GRANNY: Granular Management of Compute-Intensive Applications in the Cloud

Carlos Segarra[†] Simon Shillaker[†] Guo Li[†] Eleftheria Mappoura[†]
Rodrigo Bruno[⋆] Lluís Vilanova[†] Peter Pietzuch[†]
[†]*Imperial College London* [⋆]*INESC-ID, Instituto Superior Técnico, University of Lisbon*

Abstract

Parallel applications are typically implemented using multi-threading (with shared memory, e.g., OpenMP) or multi-processing (with message passing, e.g., MPI). While it seems attractive to deploy such applications in cloud virtual machines (VMs), existing cloud schedulers fail to manage such applications efficiently: they cannot scale multi-threaded applications dynamically when more CPU cores in a VM become available, and they cause fragmentation over time due to the static allocation of multi-process applications to VMs.

We describe GRANNY, a new distributed runtime that enables the fine-granular management of multi-threaded/process applications in cloud environments. GRANNY supports the vertical scaling of multi-threaded applications within a VM and the horizontal migration of multi-process applications between VMs. GRANNY achieves both through a single WebAssembly-based execution abstraction: *Granules* can execute application code with thread or process semantics and allow for efficient snapshotting. GRANNY scales up applications by adding more Granules, and de-fragments applications by migrating Granules between VMs. In both cases, it launches new Granules from snapshots efficiently. We evaluate GRANNY with dynamic scheduling policies and show that, compared to current schedulers, it reduces the makespan for OpenMP workloads by up to 60% and the fragmentation for MPI workloads by up to 25%.

1 Introduction

Compute-intensive parallel applications are common in many domains including machine learning [23], weather forecasting [44], hydrodynamics [35], genomics [5], simulation, and modeling [43]. These applications exploit massive parallelism and are built using multi-threaded or multi-process programming models, such as OpenMP [36] and MPI [33] that utilize the CPU cores within a node and across nodes.

To satisfy their resource needs, such applications are increasingly deployed on cloud-hosted clusters with a high total virtual CPU (vCPU) core count made up of virtual machines (VMs). These cloud clusters are managed by cloud resource schedulers, e.g., Slurm [48] or Azure/AWS/Google Batch [2, 16, 27], which monitor a queue of submitted applications and allocate them to VMs and vCPUs.

Existing cloud resource schedulers, however, cannot change an application's resource allocation after it has started executing. This lack of *elasticity* prevents multi-threaded applications from leveraging new cloud resources as they become available, e.g., when other applications have released vCPUs. This reduces utilization: our experiments show that, when deploying multi-threaded (OpenMP) applications, Azure Batch and Slurm consistently leave 60% and 40% of vCPUs idle, respectively, even when there are pending applications to be scheduled (§6.3). No matter how effective a resource manager's bin-packing approach is, it cannot completely avoid idle resources.

For multi-process applications, high utilization can be achieved by allocating resources at a fine granularity (e.g., allocating vCPUs to applications as soon as they become available), but this leads to *resource fragmentation*. If multi-process applications become fragmented across many VMs, it increases their network communication, thus reducing overall application performance.

Cloud resource schedulers handle this tension between utilisation and compute/data locality differently: when executing distributed multi-process applications, the Azure Batch [27] scheduler allocates resources at a VM granularity. This achieves good locality at the cost of resource utilization, because idle vCPUs in VMs cannot be used to execute other applications; in contrast, Slurm [48], another popular scheduler, allocates resources at vCPU granularity. It exhibits high utilization, but incurs fragmentation.

While high utilization is desirable for cloud providers, low fragmentation improves application performance. We observe that an ideal cloud resource scheduler for parallel applications must navigate this trade-off by *elastically scaling* and *migrating* applications at a fine granularity (i.e., individual vCPUs) at runtime. Existing runtimes for multi-threaded and multi-process applications in cloud environments, however, do not support such fine-granular management.

We describe **Granny**, a distributed runtime for executing unmodified multi-threaded/multi-processing (OpenMP/MPI) applications in cloud environments, which supports *vertical scaling* of multi-threaded application and *horizontal migration* of distributed multi-process applications. Granny achieves both with the help of a new execution abstraction called *Granules*. It makes the following contributions:

(1) Granules for thread/process execution. Granny executes each application as a set of Granules (§3.2). Each Granule contains application code with a single execution thread, capable of executing with thread (i.e., for multi-threaded applications) or process (i.e., for multi-process applications) semantics. A Granule executes a sandboxed WebAssembly [17] module, so that all Granules on a VM execute safely within a single host process. This allows Granules to implement an efficient snapshotting operation, share memory directly by mapping pages, and exchange messages with low overhead.

A Granule's execution state can be captured completely as part of a *snapshot* (§3.3). A Granule's snapshot contains its WebAssembly linear memory, with its static data sections, stack and heap, as well as other execution state, such as stack pointers, function tables, messaging queues, and file descriptors. Since their execution state is self-contained, Granules, unlike processes or containers [60], can be snapshotted and migrated robustly and with low performance impact.

(2) Granule elasticity and migration. Granny leverages Granules to perform management actions: it can (a) spawn a new Granule with thread semantics to scale a multi-threaded application (§4.2); or (b) change the distribution of a multi-process application by migrating the snapshot of a Granule with process semantics to a different VM (§4.3).

When performing the above management actions, Granny must not break the consistency of a multi-threaded/process application. Granny therefore only controls Granules when their execution reaches well-defined *control points* (§4.1), such as certain system calls or OpenMP/MPI calls. At these control points, a Granule is guaranteed to have a consistent state with respect to its shared data and messages, and it is safe to spawn new Granules, or snapshot and migrate them.

(3) Granule-aware scheduling policies. Using Granule's support for vertical scaling and horizontal migration, Granny implements dynamic scheduling policies for fine-granular management of compute-intensive applications in the cloud (§5). The policies enable (i) multi-process (MPI) applications to migrate processes between VMs to reduce inter-VM communication and thus improve application performance (§5.1); (ii) multi-threaded (OpenMP) applications to launch extra threads, increasing their parallelism when further vCPUs become available at runtime on a VM (§5.2); and (iii) checkpoint applications on spot VMs [7] before they are evicted and migrate them to replacement VMs (§5.3).

In our experiments on a 32-VM cluster, we execute a workload of existing OpenMP/MPI applications based on a 100 job trace. We compare Granny's execution to that with Azure Batch [27] and Slurm [48]. Azure Batch and Slurm use the OpenMPI [39] and LLVM OpenMP (`libomp`) [26] runtimes to execute MPI and OpenMP jobs, respectively.

We show that, by defragmenting multi-process applications using horizontal migration, Granny reduces end-to-end execution time (makespan) by 20% and fragmentation by 25% (§6.2). By elastically scaling multi-threaded applications to use idle CPU cores, Granny reduces makespan by 60% and the tail job completion time by 50% (§6.3). Finally, when using spot VMs, Granny reduces makespan by 50% (§6.4).

2 Compute-Intensive Workloads in Cloud

Next, we discuss compute-intensive parallel applications (§2.1), their runtime support (§2.2), describe associated schedulers (§2.3), and explain why existing schedulers fail to manage these applications efficiently in cloud settings (§2.4).

2.1 Compute-intensive applications

Compute-intensive applications include large-scale data analytics [61], video processing [4], and deep learning training [52], and also typical high performance computing (HPC) workloads, such as fluid dynamics [35], molecular simulation [47], and weather forecasting [44]. These applications require plentiful hardware resources, because they parallelize computation by distributing it across many CPU cores, both within nodes and across nodes.

To handle large problem sizes without increasing execution time or exhausting memory, compute-intensive applications make use of vertical *scale-up* and horizontal *scale-out* patterns. These are typically implemented using multi-threading (with shared memory among threads) and/or multi-processing (with distributed message passing between processes located on different nodes), as offered by programming models such as OpenMP [36] and MPI [33]. In this work, we focus on OpenMP and MPI, as representative examples of APIs and runtimes for parallel applications. They also benefit from long-standing popularity and widespread adoption.

2.2 Runtimes for multi-threading/processing

Multi-threaded/process applications typically require *runtime* support. Multi-threaded runtimes such as LLVM's OpenMP runtime (`libomp`) [26] manage threads, coordinate access to shared memory, and provide synchronization primitives; multi-processing runtimes such as OpenMPI [39] provide APIs for inter-node message passing, message synchronization, and data partitioning.

