Kobe University Repository : Kernel

PDF issue: 2024-05-21

Distributed Cell Set : A Library for Space-Dependent Communication/Computation Overlap on Manycore Cluster

Kawanishi, Yoshiki

Finnerty, Patrick

Kamada, Tomio

Ohta, Chikara

(Citation)
PMAM'23: Proceedings of the 14th International Workshop on Programming Models and Applications for Multicores and Manycores:11-19

(Issue Date)
2023-02-25

(Resource Type)
conference proceedings

(Version)
Accepted Manuscript

(Rights)
© 2023 ACM

(URL)
https://hdl.handle.net/20.500.14094/0100483437

Distributed Cell Set : A Library for Space-Dependent Communication/Computation Overlap on Many Core Cluster

Yoshiki Kawanishi
kawanishi@fine.cs.kobe-u.ac.jp
Graduate School of System Informatics
Kobe University
Kobe, Japan

Patrick Finnerty
finnerty.patrick@fine.cs.kobe-u.ac.jp
Graduate School of System Informatics
Kobe University
Kobe, Japan

Tomio Kamada
t_kamada@konan-u.ac.jp
Department of Intelligence and Informatics
Konan University
Kobe, Japan

Chikara Ohta
ohta@port.kobe-u.ac.jp
Graduate School of System Informatics
Kobe University
Kobe, Japan

ABSTRACT

The increase in the number of cores available in modern processors makes it important for implementations to maximize their use of cores within a node by overlapping communication and computation. However, when the dependency between communication and computation are complex and evolve over the course of the execution, their implementation becomes tedious and may lead to bugs. In this paper, focusing on spatial simulation, we propose a Distributed Cell Set Library that manages the distribution of elements positioned in the space into divided area units (*cell* units). We make it possible to manage the granularity of cells with which communication may overlap with computation and for multithreaded computation. In addition, we make it possible to describe the relationships between inter-node communication and the pending computation in cell units. For evaluation, we introduce the implementation of a two-dimensional molecular dynamics simulation using our library. We show that using our computation and communication overlapping method, the delays to reach global synchronization that are necessary in the original implementation can be avoided.

CCS CONCEPTS

• **Computing methodologies → Parallel computing methodologies**; **Distributed computing methodologies**; **Parallel programming languages**.

KEYWORDS

distributed data structure, communication/computation overlap, many core cluster

1 INTRODUCTION

Parallel & distributed programs require implementations that take advantage of the large number of cores available in modern processors. One such method consists in overlapping communication and calculation that do not depend on each other to eliminate bottlenecks due to communication. So far, this has generally been implemented by combining a library for inter-node communication with a library for thread parallelism.

However, implementing overlap is still a challenge for programmers due to the complex dependency relationships between the computation and its necessary data. The computation should be defined with respect to each piece of data with a certain degree of granularity, defining whether it can overlap with communication and how it can be processed through intra-host parallelism. Moreover, those dependencies should be able to adapt to support flexible distributions. This can become a considerable burden for programmers who risk obfuscating their program. We believe such challenges should be handled by libraries instead.

In this paper, we present a distributed cell set library, which is introduced in our distributed collections library [4]. The distributed collections library provided distributed array management in index range units (*chunk* units). The distributed cell set library we present here focuses on spatial simulation. It manages the distribution of elements with respect to their position in the space divided into unit areas (*cell* units). The library manages the element migration between cells by checking the positions of elements. In addition, the library provides asynchronous inter-node communication functions to enable inter-node element migration and remote cell caching. Our most significant contribution is that these communication functions make it possible to easily and briefly describe the relationship between inter-node communication and their dependent computation in cell units. The `CompletableFuture` paradigm implemented in Java allows programmers to describe their dependencies with great flexibility.

The remainder of this article is organized in the following sections. We start by introducing the motivation case for our library in Section 2. In Section 3, we introduce some necessary background. In Section 4, we discuss the design of our library before discussing its implementation in Section 5. We present the results of our evaluation in Section 6. In Section 7, we discuss related work. We then conclude and discuss future work in Section 8.

2 MOTIVATION CASE: MOLDYN

In this article, we use the Molecular Dynamics (*MolDyn*) simulation from the Java Grande benchmark suite [11] as a running example of how to use our distributed cell set library. This section describes MolDyn's parallelization method and dependencies for communication-computation overlap.

