Parallel simulation of scattering analysis on a shared dataspace

Janssen, D.F.B.W.

Award date:
2007

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computing Science

MASTER'S THESIS

Parallel simulation of scattering analysis on a shared dataspace

by

D.F.B.W. Janssen

Supervisor:

R.H. Mak (TU/e)

Eindhoven, July 2007

Abstract

Computer simulation of physical experiments has proven itself as a valuable method to perform scientific research. Due to complex computations and/or large problem sizes, these experiment simulations can become computationally intensive. Furthermore, simulation software is often bound to specific hardware and/or software environments and therefore has limited portability.

In this master's thesis, an object-oriented model for parallel experiment simulations is presented. The implementation of this model provides a framework to create simulations (for a class) of experiments that can be modeled as functions.

An example of a computationally intensive experiment was found in the domain of scattering analysis: based on the framework, a parallel simulation program was developed which performs scattering experiment simulations in parallel. For inter-process communication, an implementation of the *Tuple Space* paradigm is used, which leads to simulation software that is both scalable and portable.

Acknowledgements

With the completion of this master's thesis, I finally arrived at the destination of what has become a very long journey. I am deeply grateful to all the people that helped me along the way.

I would like to thank Michel R. V. Chaudron and Peter A.J. Hilbers for their participation in my examination committee, Richard Verhoeven for his technical support, Giovanni Russello for helping me getting started with *GSpace* and Marcel Koonen for the many interesting and pleasant discussions.

Most of all, I want to express my gratitude to Jaap van der Woude, who helped me get started on this journey and get back on the road when I went astray, and to my supervisor Rudolf Mak, without whose support, encouragement, patience and guidance I would have never arrived at this destination.

Table of Contents

Chapter 1

Introduction

This chapter lays the foundation for the work presented in this master's thesis; it describes the background of and the motivation for the work.

1.1 Experiments

In the scientific method, experimentation plays a central role. An experiment is defined in [Oxf96] as *a controlled manipulation of events, designed to produce observations that confirm or disconfirm one or more rival theories or hypotheses.*

But opposed to putting the questions to nature, advancements in computer science have made it possible to perform experiments *in silico*; in computer experiments, a model of a phenomenon is simulated and the obtained results are used to make inferences about the underlying system.

In this way, computer experiments may seem to be of limited use because of the uncertainty they introduce: the model is an imperfect representation of the underlying system and the obtained results can be inexact (e.g. when the model is stochastic, or requires numerical approximation). The advantage of performing experiments on computers however, is that it allows to do experiments that —when performed on the real system— would be

- too dangerous, because of its effects on health or environment (e.g. a nuclear detonation, or testing the effects of a new drug);
- not feasible, for example experimenting under conditions that cannot easily be attained (e.g. under extreme low/high temperatures, in the absence of gravity, small/large time scales);
- too theoretical: to gain more insight into a theory, one can perform experiments that are purely theoretical.

In the next section one particular type of experiment will be presented in more detail. This family of experiments is the subject of the simulations presented in the following chapters.

1.2 X-ray scattering analysis

X-ray scattering experiments are widely used in (bio-)chemistry to determine the structure of a wide range of particles or aggregates (e.g. molecules, crystals or gels). To perform this type of experiment, a beam of X-ray radiation is directed to a substance of particles (which can be anything from single atoms to large proteins or even polymers) and a spectrum, which results from the rays being diffracted, is observed. Analysis of this spectrum reveals information about the structure of the substance.

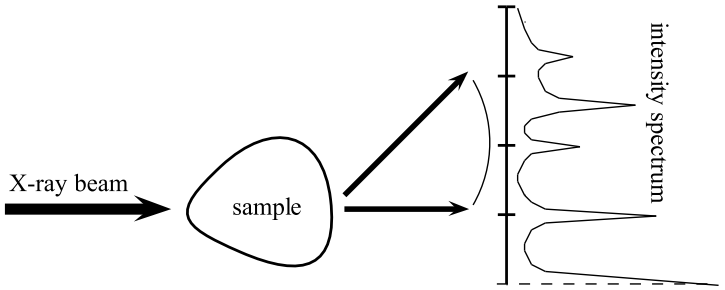

Figure 1.1: Schematic setup of an X-ray scattering experiment

In a computer simulation of a scattering experiment, a spectrum is computed for (a model of) a physical sample. Since the sample's structure must be known in order to compute the spectrum, one may argue that these simulations are of limited use. They are however useful, e.g. for