Internally, these runtimes use OS primitives to implement their APIs. In Linux, most OpenMP implementations use POSIX threads and standard synchronization primitives, such as mutexes and locks. MPI implementations use the OS kernel's TCP/IP stack to send and receive messages between processes reliably. Runtimes typically do not restrict other OS system calls e.g., for file system access. In this case, however, it becomes the responsibility of the developer to access the underlying resources in a race-free manner, as the runtime's coordination and synchronization guarantees do not apply.

Depending on whether an application uses multi-threading or multi-processing, application state is handled differently. For multi-threaded applications, application state resides in the process' address space, but may be divided in different thread's stacks and local storage (TLS). The state associated with synchronization primitives, e.g., mutexes and locks, resides in the OS kernel. For multi-process application, the state is partitioned across all processes. In addition, the state of the network stack is spread across all VMs involved in the distributed computation.

2.3 Resource scheduling for applications

Compute-intensive applications are deployed on large shared clusters (either on-premise or in the cloud) with high CPU core and node counts. Users submit applications as jobs to a work queue managed by a resource *scheduler*. The scheduler allocates applications to compute resources (i.e., CPU cores) according to an application's demand for parallelism. Multi-threaded/process applications specify their resource needs at deployment time, e.g., using `mpirun`'s `np` flag [40], or the `OMP_NUM_THREADS` environment variable [37].

Cloud providers use a range of resource schedulers to execute compute-intensive parallel applications, such as AWS Batch [2], Azure Batch [27], and Google Batch [16]. These schedulers control a pool of cloud VMs that can be scaled up or down on-demand. They schedule jobs once sufficient VMs are available, and idle vCPU cores in VMs cannot be allocated to other pending jobs.

In addition, some cloud providers also support general-purpose schedulers such as Slurm [48], KubeBatch [21], or Volcano [53], which are often used for on-premise clusters. These schedulers support more fine-grained policies in which they allocate applications to individual vCPU cores.

2.4 Challenges in cloud scheduling

While cloud providers strive for high utilization, cloud users want to combine high parallelism with locality, as this reduces avoidable inter-VM communication. Cloud schedulers thus try to optimize both for resource utilization and compute/data locality [20, 49], but they struggle to achieve this goal: the allocation of a multi-threaded application to a VM determines its parallelism based on the number of available vCPUs. If the scheduler cannot perfectly bin-pack multi-threaded applications to VMs, vCPUs remain idle, decreasing utilization.

Similarly, a scheduler must allocate the processes of a distributed multi-process application to sufficiently many VMs that offer the required total number of vCPUs. This potentially reduces locality, as it may spread the allocation across multiple VMs, depending on their vCPU availability.

Ideally, the scheduler should be able to *change* this initial allocation of a parallel application at the granularity of individual vCPUs. In particular, it should (a) add more threads to a multi-threaded application when vCPUs become available, increasing utilization; and (b) move processes to VMs that already contains other processes of the same application, reducing fragmentation and improving locality.

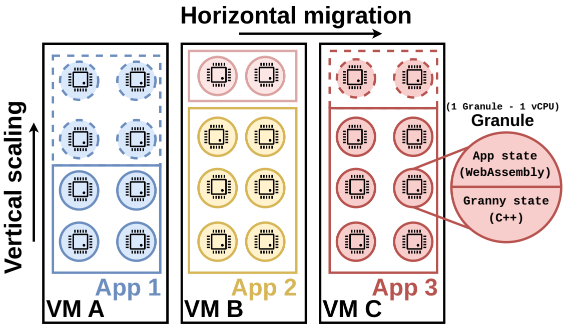


Figure 1: GRANNY executes applications as sets of Granules (GRANNY supports elasticity by vertically scaling multi-threaded applications, adding more Granules, or horizontally migrating Granules of multi-process applications to consolidate them across VMs.)

Such fine granular resource management, however, is not possible due to the limitations of today's runtimes for multi-threaded/process applications: elastically scaling threads requires runtime support, otherwise it may violate consistency with concurrent memory accesses by threads. For example, threads may have shared variables pending synchronization; similarly, migrating processes between VMs requires support for *process-level checkpointing*, e.g., CRIU [11], which incurs significant overheads [18, 60]. Without runtime support, it also cannot ensure consistent checkpoints across all distributed processes when messages are in-flight. Applying existing migration techniques [54] thus would require changes to applications, runtimes, and OS kernels.

3 GRANNY Design

We describe the design of GRANNY, a distributed runtime for multi-threaded/processing applications (§3.1). At its core lies the abstraction of a *Granule* (§3.2), which is the building block for the execution of both thread- and process-based applications. The state of Granules can be captured efficiently using *snapshots* (§3.3), thus enabling elasticity and migration.

3.1 Overview

GRANNY executes multi-threaded and multi-process applications as a set of Granules (shown as circles in Fig. 1). A Granule represents a single thread of execution and has its own mappings for code and data: a multi-threaded application (OpenMP) is therefore a set of Granules with shared-memory code and data mappings; a multi-process application (MPI) is a set of Granules with non-shared data mappings.

Each Granule runs in a single WebAssembly [17] module with its application code and executes on a vCPU in a VM. The GRANNY runtime decouples Granule state from OS ker-

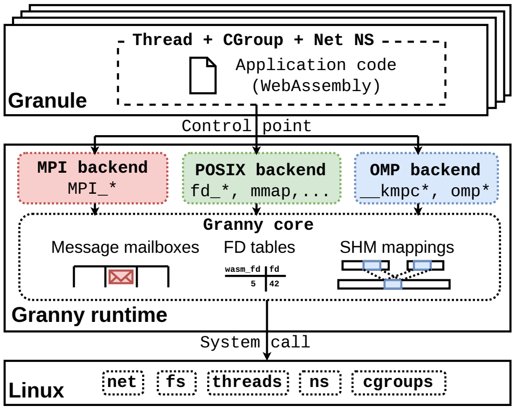


Figure 2: Granny architecture (Application code, cross-compiled to WebAssembly, can make calls, via control points, to different backends. The Granny core maintains relevant runtime system state and defers to OS system calls when necessary.)

nel state, which makes the snapshotting of Granules simpler and faster than traditional process checkpoints [11]. It also allows Granny to execute multiple Granules within the same process (shown as different colors in Fig. 1), because WebAssembly modules are isolated from each other and the runtime. Each VM runs a single instance of the Granny runtime, and there is also one cluster-wide Granny scheduler.

Granny uses the Granule abstraction (§3.2) and snapshots (§3.3) to implement management actions (§4): it can spawn Granules with thread semantics to vertically scale multi-threaded applications (§4.2), and it can horizontally migrate the snapshot of a Granule with process semantics (§4.3). The Granny scheduler uses these management actions to implement dynamic scheduling policies (§5): as shown in Fig. 1, it can speed up App 1 with the newly released vCPUs of VM A, improving utilization, or consolidate App 3 to VM C, improving locality.

3.2 Granule abstraction

Fig. 2 shows the Granny architecture. A Granule executes as an OS thread inside the VM in which it is spawned. Application code is deployed in an isolated WebAssembly module that only has external access to pre-defined OpenMP, MPI, and POSIX operations implemented by the Granny runtime (the various backends in Fig. 2). The Granny backends are built on top of a shared Granny runtime core that keeps the per-Granule state and decouples backends from the OS.

Applications must be cross-compiled to WebAssembly [17], a memory-safe and platform-independent binary instruction format that supports a wide range of programming languages. The use of WebAssembly allows Granules to execute side-by-side within one virtual address space, together with the Granny runtime, while enforcing memory safety [19] and reducing interaction with the privileged OS kernel. In addition to the file system and network operations necessary for multi-process applications, Granny only relies on the OS kernel to schedule threads and guarantee resource (CPU and network) fairness.