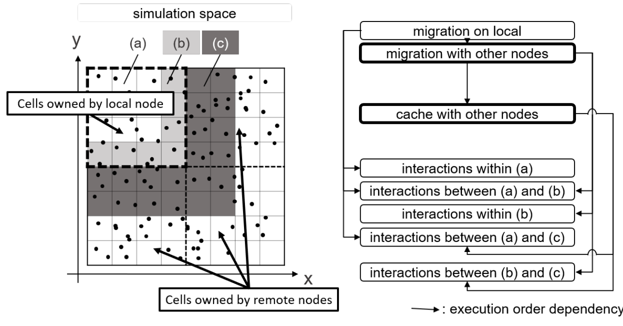

Figure 1: MolDyn area classification and dependencies on each processing

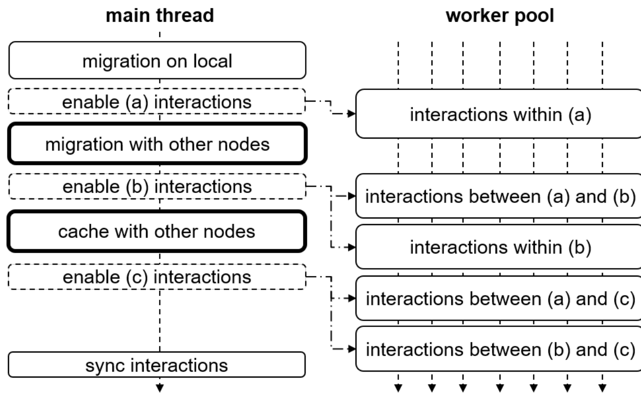

Figure 2: MolDyn overlap example with multiple threads

MolDyn simulates the evolution of a system of particles by calculating the interaction forces between the particles in the simulation space. After deriving the interactions forces, the position of each particle is updated and the cycle is repeated.

Parallel/distributed computation of MolDyn is generally implemented by decomposing the simulation space into into cells and distributing these cells to the compute nodes. The particles contained in each cell performs the interaction calculations with the particles contained in other cells. Each node needs the information of the cells involved in the interaction. If these cells are handled by a remote node, the node collects the necessary information from the remote cell. Furthermore, if a cut-off distance is considered in the interaction calculation, the cells to be collected are restricted the surrounding area (remote "Halo" area) of the cells owned by the node. Each particle updates its position after deriving the force exerted on itself by other particles. Particles “migrate” between cells according to the updated positions, involving communication when particles migrating to a remote cell. Following this method, processing for cells in each node consists in three steps: *Particle Migration*, *Halo Cache* and *Interaction computation*.

In MolDyn, the higher the number of nodes, the shorter the interaction computation time. As the time for communication time does not decrease significantly, its proportion increases. As a countermeasure, it is possible to overlap the computation and communication to hide the communication time.

In order to overlap computation and communication, the programmer must classify the processing of each step for each cell, organize the dependencies correctly, and implement it. Interaction dependencies are classified for each cell as shown in Figure 1. With respect to the node responsible for the cells in the top left corner, Area (a) shows cells without concern for particles migrating from other nodes, area (b) are the cells where migrated particles may come from other nodes, and area (c) shows cells owned by other nodes that need to be cached.

Figure 2 represents the dependencies on Migration, Cache Halo, and Interaction for each area. The timing at which each region interaction calculation becomes possible is as follows:

- *area(a)*: possible after local migration is completed
- *area(b)*: possible after migration communication with remote nodes
- *area(c)*: possible after caching

In the example presented in Figure 2, realizing overlaps between communication and calculation is performed by entrusting the main thread to perform the migration and cache communications while the interactions calculation is executed by workers extracted from the worker pool.

That dependencies are complicated. In addition to the above (a), (b), and (c) area decomposition, the (b) and (c) areas can be further decomposed. For instance, “as soon as the communication from the lower area ends,” or “as soon as the communication from the right area ends,” and so on. While it is possible to express every individual cell-to-cell dependency, this makes the program more complicated. There are other kinds of dependencies, communication-to-communication ones. In this example, cache communication is executed after inter-node migration become completed.

This implementation becomes complicated. These communication tasks may be scheduled in sequence, but each communication task must be connected to the corresponding cell and trigger dependent computations. For interactions between two cells, the calculation can only be triggered when both cells are ready for calculation. Managing such dependencies can become troublesome.

3 BACKGROUND

3.1 Distributed Collections Library

In this section, we recall the elements concerning our distributed collections library on which this work is based. More details concerning this work can be found in [4].

Our library complements the Asynchronous Partitioned Global Address Space (APGAS) model as introduced in X10 [13] and later ported to Java [12]. Under this model, a process running on a compute node is called a `Place`. In this article, we assume that a single process executes per compute node. The terms “process,” “place,” and “node” can therefore be used interchangeably in this context. Asynchronous activities can be launched on a remote place with method `asyncAt`. Termination detection is implemented through an elegant method called `finish`, which only returns when all transitively spawned asynchronous activities complete.

```
1  TeamedPlaceGroup world =
2      TeamedPlaceGroup.getWorld();
3  DistMap<String, String> dMap =
4      new DistMap<>(world);
5  dMap.put("main", "running");
6  world.broadcastFlat(()->{
7    dMap.put(here(), "says hello");
8    CollectiveMoveManager mm =
9        new CollectiveMoveManager(world);
10   if (here() == place(0)) {
11     dMap.moveAtSync("main", place(1), mm);
12   }
13   mm.sync();
14 });
```

Listing 1: Distributed map creation and record insertion

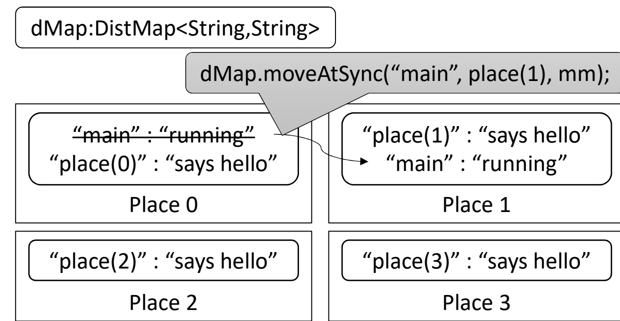

Figure 3: State of the distributed map dMap after the Listing 1 program has run with 4 processes

In our library, we provide a number of distributed collections (distributed map, distributed array). Each distributed collection is implemented as a group of local handles located on each process and linked together by a globally unique ID. Programmers can record and read entries into the local handle of a distributed collection by using the usual APGAS asynchronous activities. It is also possible to relocate entries between local handles at the programmer's initiative, with entries to relocate described by their keys in the case of a distributed maps, or by ranges in the case of distributed arrays.

