.0.pdf.
34. Posner J, Fohry C. Cooperation vs. coordination for lifeline-based global load balancing in APGAS. In: Proceedings of workshop on X10. ACM; 2016. p. 13–17. https://doi.org/10.1145/2931028.2931029.
35. Posner J, Fohry C. Transparent resource elasticity for task-based cluster environments with work stealing. In: International conference on parallel processing workshop. ACM; 2021. p. 1–10. https://doi.org/10.1145/3458744.3473361.
36. Posner J, Hupfeld F, Finnerty P. Enhancing supercomputer performance with malleable job scheduling strategies. In: Proceedings Euro-Par parallel processing workshops (PECS). Springer; 2023 **(to appear)**.
37. Prabhakaran S, Neumann M, Rinke S, et al. A batch system with efficient adaptive scheduling for malleable and evolving applications. In: Proceedings international parallel and distributed processing symposium. 2015. p. 429–438. https://doi.org/10.1109/IPDPS.2015.34.
38. Saraswat V, Almasi G, Bikshandi G, et al. The asynchronous partitioned global address space model. In: Proceedings SIGPLAN workshop on advances in message passing (AMP). ACM; 2010.
39. Saraswat VA, Kambadur P, Kodali S, et al. Lifeline-based global load balancing. In: Proceedings principles and practice of parallel programming. ACM; 2011. p. 201–212. https://doi.org/10.1145/1941553.1941582.
40. Shahzad F, Wittmann M, Kreutzer M, et al. A survey of checkpoint/restart techniques on distributed memory systems. Parallel Process Lett. 2013. https://doi.org/10.1142/s0129626413400112.
41. Staples G. TORQUE resource manager. In: Proceedings international conference on high performance computing, networking, storage and analysis (SC). ACM, New York, NY, USA; 2006. https://doi.org/10.1145/1188455.1188464.
42. Tardieu O. The APGAS library: resilient parallel and distributed programming in Java 8. In: Proceedings of the ACM SIGPLAN workshop on X10. ACM; 2015. p. 25–26. https://doi.org/10.1145/2771774.2771780.
43. Tardieu O, Herta B, Cunningham D, et al. X10 and APGAS at Petascale. In: Proceedings principles and practice of parallel programming. ACM; 2014. p. 53–66. https://doi.org/10.1145/2555243.2555245.
44. Yamashita K, Kamada T. Introducing a multithread and multistage mechanism for the Global Load Balancing Library of X10. J Inf Process. 2016;24(2):416–24. https://doi.org/10.2197/ipsjjip.24.416.
45. Yelick KA, Semenzato L, Pike G, et al. Titanium: a high-performance Java Dialect. Concurr Pract Exp; 1998. 10(11–13):825–836. https://doi.org/10.1002/(SICI)1096-9128(199809/11)10:11/13%3C825::AID-CPE383%3E3.0.CO;2-H
46. Yoo AB, Jette MA, Grondona M. SLURM: simple Linux utility for resource management. In: Job scheduling strategies for parallel processing (JSSPP). Springer; 2003. p. 44–60. https://doi.org/10.1007/10968987_3.
47. Zhang W, Tardieu O, Grove D, et al. GLB: lifeline-based global load balancing library in X10. In: Proceedings workshop on parallel programming for analytics applications (PPAA). ACM; 2014. p. 31–40. https://doi.org/10.1145/2567634.2567639.
48. Zheng G, Ni X, Kale LV. A scalable double in-memory checkpoint and restart scheme towards exascale. In: Proceedings international conference on dependable systems and networks workshops (DSN). IEEE; 2012. https://doi.org/10.1109/dsnw.2012.6264677.

Authors and Affiliations

Patrick Finnerty[1] · Jonas Posner[2] · Janek Bürger[2] · Leo Takaoka[1] · Takuma Kanzaki[1]

✉ Patrick Finnerty
finnerty.patrick@fine.cs.kobe-u.ac.jp

✉ Jonas Posner
jonas.posner@uni-kassel.de

Janek Bürger
janekbuerger@outlook.de

Leo Takaoka
takaoka@fine.cs.kobe-u.ac.jp

Takuma Kanzaki
kanzaki@fine.cs.kobe-u.ac.jp

[1] Kobe University, Rokkodai-cho 1-1, Kobe, Hyogo 657-0013, Japan

[2] University of Kassel, Wilhelmshoeher Allee 73, 34121 Kassel, Hessen, Germany