WebAssembly code cannot, in general, interact with its host environment. In Granny, application code can use *control points* to interact with the runtime (§4.1). A control point is triggered by a call to one of the supported APIs, POSIX, MPI, and OpenMP, implemented in the backends. Control points come at no cost for developers and at little cost for the runtime: they are injected by leaving the corresponding API symbol (e.g., `MPI_Barrier`) as undefined during cross-compilation and marked as a function import [13]. The symbol is resolved at runtime and triggers a WebAssembly context switch executed in tenths of cycles, similar to a function call [34].
The Granny runtime makes, whenever necessary, system calls to the underlying OS, e.g., to send cross-VM messages or write to a file descriptor. It records these interactions to ensure that Granule state can be encapsulated in a snapshot.

3.3 Granule implementation

Control points. Our Granny prototype has three backends:

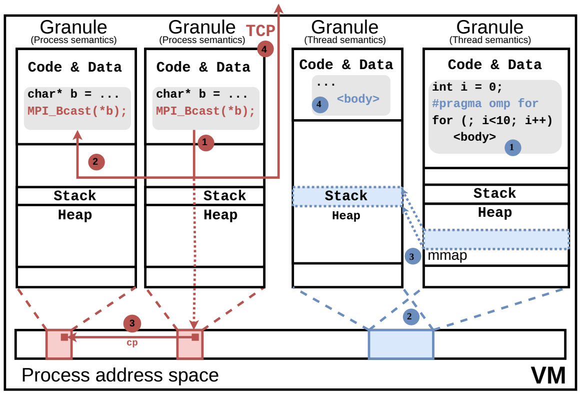
The **MPI** backend implements the standard `MPI_*` APIs (e.g., `MPI_Reduce` [33]) and provides reliable Granule-to-Granule messaging. It uses in-memory message mailboxes for message reception, and the host network stack for cross-VM messaging. The Granny runtime maintains consistent Granule addressing tables across Granule migrations.

The **OpenMP** (OMP) backend implements the interface exposed by LLVM's OpenMP runtime (`libomp`) after OpenMP pragmas have been expanded (e.g., `__kmpc_fork_call` [26]). For correct and safe OpenMP execution, the Granny runtime must carefully manage the shared and private memory regions of different Granules (see §3.3).

The **POSIX** backend implements WebAssembly's standard system interface (WASI) [55]. The use of WASI simplifies the cross-compilation of large codebases, because it enables the static linking of applications with WASI-aware libraries, such as `wasi-libc` [57]. The backend only implements the symbols needed by compute-intensive applications, which are mostly filesystem APIs. To maintain a consistent Granule state and facilitate snapshots and migration, the Granny runtime maps the file descriptors used internally in WebAssembly code, to the OS file descriptors used in practice.

Memory layout. Due to WebAssembly's linear memory model [17], Granules have a simple memory layout, shown in Fig. 3, which facilitates spawning and snapshotting. A Granule occupies a contiguous region of virtual memory with the code, data, a stack, and a heap. Spatial isolation comes from a combination of WebAssembly's memory safety and guard pages, and is it enforced by the WebAssembly runtime [10].

The main difference between a Granule executing with process semantics and with thread semantics is their memory layout (red and blue Granules in Fig. 3, respectively): Granules



```
char* b = ...
MPI_Bcast(*b);
```

```
char* b = ...
MPI_Bcast(*b);
```

```
...
<body>
```

```
int i = 0;
#pragma omp for
for (; i<10; i++)
  <body>
```

Figure 3: Granules memory layout in a single GRANNY instance (Granules execute side-by-side in an address space, and rely on WebAssembly's sandboxing for spatial isolation. Granules executing with processes semantics execute in different sandboxes, whereas Granules executing with thread semantics share the same sandbox.)

with processes semantics have separate linear memories, with the same application mapped into them. To send messages, the sender Granule indicates the buffer to be sent (Fig. 3, ❶), and the MPI backend in the GRANNY runtime decides if it is a local or a remote message. For a local message, the runtime can directly enqueue the message metadata (❷) and copy the buffer contents into the reception buffer (❸). For a remote message, the runtime will send the whole payload over TCP to the appropriate GRANNY runtime instance (❹).

Granules with thread semantics share a linear memory and have separate stacks. When a Granule creates new threads, such as with OpenMP `#pragma omp for` (Fig. 3, ❶), the GRANNY runtime spawns a new Granule mapped to the same linear memory (❷). To give each thread a separate execution context, but maintain WebAssembly's memory sandboxing, GRANNY allocates an area in the parent's heap for the child's stack (❸), and sets the child's code entrypoint to the corresponding OpenMP task (❹). Additional care goes into maintaining shared variable visibility, as well as their consistency.

Snapshots. Granules thus have a simple memory layout, which means that their execution state can be captured combining the span of the linear memory, together with the Granule state in the GRANNY runtime (Fig. 2). The combined linear memory and runtime state is what we call a Granule *snapshot*. Having a concise snapshot representation for Granules is essential for migration. Since all state is contained in a snapshot, GRANNY does not require host kernel modifications to obtain a Granule's full execution state. This is in contrast to general process checkpointing [11], which must also extract process state from the host kernel.

4 Granule Management

GRANNY performs management actions, vertical scaling (§4.2), and horizontal migration (§4.3) by interrupting application execution at well-defined *control points* (§4.1). This ensures GRANNY always operates with a consistent application state. The following sections describe these operations in more detail.

4.1 Interrupting Granules at control points

GRANNY takes control over Granule execution at control points, i.e., at every call to one of the supported runtime backend APIs (see Fig. 2). GRANNY defines two types of control points, *regular* and *barrier* control points, that enable the runtime core to perform different operations. At regular control points, such as calls to the POSIX API, the application state is not guaranteed to be consistent, but this still allows GRANNY to send point-to-point messages or operate on shared memory. In contrast, barrier control points guarantee that the application state is consistent, i.e., no messages are in-flight and no modifications to shared variables are pending to be synchronized. Therefore, GRANNY can only perform management operations at barrier control points.

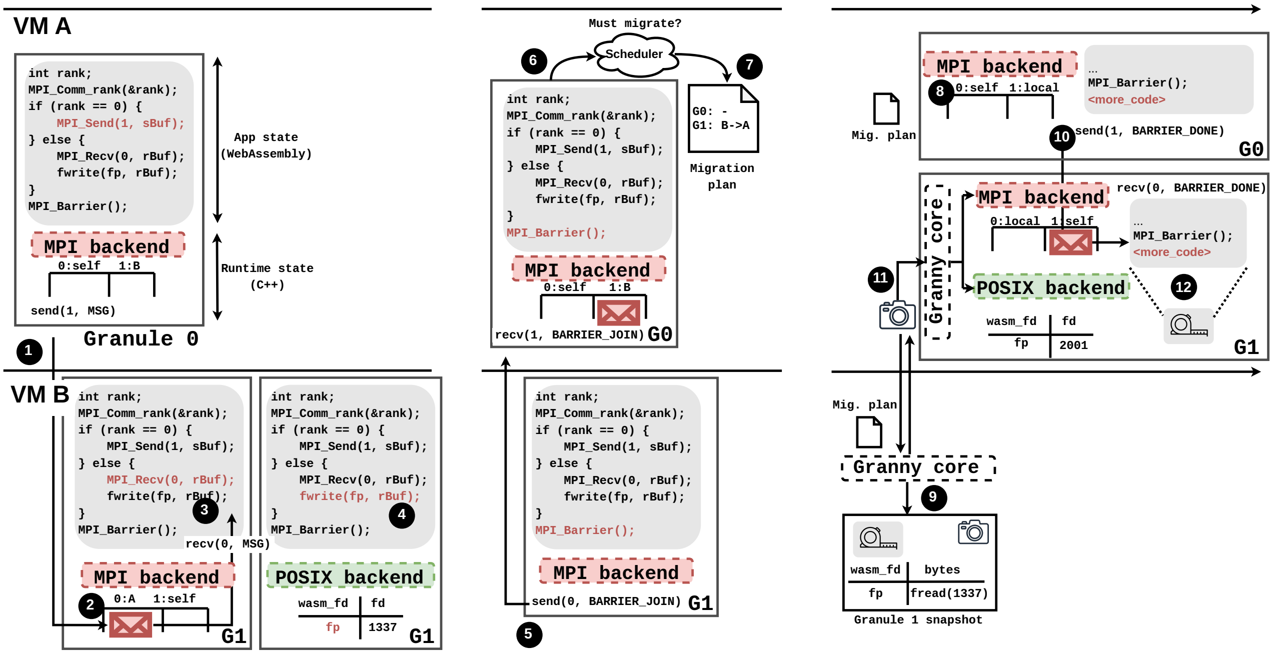
Differentiating between regular and barrier control points requires semantic knowledge of the shared memory/message passing API, as well as control over its implementation. For example, GRANNY's MPI backend implementation of `MPI_Barrier` has a reduce phase in which all Granules send messages to the Granule with the lowest MPI rank, and a broadcast phase in which all Granules are notified that the barrier has completed. After the reduce phase, the Granule with the lowest MPI rank can rely on the fact that there are no outstanding messages and has thus reached a consistent state. A similar explanation, with shared memory synchronization instead of messages, applies to GRANNY's OpenMP backend implementation of `#pragma omp barrier`.