The sample program presented in Listing 1 illustrate these characteristics. The `TeamedPlaceGroup` object obtained on the first line of Listing 1 represents the group of all APGAS places. It is used on line 3 to create the distributed map `dMap`, meaning that every process participating in the execution will have a handle for this distributed collection. On line 5, a first entry is recorded into the map by the main thread, resulting in the `main:running` mapping to be recorded on the first process.

From lines 6 to 14, the `broadcastFlat` method is used to spawn the same asynchronous activity on all hosts participating in the computation. This short-hand replaces the otherwise explicit `finish` and for loop needed to spawn the same activity on all places, enhancing the clarity of the program. The asynchronous activity given as parameter to method `broadcastFlat` makes every place add a new entry into their local handle of the distributed map `dMap` on line 7. Then, a collective relocator is created on line 8. This object is used to register various elements of distributed collections to be transferred from a handle to another. In this case, only the first place relocates its `main:running` entry to Place 1. The transfer is performed on line 13 when the `mm.sync` method is called by all the places participating in the computation. The final state of the distributed map `dmap` is shown on Figure 3. Notice in particular that the `main:running` entry has been removed from the handle on Place 0 and inserted into the handle of Place 1.

A number of computation patterns that require communication and computation are also supported, such as reductions. We call such patterns that involve the participation all the local handles of a distributed collection "teamed." Internally, these communication patterns are supported by MPI through OpenMPI Java bindings [15].

```
1  ExecutorService executor = new ForkJoinPool();
2  // submit task and create Future
3  CompletableFuture<Integer> cFuture1 =
4      CompletableFuture.supplyAsync(() -> {
5      return someFunc1();
6  }, executor);
7  // chain asynchronous task and create Future
8  CompletableFuture<String> cFuture2 =
9      cFuture1.thenApplyAsync((Integer result1) -> {
10     return someFunc2(result1);
11 }, executor);
```

Listing 2: Sample program for CompletableFuture

3.2 Completable Future

In this section, we summarize the usage of the `CompletableFuture` class in Java. Our distributed cell set relies on the features of this class and the user of the library should be familiar with it to use the asynchronous feature of our library.

`CompletableFuture` was introduced to Java as part of the Java 8 Concurrency API improvement. It is an extension of `Future`, which allows explicit completion of asynchronous operations and flexible description of processing dependencies using method-chaining. An illustrative program is shown in Listing 2. The `supplyAsync` method (lines 3-6) submits an asynchronous task doing `someFunc1()` and returns future `cFuture1` to the caller. The user can specify the `ExecutorService executor` (line 1) that executes the asynchronous task. If the executor is omitted, the `ForkJoinPool.commonPool()` will be used by default.

Then the `thenApplyAsync` method (lines 8-11) creates a new future that executes `someFunc2(result)` with the result of `cFuture1` as the argument `result1`. Unlike normal Futures, this method instantly returns a `CompletableFuture` instance without waiting for the result of `cFuture1`.

Many utility functions are also provided. `thenApplyBothAsync` waits for the results of two futures. Methods `allOf(futures..)` and `anyOf(futures..)` return a future that respectively waits for all or one of the given futures. Programmers can therefore describe complex dependencies of asynchronous processing in an easy-to-understand API.

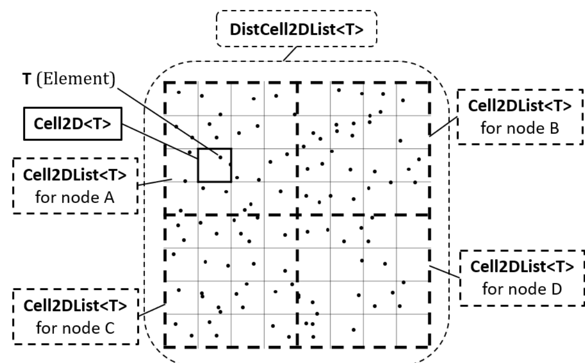

Figure 4: Object layout in a distributed cell set

4 DESIGN

In this section, we present the design of our distributed cell set library. We show how programmers can describe space-dependent communication and computation overlap using the molecular dynamics program presented in in Section 4.1 as a running example. We then describe the detailed specification of our library in Section 4.2.

4.1 General implementation in MolDyn

While the original MolDyn version adopts a 3D model, we present here a 2D version of the simulation as, currently, our library only supports 2D cells. Figure 4 presents a sample object layout in the distributed cell set. Cell2D<T> represents a determined 2D area containing some elements, the type of which is specified by parameter T. Cell2DList<T> and DistCell2DList<T> are the local and distributed versions of Cell2D<T> sets. They both receive a 2D area and split this area into cells gathered in a set. The split cells are identified through an integer id. In addition, class DistCell2DList<T> can distribute the cells over compute nodes.