- comparing a computed spectrum to an measured one; this could either be a comparison of the spectrum of an unknown sample to a large database of computed spectra, or a comparison to a spectrum of a synthesized sample that is believed to have the same structure as the sample to be analyzed.
- aggregated information about a sample's structure, that may not be easily visible from the (modeled) sample. An example of this is the fractal dimension (which may be computed from a structure factor spectrum, a type of scattering experiment), that gives information about the relation between the mass and size of an object.

1.3 Problem description

The goals of this master's project are to:

- design an object-oriented model for computer experiments,
- implement a parallel program to simulate (scattering) experiments,
- research the suitability of the *Tuple Space* model to this type of simulations.

The object-oriented model for computer experiments, named the *OOSA* model (*Object-Oriented Scattering Analysis*), will accommodate scattering experiments, but the design strives for a more general approach.

Because computer experiments generally are computation intensive, the *PROOSA* software (*Parallel Object-Oriented Scattering Analysis*) will implement this model to simulate scattering experiments in parallel.

As there are many parallel paradigms, there are many ways to design such a system. The approach taken here is to implement *PROOSA* on a network of distributed nodes, communicating through a *Tuple Space* (see chapter 4), which cooperate to simulate experiments in parallel. An alternative approach, based on a *Service Oriented Architecture*, is presented in [Koo06].

Chapter 2

Simulation of scattering experiments

How experiments will be modeled, and how this model is applied to scattering experiments, is the subject of this chapter.

2.1 Modeling experiments as functions

In a physical experiment, measurements are performed to determine the values of properties of the system that is being examined. This setup can be simulated on a computer by constructing a model of the system and defining how it responds to certain events.

In the *OOSA* model however, experiments will be modeled as functions: as there exists many relations between physical quantities, one can often be expressed as a function of the other(s). Therefore, an experiment in *OOSA* will be defined as *the determination of a result quantity as a function of a number of argument quantities*.

A quantity is defined as *a quantifiable or assignable property ascribed to a particular phenomenon, body, or substance*. Examples of quantities are the length of a wave, the mass of a particle or the intensity of radiation.

Example: the function f

$$f : A_0 \times A_1 \times \ldots \times A_{N-1} \longrightarrow R$$

defines a result quantity R as a function of argument quantities A_i ($0 \leq i < N$).

Argument quantities can be differentiated by their domains. The simplest ones are the constants, i.e. they assume only one value. Examples are the wavelength of the (monochromatic) X-ray radiation, or the sample that is being analyzed. These arguments will often be removed from the function definition, with their values used as constants in the computation(s).

Arguments without a specific value will be variables, that can take any value from a certain domain. The result quantity's value will be determined for all

possible values from these domains. In case an argument domain is discrete, the possible values can easily be enumerated. But as most quantities have continuous domains, and therefore could assume an infinite amount of values, they will approximated by sampling their domains by a finite amount of values.

Example (continued): to approximate function f, for each of its (variable) argument quantities A_i a number of values $a_{(i,j)}$ is selected that enumerate or sample the domain of A_i. For each combination of these values, the value of the result quantity is determined.

These sampled values are values of a physical quantity, which means they are expressed as the product of a number and a unit; the numerical value of a physical quantity depends on the unit it is expressed in.

The determination of a result value, as a function of the (sampled) argument values, is called performing an observation. Thus, an observation consists of a tuple of physical quantity values $(a_0, a_1, \ldots, a_{N-1}, r)$, where $r = f(a_0, a_1, \ldots, a_{N-1})$.

An observation(-tuple) now consists of two parts. The numerical values of the argument and result quantities in the tuple are together called a signal. The units are part of the signature, which describes the quantities and the units they are expressed in.