As a consequence, vertical scaling and horizontal migration in GRANNY is a co-operative process. When a Granule triggers a barrier control point, the runtime interacts with the scheduler and adds, removes, or migrates Granules as indicated. In practice, barrier control points are frequent enough, so that their co-operative nature does not hinder the benefits in resource management, as we can see later in §6.

4.2 Vertical scaling

Implementing vertical scaling in GRANNY is straightforward if we take into account the memory layout presented in Fig. 3. To create a child Granule from a parent Granule, the runtime allocates the child's stack in the parent's heap, and instantiates a new Granule on top of the same WebAssembly linear memory, changing only the stack pointer. The child's entrypoint is indicated as an index in the WebAssembly module's function table. We also add guard pages around each child's stack to mitigate potential stack overflows (typical of various threading implementations in WebAssembly [58]).

Since vertical scaling happens during barrier control points in OpenMP, the runtime can easily distribute the work across any Granules by simply setting the appropriate per-thread and



```
int rank;
MPI_Comm_rank(&rank);
if (rank == 0) {
    MPI_Send(1, sBuf);
} else {
    MPI_Recv(0, rBuf);
    fwrite(fp, rBuf);
}
MPI_Barrier();
```

```
...
MPI_Barrier();
<more_code>
```

(a) Normal execution

(b) Barrier control point

(c) Migration

Figure 4: Horizontal migration overview (When Granules reach a barrier control point, such as `MPI_Barrier`, the application's main Granule queries the resource scheduler. If triggered, the runtime performs a horizontal migration by taking a snapshot of the migrated Granule and transferring it to the destination VM. Runtime state, such as open file descriptors, is also snapshotted and reconstructed at the destination.)

global variables in the OpenMP specification. We design a scheduling policy using vertical scaling in §5.2.

4.3 Horizontal migration

Fig. 4 illustrates horizontal migration. The MPI backend gives each Granule a unique integer identifier, akin to the MPI rank, and implements messaging as a three-step process: it captures the send operation as a call to a backend operation such as `MPI_Send` (❶); it copies the message into the receiving Granule's mailbox within the target GRANNY runtime core (❷); and it delivers the message to the application by capturing (❸), and potentially blocking, calls to a receive operation (which the MPI backend matches to the corresponding mailbox, according to the MPI specification). Note that Granules may also call other API backends, such as file writes via the POSIX backend (❹). As previously introduced, the POSIX backend serves the request and keeps track of the mapping between WebAssembly file descriptors and OS ones.

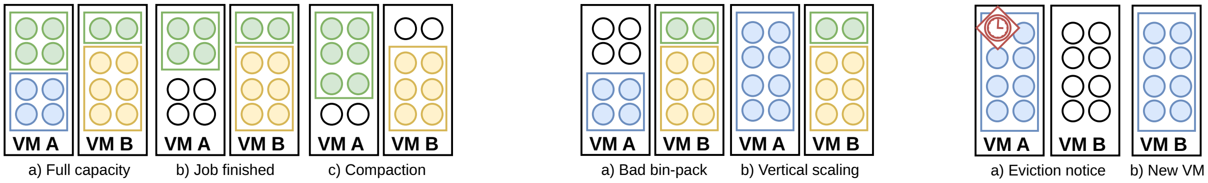
When Granules reach a barrier control point (such as `MPI_Barrier` in Fig. 4b), the runtime can perform horizontal migrations safely. For example, in the case of `MPI_Barrier`, all Granules send a `BARRIER_JOIN` message (❺) to the zero-th MPI rank (Granule 0 in VM A in the example), and wait for a `BARRIER_DONE` message before continuing. When the runtime for Granule 0 has received all messages, it queries the scheduler for any horizontal migrations (❻). The resource scheduler then applies the scheduling policy and returns a migration plan (❼). After all migrations have been performed (if any), the runtime broadcasts a `BARRIER_DONE` message so all blocked Granules resume execution. In our example, Granule 1 must be migrated from VM B to VM A.

To perform the migration, shown in Fig. 4c, the runtime in VM A distributes the migration plan to the other runtimes involved in the execution (in this case the runtime in VM B), and all proceed to individually prepare for the migration. For VM A, this means the runtime must create a new, empty Granule 1, setup its mailbox mappings, and update the entry for Granule 1 on the mailbox of Granule 0 (❽). For the runtime in VM B, this means creating a snapshot of Granule 1, including any runtime state, and terminating the old Granule 1 (❾).

Once the migration has been prepared, the runtime in VM A can resume execution of Granule 0, which broadcasts the `BARRIER_DONE` message to the *updated* mailboxes (❿). The runtime in VM A can also restore Granule 1 from the received snapshot, re-construct the file descriptor tables, and resume execution where it left off, i.e., blocked waiting for a `BARRIER_DONE` (⓫). Since the restored Granule 1 uses the updated mailboxes, it has the message and resumes execution (⓬). We design two scheduling policies using horizontal migration in §5.1 and §5.3.

5 Granular Management Policies

We implement three scheduling policies, described in the following sections, to improve the utilization and performance



(a) Compaction policy (b) Elastic policy (c) Spot policy

Figure 5: Dynamic scheduling policies implemented for their use with GRANNY (Different colors indicate different jobs.)

of cloud-based multi-process and multi-thread execution. The policies use GRANNY's support for vertical scaling and horizontal migration, and treat scheduled the applications as black boxes, i.e., without knowledge of future applications, their distribution of sizes, or their expected duration.

5.1 Improving locality with compaction policy

When scheduling MPI applications, existing cloud schedulers face a utilization/locality trade-off: they can either assign available vCPUs to an MPI application to achieve high utilization, or assign an entire VM to an MPI application to achieve high locality at the expense of having unused vCPUs in those VMs. The former reduces queue wait times for applications, but it may increase execution time, because applications may have to communicate across different VMs; the latter optimizes execution time at the cost of increasing queuing time due to the wait for an available VM of the required size.

Instead, we adopt a *compaction* policy (see Fig. 5a) that uses GRANNY to combine the benefits of both approaches. The scheduler first eagerly deploys MPI applications on any available vCPUs, which results in the highly fragmented green application in our example. When vCPUs are later released (e.g., the blue application terminates), the scheduler performs horizontal migration of Granules to increase the locality of executing applications. This results in less cross-VM communication, which speeds up the green application.

We define our fragmentation metric as the total number of Granule-to-Granule connections that cross a VM boundary in an application. When the application reaches a barrier control point, the scheduler checks if it can reduce its cross-VM links by performing any set of horizontal migrations. We show experimentally in §6.2 that, using this policy, GRANNY maintains high cluster utilization while keeping fragmentation low, thus improving performance.

5.2 Improving utilization with elastic policy

Even with a compaction policy, some cluster resources remain idle due to the nature of bin-packing scheduling. VMs may have some spare capacity that is insufficient to deploy the next application in the queue. This is particularly true with OpenMP applications that cannot be distributed across VMs.

In response, we introduce an *elastic* policy (see Fig. 5b). When there is spare capacity in a VM (Fig. 5b-a) and an OpenMP application reaches a barrier control point, the scheduler triggers a vertical scale-up to utilize these idle resources (Fig. 5b-b). This improves cluster utilization and application performance by exploiting extra parallelism. Our policy is careful to not mistake fork-join patterns of co-located OpenMP application with truly available resources by keeping track of the `OMP_NUM_THREADS` environment variable.

In §6.3, we show that the elastic policy reduces end-to-end execution time while decreasing idle vCPUs.