The elements contained in the cells must implement a position() method returning a Position2D object. When the migration method is called, our library checks the elements' position and migrates those that have moved to different cell. When elements move to cells assigned to other compute nodes, DistCell2DList<T> gathers such elements and performs the transfer.

The DistCell2DList<T> class provides two kinds of methods for communication: migrate() for element migration, and cacheHalo() for caching elements. To enable asynchronous communication between compute nodes, both have CompletableFuture variants. These methods receive future objects on which they have to wait for completion to perform the communication, and return future objects to notify of the completion of their communicating tasks. In the case of method thenMigrateAsync(), this receives a function that changes elements' position. The method first applies the function to all the local elements, gathering the elements that move to different cells, and then proceeds to perform element migration within the local cells. The method returns a collection of future objects, each representing the status of migration acceptance to a cell, before completing its own communication process. Its communication starts only after confirming the completion of the dependent communication tasks as specified by the arguments. Our library manages the migration process in each cell and notifies the completion of the acceptance by cells. Method thenCacheHaloAsync() instantly returns a collection of future objects, each representing the status of cache acceptance in each cell. The communication process is started after confirming the completion of the dependent tasks.

Using these abstractions, we can describe the communication and computation dependencies described in Section 2. The molecular dynamics program has two main operations, interact() which consists in calculating the force interactions between the particles, and domove() which is used to update the velocity and positions of the particles. The former processes parallel read access to particles and the latter is executed in an owner-compute pattern on the particles. In addition, the forces calculated for each cell pair must be summed up for each cell. We use an "accumulator" mechanism to handle this pattern which we will discuss further in the next subsection.

Listing 3, presents the main procedure of the MolDyn program written with our library and Figure 5 represents the dependency between communication and computation in the program. The program first performs the update operation and then proceeds with the force calculation. cells (line 8) is an instance of DistCell2DList and contains all the cells, distributed across nodes. The migrateAsync() method is a variation of the thenMigrateAsync() method that does not wait for any future and returns a future linked to the created asynchronous task. In the migrateAsync() method, cells applies the domove() method on the particles, processes the intra-node particle migrations, and then returns the future objects cellsStage2 that represent the completion of the migration process of for its respective cells, triggering the inter-node element migration. These cellsStage2 are accessed by two operations: the cache operation (lines 16-19) and the force calculation (lines 24-39).

For the cache operation, the cellsStage2 are reduced to a single future object allOfStage2 (lines 11-14), and given to thenCacheHaloAsync() method. Thus, the cache operation is scheduled after all the migration processes are completed. The thenCacheHaloAsync() method returns the future objects cachedCells that represents the completion of the cache acceptance for remote cells.

In the case of the force calculation, the elements of cellsStage2 (future0 on line 24 and cellsStage2.get(cellId1) on line 29), and the elements of cachedCells (cachedCells.get(cellId1) on line 34) are given to the interactFuture() method defined by the programmer. The method interactFuture(f0, f1, executor) (line 28, 33) receives two CompletableFuture<Cell2D<Particle» f0, f1 and an executor for task execution. Here, f0 and f1 represent the completion of particle migration of local cellular spaces or the completion of cache acceptance of the remote cellular spaces. The interactFuture() instantly returns a future object representing the completion of the force calculation (interact()) for each pair of cells.

```
 1 Collection<Future<Void>> allFutures
 2     = new ArrayList<>();
 3 ExecutorService executor =
 4     ForkJoinPool.commonPool();
 5 // migrate async
 6 Map<Integer, CompletableFuture<Cell2D<Particle>>>
 7     cellsStage2 = cells.migrateAsync(
 8     domove, MAX_MGN_DISTANCE, EDGE_TYPE,
 9     commType, executor);
10 // reduce cellsStage2 to one
11 CompletableFuture<Void>
12     allOfStage2 = CompletableFuture.allOf(
13     cellsStage2.values().toArray(
14     new CompletableFuture[cellsStage2.size()]));
15 // cache async
16 Map<Integer, CompletableFuture<Cell2D<Particle>>>
17     cachedCells = cells.thenCacheHaloAsync(
18     allOfStage2, CUTOFF, EDGE_TYPE,
19     commType, executor);
20 // create accumulator
21 accumulator = new Cell2DAccumulator<>(cells,
22     (p) -> new ParticleAccumulator(p));
23 // do force calculation
24 cellsStage2.forEach((cellId0, future0) -> {
25     cells.forEachNeighborsId(cellId0, CUTOFF,
26     EDGE_TYPE, (cellId1) -> {
27         if (cellsStage2.containsKey(cellId1)) {
28             Future<Void> f = interactFuture(
29             future0, cellsStage2.get(cellId1),
30             executor);
31             allFutures.add(f);
32         } else {
33             Future<Void> f = interactFuture(
34             future0, cachedCells.get(cellId1),
35             executor);
36             allFutures.add(f);
37         }
38     });
39 });
40 // sync
41 for (f : allFutures) {
42     f.get();
43 }
```

Listing 3: Sample Application (MolDyn)

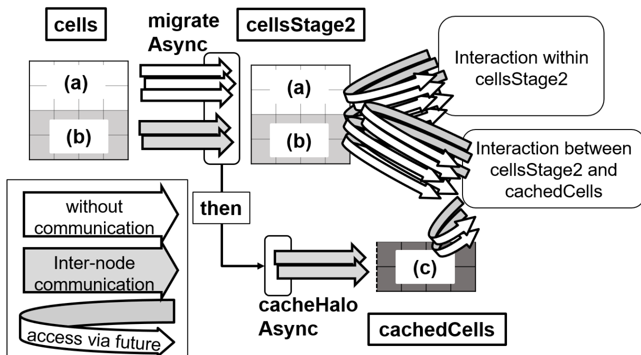

Figure 5: Dependency Representation (MolDyn)

4.2 Library Specification

In this section, we present a more formal specification of our library.