5.3 Ephemeral VMs with spot policy

Compute-intensive applications may be deployed on a pool of spot VMs [7] to benefit from lower costs. This introduces the challenge of handling partial failures, because spot VMs are withdrawn by the cloud provider after a short grace period.

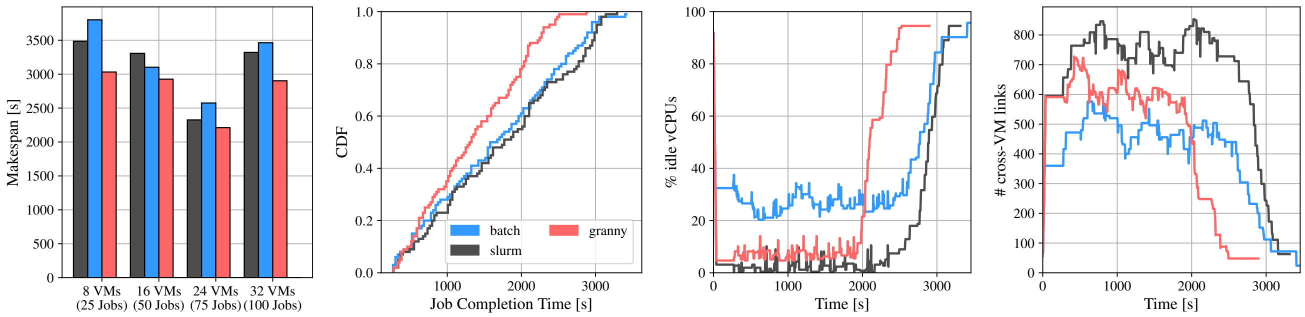
Fig. 5c shows our *spot* policy. We follow a two-part horizontal migration approach: (1) when the cloud provider notifies the scheduler of an upcoming spot VM eviction (Fig. 5c-a), the scheduler stops scheduling Granules to that VM; (2) when a Granule running on the to-be-evicted VM reaches a barrier control point (e.g., `MPI_Barrier` or `#pragma omp barrier`), the resource manger tries to re-schedule it on the remaining VMs (Fig. 5c-b). If there are insufficient resources, snapshots are taken of all Granules in the application, and they are terminated. Interrupted applications are then added to the beginning of the scheduler's queue.

We show experimentally in §6.4 that, using a spot policy, GRANNY reduces end-to-end execution time when deploying compute-intensive applications on spot VMs.

6 Evaluation

Our evaluation explores the benefits of using GRANNY to run MPI and OpenMP applications. We show how, using GRANNY's dynamic scheduling polices (§5), we can: (i) improve performance and locality of MPI applications while maintaining a target utilization (§6.2); (ii) improve performance and utilization of OpenMP applications by allocating extra CPU cores (§6.3); and (iii) ensure efficient fault-tolerant execution with ephemeral spot VMs (§6.4).

In our microbenchmarks (§6.5), we analyze the baseline overheads of: (i) executing MPI applications with GRANNY instead of OpenMPI (§6.5.1); (ii) executing OpenMP applications with GRANNY instead of LLVM's `libomp` (§6.5.2); (iii) the horizontal migration of Granules (§6.5.3); and (iv) the vertical scale-up of Granules (§6.5.4).



(a) Makespan (b) CDF JCT (32 VMs, 100 Jobs) (c) Idle vCPUs (32 VMs, 100 Jobs) (d) xVM Links (32 VMs, 100 Jobs)

Figure 6: Improving locality with compaction policy (We compare with Azure Batch and Slurm's default policies.)

6.1 Experimental setup

Implementation. GRANNY is implemented in 24,000 lines of C++20 code (compiled with clang-17) on top of the Faasm runtime [51]. GRANNY extends Faasm by implementing the MPI/OpenMP backends, a centralized scheduler, and migration/scale-up, but it re-uses the underlying sandbox abstraction (using WAMR [10] as the WebAssembly runtime). GRANNY is available as open-source.[1] We compile all deployed applications and their library dependencies, such as *libc*, to WebAssembly [17] using clang-18 [45], as part of GRANNY's CPP toolchain.

As baselines, we compare against two cloud resource schedulers, batch and slurm, that mirror the behaviour of Azure Batch [27] and Slurm [48], respectively. Unless otherwise stated, batch and slurm execute MPI applications with OpenMPI v4.1 [39] and OpenMP ones with `libomp` v4.5 [25].

Testbed. We deploy GRANNY and any baselines on a Kubernetes cluster [22] on Azure [28]. The cluster consists of *Standard_D8_v5* VMs [29] with 8 vCPU cores and 32 GB of memory. We deploy the resource scheduler in a separate VM in the same cluster.

Workloads. We evaluate GRANNY with unmodified applications that use OpenMP [36] for multi-threaded shared memory computation, and MPI [33] for multi-process message passing. Applications require no changes to source code, just re-compilation with the GRANNY toolchain.

6.2 Improving locality with compaction policy

This experiment explores the benefits of using a *compaction* policy (§5.1) to navigate the utilization/locality trade-off: it horizontally migrates Granules executing with process semantics. We execute a trace of MPI applications as jobs, and each application executes the LAMMPS [43] molecule dynamics simulator, running the Lennard-Jones (LJ) benchmark with a varying number of MPI processes. Applications are executed in order and may wait in a queue until sufficient resources become available in the cluster. To measure only the benefits of the policy and not differences in the MPI implementation, we configure our two baselines, batch and slurm, to use GRANNY's MPI backend.

Fig. 6 shows various performance metrics, as we increase the number of VMs and the number of applications in the trace. Fig. 6a shows that the compaction policy improves end-to-end execution time (makespan) by up to 20%. Using compaction, GRANNY always improves makespan across all baselines and cluster sizes.

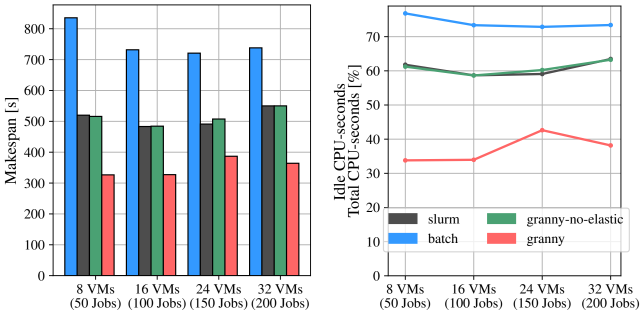
To show the compaction policy in action, Fig. 6c and Fig. 6d plot the time-series of idle vCPUs (as a proxy for cluster utilization) and cross-VM network links (as a proxy for locality) for the (32 VMs, 100 jobs) execution. We see that, differently to Slurm, GRANNY deliberately leaves a percentage of vCPUs idle corresponding to a target utilization (5% in this experiment). GRANNY can use these spare vCPUs to defragment applications at runtime by performing horizontal migrations, achieving consistently 25% less fragmentation than Slurm. In fact, GRANNY, with only 5% idle vCPUs, is closer in terms of fragmentation to Azure Batch, which behaves optimally with respect to this metric but leaves 30% of vCPUs unused.

Fig. 6b shows that this reduction in fragmentation at high cluster utilization has a direct impact on job completion time (JCT). For the (32 VMs, 100 jobs) execution, GRANNY improves median and tail JCT by up to 20%. We conclude that the compaction policy enables GRANNY to exploit the utilization/locality trade-off more effectively compared to Azure Batch and Slurm.

6.3 Improving utilization with elastic policy

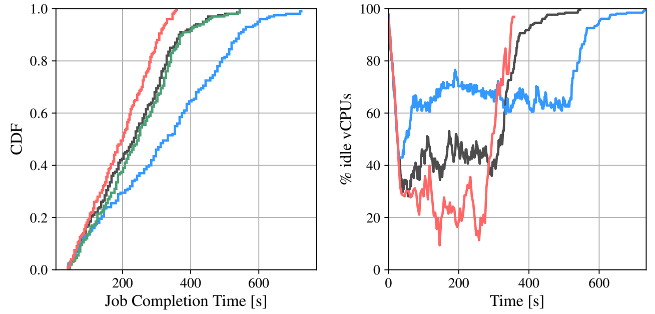
This experiment explores the benefits of using an *elastic* policy (§5.2) to vertically scale OpenMP applications by adding Granules with thread semantics (§5.2). We execute a trace of OpenMP applications as jobs, and each application executes a large-scale version of the p2p ParRes OpenMP kernel [41], which performs a compute-intensive pipelined parallel algorithm on a large matrix, and requires a different number of

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



(a) Makespan

(b) Aggregate idle vCPUs



(c) JCT (32 VMs, 200 Jobs)

(d) Idle vCPUs (32 VMs, 200 Jobs)

Figure 7: Improving utilization with elastic policy (We compare to Azure Batch and Slurm with `libomp` as OpenMP runtime, and Slurm with GRANNY's OpenMP backend, granny-no-elastic.)

OpenMP threads. In this experiment, the baselines (batch and slurm) execute applications using `libomp` [25]. We include an additional baseline, granny-no-elastic, which corresponds to slurm using GRANNY' OpenMP backend.

Fig. 7a shows that GRANNY improves makespan by up to 60% compared to the native baselines and GRANNY without the elastic policy (granny-no-elastic). This confirms that the performance improvements come from the policy. Indeed, we can assert that GRANNY reduces the idle CPU cores in the cluster by up to 30% (Fig. 7b), and it uses these extra cores to improve median and tail JCT by up to 50% (Fig. 7c).

This large performance gap can be understood when considering how many compute resources are left idle by the native baselines. Fig. 7d shows a time series of the percentage of idle vCPUs when running 200 jobs on a 32-VM cluster. Azure Batch and Slurm, even when the job queue is not empty, consistently leave 60% and 40% of vCPUs idle. This is due to multiple reasons: (i) OpenMP applications have fixed parallelism; (ii) it is not possible to distribute them across VMs; and (iii) in the case of Azure Batch, it is not possible to run different applications on the same VM concurrently. By vertically scaling-up, GRANNY maintains the fraction of idle vCPUs at around 20% while there are still pending jobs.

GRANNY's improvements in terms of JCT are partly due to the fact that our OpenMP workload always benefits from increasing its parallelism (§6.5.4). In a real deployment, an elastic policy should be accompanied by runtime profiling to determine when a workload exhausts its parallelism.

6.4 Ephemeral VMs with spot policy

This experiment explores the performance benefits of using a *spot* policy (§5.3) to execute compute-intensive applications on a cluster with ephemeral spot VMs. The submitted applications and the native baselines are the same as in §6.2. In this experiment, the native baselines execute applications using the OpenMPI [39] runtime. To emulate the behaviour of spot VMs while making our findings reproducible, we withdraw VMs at a pre-defined rate with a 1-min grace period.

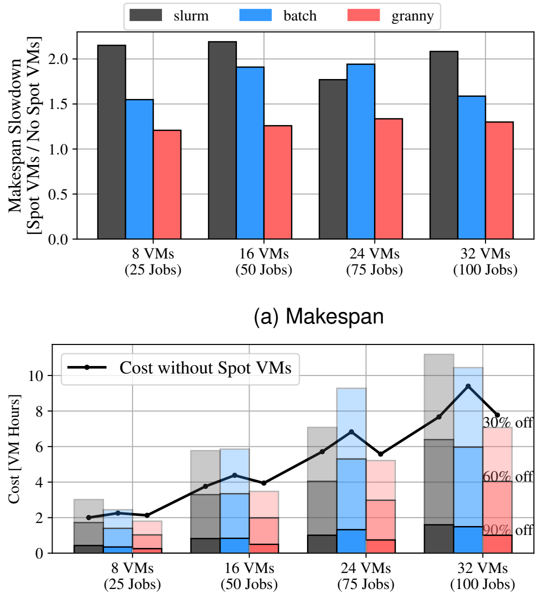
The eviction rate for this experiment is 25% of the VMs, selected at random, each minute. We choose this eviction rate after measuring the eviction rate for `Standard_D8_v5` spot VMs using Azure's Resource Graph Analyzer [8] (25% per hour) and scaling it to our MPI applications' length (minutes, instead of hours, to make the experiments reproducible).

Fig. 8a shows the reduction in makespan when comparing each baseline to itself without evictions. We see that, across cluster and batch sizes, all native baselines experience a minimum of a 50% slowdown, and a maximum of a 2× slowdown. This is because batch and slurm are unaware of evictions and must restart jobs each time they fail due to an evicted VM. Instead, GRANNY uses the spot policy and, as a consequence, its slowdown is at most 25%, consistent with the eviction rate.

The native slowdowns of 50%–100% can potentially thwart the cost benefits of using spot VMs. Indeed, Fig. 8b shows the normalized cost of each execution for the range of discounts based on Azure's spot VM price list [6]: 30%, 60%, and 90%. We calculate the cost by assuming a unit price per VM-hour and applying the corresponding discount. We also overlay the cost of not using spot VMs.

We observe that, for the native baselines, the effectiveness of spot VMs in terms of cost savings depends on the discount rate at which they are offered. Counterintuitively, for many discount ranges, it is not cost-effective to run applications using OpenMPI on spot VMs, because the re-execution costs outweigh the price discount. In contrast, with GRANNY, it is always cost-effective to use spot VMs, because the eviction slowdown is lower than the smallest cost discount.

Discussion. Our experiments are conducted on a 32-VM cluster with job traces that contain one type of application and do not capture inter-arrival times. Given the co-operative and on-demand nature of vertical scale-up and horizontal migration, however, we expect our results to generalize to larger cluster sizes. We also believe that GRANNY can improve performance and utilization for other workloads, but this may require the design of further scheduling policies.



(a) Makespan

(b) Cost (Assuming a unit cost per VM hour.)

Figure 8: Ephemeral VMs with spot policy (We compare to Azure Batch and Slurm with their default policies and native runtimes.)

6.5 Microbenchmarks

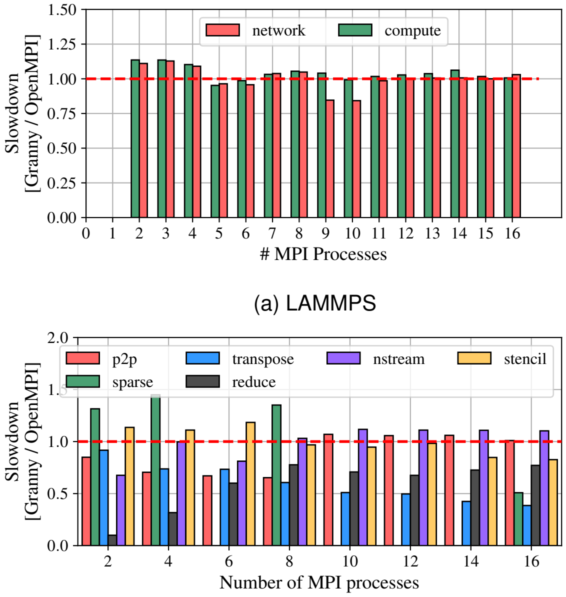
In this section, we isolate different components of Granny and evaluate overheads. We evaluate Granny's MPI backend (§6.5.1), OpenMP backend (§6.5.2), as well as horizontal migration (§6.5.3) and vertical scale-up (§6.5.4).

6.5.1 MPI backend

This experiment investigates the overhead of Granny's MPI backend implementation compared to OpenMPI [39]. We run the same MPI application as in §6.2. To stress Granny's communication layer, we update the benchmark and increase the synchronisation steps, resulting in three orders of magnitude more cross-VM messages. We refer to the unmodified LJ benchmark as compute, and the modified one as network. We also execute a subset of the ParRes kernels [41] to evaluate specific parts of Granny's MPI implementation.

Fig. 9a shows the slowdown in execution time of Granny compared to OpenMPI when executing the two LAMMPS simulations with different levels of parallelism on two VMs. We observe that the overhead introduced by Granny is within 10% and often negligible. Granny occasionally introduces minor performance gains due to the benefits of intra-process co-location of Granules.

Fig. 9b shows the slowdown when executing the ParRes kernels with different levels of parallelism. For this workload, the performance of Granny and OpenMPI varies more than for LAMMPS, because these kernels execute MPI operations in a tight-loop. As a consequence, the performance benefit of intra-process co-location of Granules becomes more significant, leading to a performance benefit for Granny. In these cases, Granny can replace the sending of messages (reduce) with reducing shared memory variables.