Creation: When initializing a distributed collection managing the cellular spaces, the programmer provides the range of the target 2D area and the numbers of splits in each dimension. The distributed collection object is initially created on the node where the constructor is called and exported to other nodes when the handle to this distributed collection is used in a remote node as part of an APGAS activity. The allocation of the local handle in the remote node is made as part of the deserialization process of the asynchronous activity. The Programmer can initially create all the cellular spaces on the creation node and then arbitrarily distribute these spaces over nodes using collective communication.

Migration: The migration functions receive four arguments: a function that changes the position of elements, an upper bound to the distance that any individual element may travel, whether the 2D area is organized as a torus, and the communication method used for the relocation. Currently, we have two communication implementations, either relying on MPI `allToAllV` collective communication or `sendRecv` one-to-one communication. The `thenMigrateAsync()` methods returns the list of futures that represent the completion of the migration process for each cell, including the acceptance of elements on the remote nodes. Cells that have no inter-node element migration will see their corresponding future already completed when the `thenMigrateAsync()` method returns.

Cache Halo: The cache method receives three arguments: the width of the halo area, whether the 2D area is organized as a torus, and the communication method used for the relocation. The last two parameters are identical to the migration method described above. The `thenCacheHaloAsync()` method return the list of futures that represent the completion of the cache acceptance for each cell residing on remote nodes. When a node receives cache data for a cell and deserializes it, the node notifies the completion of this acceptance via the corresponding future, releasing the depending tasks.

Accumulator for reduction operation: Our distributed collection library offers *accumulators* to reduce the results from multiple threads. The accumulators are used to hold intermediate results generated by threads with respect to a particular entry in a collection. We provide accumulators for class `Cell2D`, which is used to reduce forces in the `MolDyn` program. Listing 4 shows the use of the accumulator mechanism in the *MolDyn* program. `Cell2DAccumulator<P,A>` has two type parameters: `P` for the targeted elements and `A` for the intermediate values. The constructor of `Cell2DAccumulator` receives two arguments, the `Cell2DList<P>` instance containing the elements targeted by the accumulator, and a function to create a `A` instances from a `P` instance. Each thread obtains its dedicated accumulator instance by calling `acc.createAndGet(cell)`. The instance is created on demand if the thread has not previously called this method.

5 IMPLEMENTATION

In this section, we present some implementation topics of our library.

```
// Accumulator Definition
Cell2DAccumulator<Particle, ParticleAccumulator>
    acc = new Patch2DAccumulator<>(cellList,
    (p) -> new ParticleAccumulator(p));

// Accumulator Usage
void interact(Cell2D<Particle> cell0,
    Cell2D<Particle> cell1) {
  acc.createAndGet(cell0).forEach((p0) -> {
    cell1.forEach((p1) -> {
      interact(p0, p1);
    });
  });
}
```

Listing 4: Accumulator for Cell2D

Multithreading: In our prototype implementation of `DistCell2DList`, we mainly rely on Java's `CompletableFuture` and `Executor` for the multithreaded features. `DistCell2DList` splits their tasks by cell and executes their tasks using the asynchronous methods of `CompletableFuture`, specifying their executor. In the molecular dynamics program, `ForkJoinPool.commonPool()` is used as the executor. As other classes of our distributed collections also rely on the multithread features of the APGAS library, we will check the compatibility between these implementations.

Data dependency management between cells: `Cell2DList` and `DistCell2DList` manages their cells using a list, mapping a 2D position to an integer index. Thus, they can find the adequate cell for an elements within $O(1)$ time. `DistCell2DList` can check the latest assignment of cells to compute nodes using the distributed tracking feature of our distributed collection library, including in cases where cells are relocated between nodes. Data dependencies between cells are determined by the distance between them. For halo caching, each node finds out the remote cells that exist in the halo region of local cells and sends copies of the concerned cells to the owner of the remote cells. In the case of element migration, each node first gathers the elements that need to transfer to remote cells and sends them to the owner of the corresponding remote cells. To enable fast complete confirmation of migration acceptance, our library checks the number of compute nodes susceptible to send migration elements to each cell. When each cell receives the expected number of migration messages from all such nodes, it notifies the completion of the migration acceptance to the depending tasks.

Communication with multiple nodes: As mentioned in section 4, we currently provide two communication patterns using MPI `sendRecv` and `allToAllV`. We used the `sendRecv` version in the performance evaluation in Section 6, both for communication overlap and no overlap. In this implementation, our system builds a binary tree with the compute nodes as leaves. After determining the communication counterparts, each node sorts its counterparts using the distance in the graph and performs `sendRecv` operations in that order. As the communication order might change by communication methods, the common interface of the communication method offers a method that returns a `CompletableFuture<ByteArrayInputStream>` for each node. Using this future mechanism, our library can cope with any arbitrary scheduling order of communication.