(a) LAMMPS

(b) MPI ParRes Kernels

Figure 9: MPI backend performance

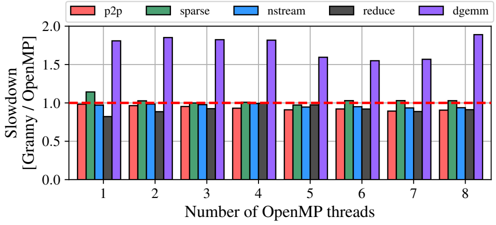
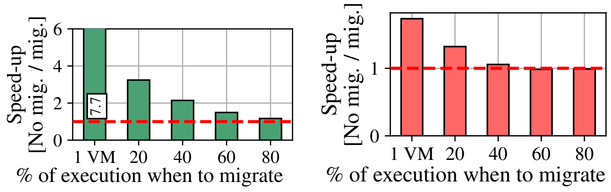


Figure 10: OpenMP backend performance

6.5.2 OpenMP backend

Next, we investigate the overhead of Granny's OpenMP backend compared to LLVM's `libomp`. We execute the ParRes kernels [41] in their OpenMP implementation. We execute each kernel with a varying number of threads and take the average execution time over 5 runs.

Fig. 10 shows the slowdown in execution time of Granny compared to `libomp` for a variety of kernels. We observe that, for most kernels, Granny's performance matches the native baseline. This is because Granny's OpenMP backend adds minimal book-keeping at runtime and relies on the OS' synchronization primitives. For the `dgemm` kernel, Granny introduces an 80% slowdown. This kernel performs a dense matrix multiplication, and the overheads come from WebAssembly's less efficient floating-point operations [51]. We expect future WebAssembly releases and compilers to improve floating-point performance.



(a) All-to-all

(b) LAMMPS

Figure 11: Speedup when migrating Granules (We deploy 8 Granules with process semantics across two VMs, and migrate 4 at runtime. We report the speedup over no migration.)

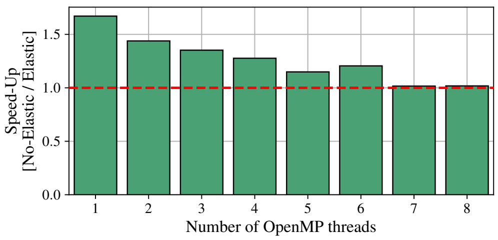


Figure 12: Speedup when elastically scaling to more vCPUs (We deploy a varying number of Granules with thread semantics, and elastically scale up to all CPU cores after 50% of execution.)

6.5.3 Granule migration

This experiment explores GRANNY's performance overhead when doing a horizontal migration of Granules at runtime, and the benefits of improved co-location. As workloads, we use the compute-bound LAMMPS simulation, and a network-bound *all-to-all* kernel, which performs synchronisation over a vector in a loop. For each experiment, we artificially fragment the 8 Granules with process semantics across two VMs. We then trigger a horizontal migration of half of them to the other VM at 20%, 40%, 60%, or 80% of execution time.

Fig. 11a shows the speedup when migrating a network-bound application compared to no migration. For such an application, fragmentation has a high cost: the speedup for running in one VM (1 VM) is 7.7×. By migrating after 20%, 40%, 60%, and 80%, of execution, GRANNY achieves speedups of 3.5×, 2.2×, 1.5×, and 1.1×, respectively. For network-bound applications, the overheads of horizontal migration are thus outweighed by the benefits of co-location.

Fig. 11b shows the speedup when migrating a compute-bound application. Here, fragmentation is less of an issue: the speedup for running in one VM is 1.7×. By migrating after 20% and 40% of execution, we achieve speedups of 1.3× and 1.1×, respectively. We observe no benefit when migrating later during execution. There exists a trade-off between the frequency and cost of migration, however, GRANNY's horizontal migration mechanism introduces a negligible overhead, thus enabling cloud providers to optimize for locality.

Migration time is dominated by the time to transfer the snapshot from one VM to another. For a 4 MB snapshot, corresponding to the *all-to-all* kernel, Granule migration is on the order of 30 ms. From this, only 3 ms correspond to creating the snapshot, which is almost an order of magnitude faster than a highly-optimized version of CRIU [60].

6.5.4 Elastic scale-up

Finally, we investigate GRANNY's performance overhead when doing vertical scaling of a multi-threaded application, and the potential benefits of increased parallelism. As a workload, we deploy the same OpenMP application as in §6.3. We initially use a varying number of Granules, and scale up to all available CPU cores (8) after 50% of execution.

Fig. 12 shows that, by scaling-up to use more CPU cores, GRANNY achieves a speedup of up to 60% when scaling from 1 to 6 threads. With more than 7 initial threads, we do not observe a benefit, because we have exhausted the application's parallelism. We conclude that elastically scaling-up does not slow down execution, because elastic scaling naturally fits with OpenMP's fork-join semantics.

7 Discussion

We now discuss some of GRANNY's design decisions and how its insights could be transferred to other cloud systems.

WebAssembly. Despite offering an efficient sandboxed environment where Granules execute, WebAssembly is still an evolving technology. The performance of some WebAssembly operations, particularly floating point operations, still remains below natively compiled code [14] and sandboxes that exceed 4 GB incur a higher sandboxing overhead [56]. We expect these issues to be addressed as compiler support for WebAssembly matures.

Programming models. GRANNY supports OpenMP and MPI and is therefore capable of running a large set of compute-intensive applications. GRANNY's design, however, is not tied to any programming model, and other programming models could be supported after identification of barrier control points for elastic scaling and migration.

A limitation of GRANNY is that it offers a CPU execution model only and cannot support GPU-based computation. This is due to its reliance on WebAssembly as an execution format. We plan on exploring GPU-based Granules with APIs such `wasi-nn` [59] in future work.

Resource schedulers and policies. GRANNY can integrate with other schedulers. For this, a resource scheduler needs to expose an endpoint to receive notifications from running applications and implement dynamic scheduling policies.

8 Related Work

Compute-intensive cloud applications. All major cloud providers offer solutions to support compute-intensive applications in the cloud [5, 30, 31]. To execute such applications, providers deploy batch scheduling solutions [2, 27] inspired by HPC batch schedulers. Much previous work focuses on making scheduling decisions more effective [24].

Complementary to this, our work demonstrates that vertical scaling of multi-threaded applications and horizontal migration of multi-process ones is a fundamental missing piece for granular management of compute-intensive applications in cloud environments. Granny enables a number of scheduling policies that can be implemented to jointly optimize resource utilization and compute and data locality.

Checkpointing and migration. Nu [46] is a distributed computing platform that supports resource *fungibility* through migration. It uses multi-threaded *Proclets*, which can only communicate with each other by sending messages. This means that Nu's execution model is not transparent to existing multi-threaded applications, but rather requires a complete application re-write to partition shared application state and use its C++ messaging API. In contrast, Granny transparently executes and migrates existing multi-threaded and multi-process applications with no extra source code modifications required (with only recompilation to WebAssembly).

CloudScale [50] automates fine-grained elastic resource scaling in a shared cluster. It uses migration to handle scheduling or scaling issues. Granny is complementary to CloudScale in that it focuses on fine granular (thread/process) resource management.

There are various approaches to migrate parts of running applications transparently without violating integrity. MigrOS [42] checkpoints the state of RDMA-enabled applications executing in containers, but requires expensive modifications to the underlying network protocol. Other work [54] has performed live migration of MPI applications with process-level migration, which incurs high bandwidth consumption and necessitates OS kernel modifications. General-purpose process (e.g., CRIU [11] or DMTCP [15]) or VM migration techniques have been shown [18] to be poorly suited for efficient shared memory and message passing applications.

In contrast, Granny executes entirely in user-space and exploits lightweight sandboxing and traps using WebAssembly. Through its semantic knowledge of shared memory and message passing APIs, Granny achieves the same migration support but at a lower performance cost, independently of the employed VMs or host OS kernel.

Shared memory and message passing runtimes. Hoplite [62] uses collective communication primitives for building fault-tolerant task-based distributed applications. Ray [32] unifies task-parallel and actor-based computation using a single interface. It offers transparent state and message passing, irrespective of distribution, together with transparent scaling.