Table 1: Program parameters used for MolDyn

Property	Value
cell size	x:1.0 / y:1.0
nb particles / cell	19 to 20
cut-off distance	1.0
time per 1 cycle	0.005

Notification of communication completion: In the future version of element migration and halo caching, these methods first prepare the list of future objects for local and cached cells, respectively. Using the underlying communication layer, it conducts communication with multiple nodes asynchronously. Each communication is followed by its deserialization process and the future objects will be notified of the communication completion immediately.

Class Hierarchy: `DistCell2DList` is a subclass of `Cell2DList` and implements interface `DistCell2DCollection`, which brings various default methods, including element migration and halo caching. This will be helpful for future extensions to various data structures other than `Cell2DList`. We hope to add more data structures such as hierarchical cell data structures.

6 EVALUATION

In the following sections, we evaluate the strong scaling and weak scaling behavior of MolDyn over 50 iterations and compare the performance with and without computation/communication overlap. We analyze the time taken for each step within an iteration. Finally, future issues identified from the experimental results are presented.

In the non-overlap implementation, Migration and Cache Halo are executed only in the main thread whereas in the overlap version, these steps are performed by one thread belonging to the executor. The interaction calculation uses futures in both implementations, forking asynchronous tasks for each interaction pair, allowing this computation to be performed in parallel.

The MolDyn settings used in the experiments are shown in Table 1. By initially placing the same number of particles in each cell, the particles are distributed evenly throughout the space. In weak scaling, the space size is set so that the density of particles remains constant. In MolDyn with fixed density and cut-off, the total interaction calculation is $O(n)$ (where n is the number of particles) and the computational complexity for each node is $O(n/p)$ (where p is the number of nodes). Cell distribution is done with a simplistic strategy consisting of separating the 2D space in as many horizontal strips as there are nodes in the computation.

Experiments in both strong scaling and weak scaling are performed on the Fugaku Supercomputer. The details of the hardware and software environment used are summarized in Table 2. Computation is performed with a maximum of 48 threads per node. The large number of executions necessary for this study was managed using OACIS [5].

Table 2: Hardware and Software environment on the Fugaku supercomputer

Property	Value
Processor	FUJITSU Processor A64FX (48 cores)
Memory	HBM2 32 GiB, 1024 GB/s
Java version	OpenJDK 11.0.2
MPI	Fujitsu MPI (based on Open MPI) Java Binding

6.1 Strong Scaling

Figure 6 compares the performance of the overlap and no-overlap versions of MolDyn on up to 16 compute nodes for 50 iteration. The *ideal* line shows the theoretical perfect scaling where the computation time gets exactly *n* times shorter as a result of using *n* nodes compared to the reference 1 node execution. The solid lines represent the execution times and the dotted lines represent the efficiency compared to *ideal*. The execution time of *overlap* is up to 34.1% shorter than *no overlap* (8 nodes: overlap 14682 ms / no overlap 22271 ms). Looking at the efficiency, the version without overlap drops to under 40% for 16 nodes executions. The version with overlap consistently maintains a higher efficiency, only dropping to 53.9% on 16 nodes executions.

However, this performance improvement is not due solely to communication time concealment. Figure 7 shows the average time required for each part of the iteration. The *migration* is the time including both intra-node and inter-node migration. In overlap, *fork* includes the time spent in the main thread in `migrateAsync` and `thenCacheHaloAsync`, as well as the interaction computation fork and the intra-node migrations. The *sync before migration* corresponds to the time taken for synchronization between nodes before performing inter-node migration. Since this part is performed asynchronously in the overlap version, this time is hidden and the iteration time is shortened for this version of the program as a result.

We tried to identify what causes the synchronization time to be longer than anticipated. We measured the number of particle interactions at each node in an 8-node MolDyn execution. The difference between the maximum and minimum interaction numbers was up to 53274 for the maximum interaction number of 2480188. This means that the particles are evenly distributed between nodes and that particle unbalance is not the cause of this issue.

We are unsure as to what causes the hosts to reach the synchronization point in slightly shifted timing, delaying the whole program. We suspect external factors such as garbage collection, but further investigation is needed.

6.2 Weak Scaling

The Table 3 shows the execution time of 50 MolDyn iterations for both the overlap and no-overlap versions in weak scaling. The percentages in the table represent the efficiency of the 8 node execution compared to the 2 node execution. In the case of no overlap, only 64.1% of the performance is preserved even though the computation amount of each node does not change significantly. On the other hand, using the overlapping version, 80.6% of the 2-node performance is kept. The overlapping version scales better than the no overlap version.