Granny could be extended to support these programming models. Our initial prototype implementation of Granny focuses on two existing programming models, namely MPI and OpenMP, which constitute opposite ends of the spectrum when writing parallel applications.

Granny's threading implementation is similar to that of `wasi-threads` [58]: both rely on WebAssembly's atomic intrinsics and have similar memory layouts, but Granny delegates control of shared memory synchronization (e.g., mutexes and locks) to the host runtime, whereas `wasi-threads` handles it in WebAssembly.

Elastic scaling. Elasticity for cloud-based execution has been traditionally implemented at the programming model level. MapReduce [12], actor-based models such as Akka [1] or Orleans [9], or serverless system such as AWS Lambda [3] or Faasm [51] implement elasticity at the worker/actor/function levels. In this work, we enable elasticity at the finer granularity of threads in multi-threaded applications. OpenMP has some elasticity support via `omp_set_dynamic`, but such support is only used to scale *down* in case of contention [38].

9 Conclusions

Existing resource schedulers fail to jointly optimize resource utilization and application performance, because application resource allocation cannot be modified after an application begin execution. To address this issue, we presented Granny, a new distributed runtime for unmodified multi-threaded (OpenMP) and multi-processing (MPI) applications. Through its Granule abstraction, Granny is able to spawn and migrate Granules at run-time. With this mechanism, we implement dynamic scheduling policies that improve both resource utilization and application performance when compared to existing cloud resource schedulers.

Acknowledgments. This work has been partially funded by the European Union through the Horizon Europe projects CloudStars (101086248) and CloudSkin (101092646), and by FCT (UIDB/50021/2020). We thank our shepherd, Xin Jin, and the anonymous reviewers for their helpful feedback.

References

[1] Akka. Build, Operate, and Secure Distributed Applications. https://akka.io/, 2024.

[2] Amazon Web Services. AWS Batch. https://aws.amazon.com/batch/, 2021.

[3] Amazon Web Services. AWS Lambda. https://aws.amazon.com/lambda/, 2021.

[4] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, 2018.

[5] AWS. Genomics in the cloud. https://aws.amazon.com/health/genomics/, 2022.

[6] Azure. Pricing - Linux Spot VMs. https://azure.microsoft.com/en-gb/pricing/details/virtual-machines/linux/#pricing, 2024.

[7] Azure. Use Spot VM Instances. https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms, 2024.

[8] Azure. Use Spot VM Instances - Eviction Rate and Pricing History. https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms#pricing-and-eviction-history, 2024.

[9] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.

[10] ByteCode Alliance. WebAssembly Micro Runtime. https://bytecodealliance.github.io/wamr.dev/, 2024.

[11] CRIU. Checkpoint-Restore in Userspace. https://www.criu.org/Main_Page, 2021.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 2008.

[13] FFmpeg Contributors. Webassembly lld port. https://lld.llvm.org/WebAssembly.htmlimports, 2022.

[14] Frank Denis Blog. Performance of WebAssembly runtimes in 2023. https://00f.net/2023/01/04/webassembly-benchmark-2023/, 2024.

[15] Google. Github - DeepVariant. https://github.com/google/deepvariant, 2022.

[16] Google. Google Cloud Batch. https://cloud.google.com/batch, 2023.

[17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to Speed with WebAssembly. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[18] Khaled Z. Ibrahim, Steven Hofmeyr, Costin Iancu, and Eric Roman. Optimized pre-copy live migration for memory intensive applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[19] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Veriwasm: Sfi safety for native-compiled wasm. NDSS'21.

[20] Ken Kennedy and Kathryn S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 6th International Conference on Supercomputing*, ICS '92, 1992.

[21] kube batch. A batch scheduler of kubernetes for high performance workload, e.g. AI/ML, BigData, HPC. https://github.com/kubernetes-retired/kube-batch, 2024.

[22] Kubernetes. Kubernetes- Production-Grade Container Orchestration. https://kubernetes.io/, 2022.

[23] Baolin Li, Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, Karen Gettings, and Devesh Tiwari. Ribbon: Cost-effective and qos-aware deep learning model inference using a diverse pool of cloud computing instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, 2021.

[24] Bing Lin, Wenzhong Guo, Xianghan Zheng, Hong Zhang, Chunming Rong, and Guolong Chen. Optimization scheduling for scientific applications with different priorities across multiple clouds. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, CLOUDCOM '14, 2014.

[25] LLVM Project. LLVM/OpenMP documentation. https://openmp.llvm.org/, 2022.

[26] LLVM Project. LLVM OpenMP Runtime Library Interface. https://openmp.llvm.org/doxygen/index.html, 2024.

[27] Microsoft. Azure Batch. https://azure.microsoft.com/en-us/services/batch/, 2021.

[28] Microsoft. Azure: Cloud Computing Services. https://azure.microsoft.com/en-us/, 2021.

[29] Microsoft. Azure Virtual Machines. https://docs.microsoft.com/en-us/azure/virtual-machines/dv2-dsv2-series, 2021.

[30] Microsoft. Azure Genomics. https://azure.microsoft.com/en-us/services/genomics/, 2022.

[31] Microsoft. Azure HPC. https://azure.microsoft.com/en-gb/solutions/high-performance-computing, 2022.

[32] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI'18, 2017.

[33] MPI. MPI Forum. https://www.mpi-forum.org/, 2022.

[34] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean Tullsen, and Deian Stefan. Going beyond the limits of sfi: Flexible and secure hardware-assisted in-process isolation with hfi. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, 2023.

[35] OpenFOAM. Github - OpenFOAM. https://github.com/OpenFOAM/OpenFOAM-dev, 2022.

[36] OpenMP. The OpenMP API specification for parallel programming. https://www.openmp.org/specifications/, 2021.

[37] OpenMP Api Specification. OMP_NUM_THREADS. https://www.openmp.org/spec-html/5.0/openmpse50.html, 2022.

[38] OpenMP Architecture Review Board. OpenMP application program interface version 4.5. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, 2015.

[39] OpenMPI. OpenMPI: Open Source High Performance Computing. https://www.open-mpi.org/, 2021.

[40] OpenMPI. mpirun - Man Page. https://www.open-mpi.org/doc/v3.0/man1/mpirun.1.php, 2024.

[41] ParResKernels Team. Parallel Research Kernels. https://github.com/ParRes/Kernels, 2021.

[42] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. MigrOS: Transparent Live-Migration support for containerised RDMA applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, USENIX ATC'21, 2021.

[43] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 1993.

[44] Press Office. Up to 1.2 billion for weather and climate supercomputer. https://www.metoffice.gov.uk/about-us/press-office/news/corporate/2020/supercomputer-funding-2020, 2022.

[45] LLVM Project. Clang: a C language family frontend for LLVM. https://clang.llvm.org/, 2022.

[46] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. Nu: Achieving Microsecond-Scale resource fungibility with logical processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, NSDI'23, 2023.

[47] Sandia National Laboratories. Github - LAMMPS. https://github.com/lammps/lammps, 2020.

[48] SchedMD. Slurm Workload Manager. https://slurm.schedmd.com/overview.html, 2024.

[49] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, 2013.

[50] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, 2011.

[51] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *USENIX Annual Technical Conference (USENIX ATC)*, USENIX ATC'20, 2020.

[52] TensorFlow. Github - TensorFlow. https://github.com/tensorflow/tensorflow, 2022.

[53] Volcano. Cloud native batch scheduling system for compute-intensive workloads. https://volcano.sh/en/, 2024.

[54] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in hpc environments. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[55] WASI. WebASsembly System Interface. https://wasi.dev/, 2024.

[56] WebAssembly. Memory with 64 bit indexes. https://github.com/WebAssembly/memory64, 2024.

[57] WebAssembly. WASI libc implementation for WebAssembly. https://github.com/WebAssembly/wasi-libc, 2024.

[58] WebAssembly. WASI Threads. https://github.com/WebAssembly/wasi-threads, 2024.

[59] WebAssembly. WASI Threads. https://github.com/WebAssembly/wasi-nn, 2024.

[60] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, OSDI'23, 2023.

[61] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, 2010.

[62] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: Efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, 2021.