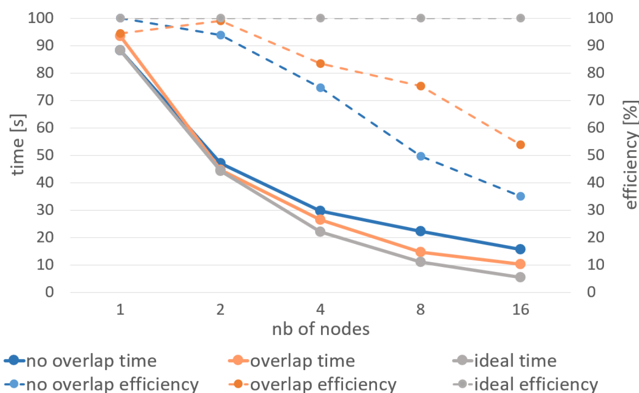

Figure 6: Performance scaling with 320,000 particles 50 cycle with and without overlap

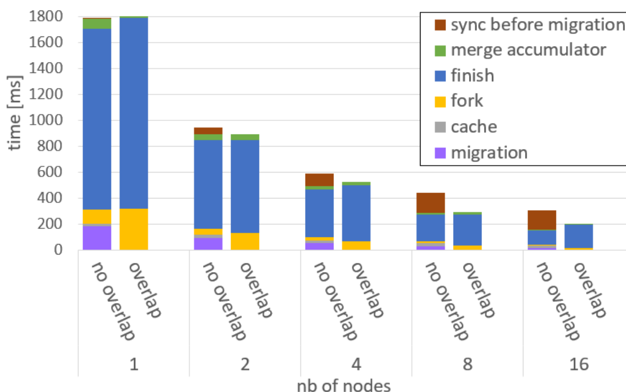

Figure 7: Each step execution time with 320,000 particles 1 cycle with and without overlap

Table 3: Execution time of MolDyn 50 cycle : particles and parallelism scaling

nb of nodes	2	8
particles	80000	320000
no overlap	14285 ms	22271 ms (64.1%)
overlap	11836 ms	14682 ms (80.6%)

6.3 Future Issues

We witnessed cases where cache halos and interaction calculations were not overlapped depending for some nodes. We are investigating the possibility that worker threads are occupied by computation, causing communication to be delayed until the computation is completed. Some applications may require some sort of priority mechanism to favor some operations over others.

When running the MolDyn program using yet larger number of particles, we experienced unexplained crashes. We suspect issues

in the buffer allocations for the MPI communications. This issue is currently under investigation.

7 RELATED WORK

The communication and computation scheduling has generally been implemented by HPC programmers explicitly managing inter-process communication and thread parallelism. On the other hand, some libraries were developed to implicitly manage communication and computation schedules. Our library fits in the latter category and aims at easing the programming cost of explicit management of the cell-based computation decomposition and the space-dependent communication/computation overlap.

Charm++ [1] proposes a unified programming model for parallel and distributed computation. Charm++ features its dynamic load-balancing, using *chare* as the relocatable computing unit. The system profiles the communication graph between chares and automatically finds out the adequate assignment of chares to processing units. Programmers are encouraged to conduct over-decomposition of problems and rely on the dynamic load-balancing feature. NAMD [7] is a molecular dynamics program designed for HPC environments and a very successful application sample for Charm++. The simulation space is divided into small boxes called *patches* and the computation is represented as a set of *computes* objects. The data are delivered by message passing between these objects. To enable its sophisticated data flow in NAMD, the program explicitly describes the particle management using patches and defines the data flow between patches and compute objects in an event-driven manner.

In XscalableMP (XMP) [6], compiler directives for C and Fortran allows programs to be distributed and parallelized automatically. XMP support distributed arrays and the notion of "shadowing." Given a distributed array assigned to nodes with a block-cyclic distribution, this mechanism allows parts of the arrays located at the edge of a node to be "reflected" on the compute nodes that own the neighboring indices. Given a nested for loop, if the computation needs to access data hosted by a different process, the compiler directives of XMP are capable of generating the code to access data points which may be located on remote hosts automatically. In our library, the caching system allows programmers to obtain similar effects. The main difference with XMP lies in the fact that the data dependencies are expressed through futures rather than compiler directives.

Our library splits the space into cells and allows programmers to write dependencies between their communication and computation tasks. The task splitting itself is widely used for load-balancing, especially on shared-memory parallel computers. For instance, the Java Stream API supports parallel streams through which programmers can easily split a stream and leave each computation to respective threads. Cilk [2] is a successful programming language designed for shared-memory computers. The tasks themselves do not have any explicitly defined data structures, and programmers can spawn child tasks like function calls. Charm++, X10 GLB [9, 17], RDMA-based continuation stealing [10] are designed for distributed systems. RDMA-based continuation stealing adopts Cilk-style programming based on the uni-address scheme, but it does not support distributed data structures yet. In the original GLB, programmers define bags of self-contained tasks and thus do not support computation made on persistent data structures such as distributed collections. We have worked on integrating our distributed collection library with the GLB scheme [3] and hope this distributed cell set library can be combined with these load balancing features using cells as units of computation in the future.

For large-scale data analysis, the *MapReduce* programming model is often used, which is designed for large datasets distributed over a number of nodes. Hadoop [14] or Spark [16] adopt this model. It consists of map operations, which are executed under the owner-compute rule, and reduce operations, which involve the shuffling of computed results of map operations. The data distribution and communication are generally managed by the system automatically, and programmers only write the data dependencies and the computation to perform using the programming model.

Our library used `CompletableFutures` to describe dependencies between communication and computations tasks. It is often used to implement the observer pattern, in which the *subject* maintain its *observers* and notify their state changes to the observers. This pattern resembles to the publish-subscribe message pattern. The observer pattern is often used to implement asynchronous stream processing. ReactiveX [8] is an API for asynchronous stream processing using Observer pattern and is widely used for GUI event handling or asynchronous server-client cooperation.

8 CONCLUSION AND FUTURE WORK

In this article we presented the distributed cell set library integrated into our distributed collection library. This library makes is easier for programmers to explicitly manage cell-based computation decomposition and the space-dependent communication and computation overlap. We re-implemented *MolDyn* from the Java Grande benchmark suite and evaluated it on a many-core cluster. We showed a maximum performance improvement of 34.1% by overlapping communication and computation compared to the version of our program which does not rely on communication/computation overlap. This is thanks to the circumvention of a globally synchronizing operation which causes performance issues.

In future work, we are planning to evaluate MolDyn with flexible distributions and extend it to 3D by developing 3D cells. Moreover, we would like to develop and evaluate other applications. We would also like to combine this approach with dynamic load balancing.

ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Numbers JP20K11841 and JP22H03585. This work used computational resources of supercomputer Fugaku provided by the RIKEN Center for Computational Science through the HPCI System Research Project (Project ID: hp220190 and hp220334).

REFERENCES

[1] Bilge Acun et al. 2014. Parallel programming with migratable objects: charm++ in practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '14). IEEE Press, New Orleans, Louisana, 647–658. ISBN: 9781479955008. DOI: 10.1109/SC.2014.58.

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30, 8, (Aug. 1995), 207–216. DOI: 10.1145/209937.209958.

[3] Patrick Finnerty, Tomio Kamada, and Chikara Ohta. 2022. Integrating a global load balancer to an apgas distributed collections library. In *Proceedings of the Thirteenth International Workshop on Programming Models and Applications for Multicores and Manycores* (PMAM '22). Association for Computing Machinery, Seoul, Republic of Korea, 55–64. ISBN: 9781450393393. DOI: 10.1145/3528425.3529102.

[4] Patrick Finnerty, Yoshiki Kawanishi, Tomio Kamada, and Chikara Ohta. 2022. Supercharging the apgas programming model with relocatable distributed collections. *Scientific Programming*, 2022, 1058-9244. DOI: 10.1155/2022/5092422.

[5] Y. Murase, T. Uchitane, and N. Ito. 2017. An open-source job management framework for parameter-space exploration: OACIS. *Journal of Physics: Conference Series*, 921, (Nov. 2017), 012001. DOI: 10.1088/1742-6596/921/1/012001.

[6] Masahiro Nakao, Jinpil Lee, Taisuke Boku, and Mitsuhisa Sato. 2010. Xcalablemp implementation and performance of nas parallel benchmarks. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model* (PGAS '10) Article 11. Association for Computing Machinery, New York, New York, USA, 10 pages. ISBN: 9781450304610. DOI: 10.1145/2020373.2020384.

[7] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. 2002. Namd: biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* (SC '02). IEEE Computer Society Press, Baltimore, Maryland, 1–18. ISBN: 076951524X.

[8] 2022. Reactivex, an api for asynchronous programming with observable streams. Retrieved Dec. 14, 2022 from https://reactivex.io/.

[9] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. 2011. Lifeline-based global load balancing. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (PPoPP '11). Association for Computing Machinery, San Antonio, TX, USA, 201–212. ISBN: 9781450301190. DOI: 10.1145/1941553.1941582.

[10] Shumpei Shiina and Kenjiro Taura. 2022. Distributed continuation stealing is more scalable than you might think. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, 129–141. DOI: 10.1109/CLUSTER51413.2022.00027.

[11] L. A. Smith, J. M. Bull, and J. Obdrzálek. 2001. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (SC '01). Association for Computing Machinery, Denver, Colorado, 8. ISBN: 158113293X. DOI: 10.1145/582034.582042.

[12] Olivier Tardieu. 2015. The apgas library: resilient parallel and distributed programming in java 8. In *Proceedings of the ACM SIGPLAN Workshop on X10* (X10 2015). ACM, Portland, OR, USA, 25–26. ISBN: 978-1-4503-3586-7. DOI: 10.1145/2771774.2771780.

[13] Olivier Tardieu, Benjamin Herta, David Cunningham, David Grove, Prabhanjan Kambadur, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Mandana Vaziri. 2014. X10 and apgas at petascale. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP '14). ACM, Orlando, Florida, USA, 53–66. ISBN: 978-1-4503-2656-8. DOI: 10.1145/2555243.2555245.

[14] Vinod Kumar Vavilapalli et al. 2013. Apache hadoop yarn: yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (SOCC '13) Article 5. Association for Computing Machinery, Santa Clara, California, 16 pages. ISBN: 9781450324281. DOI: 10.1145/2523616.2523633.

[15] Oscar Vega-Gisbert, Jose E. Roman, and Jeffrey M. Squyres. 2016. Design and implementation of java bindings in open mpi. *Parallel Computing*, 59, 1–20. Theory and Practice of Irregular Applications. DOI: 10.1016/j.parco.2016.08.004.

[16] Matei Zaharia et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59, 11, (Oct. 2016), 56–65. DOI: 10.1145/2934664.

[17] Wei Zhang, Olivier Tardieu, David Grove, Benjamin Herta, Tomio Kamada, Vijay Saraswat, and Mikio Takeuchi. 2014. Glb: lifeline-based global load balancing library in x10. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications* (PPAA '14). Association for Computing Machinery, Orlando, Florida, USA, 31–40. ISBN: 9781450326544. DOI: 10.1145/2567634.2567639.