2.2 Scattering experiments

In PROOSA, Debye's formula will be used as the method to perform observations of X-ray diffraction spectra. It is defined as

$$I(q) = \sum_{j=1}^{N} \sum_{k=1}^{N} f_j(q) f_k(q) \frac{\sin(q \cdot r_{j,k})}{q \cdot r_{j,k}} \tag{2.1}$$

where

I is the observed intensity,

q is the wave number, related to the wavelength λ and scattering angle Θ by $q = \frac{4\pi}{\lambda} \sin\Theta$,

j, k denote atomic scatterers,

$f_i(q)$ is the form factor of a scatterer i, which defines its scattering behavior at a certain angle,

$r_{j,k}$ is the distance between (the centers of) scatterers j and k.

2.2.1 Calculation of Debye's formula

Debye's formula is compute intensive function; for interesting simulations the number of scatterers N will be large, the double summation over N leads to a N^2 amount of terms

$$T(j,k) = f_j(q) f_k(q) \frac{\sin(q \cdot r_{j,k})}{q \cdot r_{j,k}}$$

To obtain a more efficient calculation, a number of transformations and approximations are applied.

The first observation to make is that, as the distance between two scatterers is symmetric and multiplication of all constituent terms of T is commutative, $T(j,k)$ equals $T(k,j)$. Applying this to formula 2.1, it transforms into

$$I(q) = N + 2\sum_{j=1}^{N-1}\sum_{k=j+1}^{N} f_j(q)f_k(q)\frac{\sin(q \cdot r_{j,k})}{q \cdot r_{j,k}} \tag{2.2}$$

thus reducing the number of terms by a factor 2.

A second observation is that, when computing the intensities for various diffraction angles for a static sample, the scatterers' positions and therefore their distances remain the same; instead of summing over all pair distances for each observation, the distances can be computed once and stored in a set of bins. This is the computation of a radial distribution function.

If a bin is defined to contain the total amount of particle pairs with a scattering behaviours j and k at a certain distance d, the formula changes to

$$I(q) = N + 2\sum_{i=1}^{N_{bins}} M_i f_i(q)\frac{\sin(q \cdot r_i)}{q \cdot r_i} \tag{2.3}$$

where

M_i is the number of particle pairs in bin i,

$f_i(q)$ is the combined form factor for scatterer types j and k, i.e. $f_j(q) \cdot f_k(q)$,

r_i is the distance between each pair

This transformation reduces the number of terms from the number of unique pairs to the number of bins (i.e. the number of unique distances). For regular structures, e.g. crystal lattices, this decrease of terms to sum can be very large. Most important, however, is that the particle pair distance calculations can now be reused, not only between the observation computations for a single experiment, but also in different types of experiments based on particle pair distances.

Instead of storing each unique distance in a separate bin, bins will be extended to store the amount of particle pairs within a (small) range of distances. Not only can this reduce the amount of terms even further, it also has advantages in the actual computation (as the amount of and distances within the bins are known in advance).

Note that this introduces an approximation of the actual radial distribution function; the smaller the bin ranges, the better the approximation will be, but the larger the number of terms in the summation will be, making the computation less efficient. The bin size used in simulations will therefore be a parameter that is configurable by the user of the software.

2.3 The experiments in *PROOSA*

2.3.1 Radial distribution experiments

In statistical mechanics, the radial distribution function $g(r)$ determines the average density of particles at a coordinate r relative to any particle in the sample. In *PROOSA*, $g(r)$ will be defined as the number of particle pairs at certain distance r apart. In terms of equation 2.3, $g(r_i) = M_i$.

In *PROOSA*, using radial distribution experiments for the computation of scattering analysis experiments increases the efficiency of the computation: the inter particle pair distances have to be computed only once (instead of for each observation) and the number of terms in de summation decreases (as similar distances contribute to the same term).

Furthermore, the results of radial distributions can be used in the computation of other experiments based on particle pair distances. Separating part of an experiment's computation into a separate experiment, allows it to be computed only once and its results to be reused in multiple experiments that follow it.

For these reasons, radial distribution experiments have been defined in *PROOSA* as its own class of experiments.

SRD

In the SRD experiment (Single Radial Distribution), the sample is considered to be homogeneous. It computes a function srd with signature

$$\text{srd: length} \longrightarrow \text{amount of substance}$$

that is defined by

$$\text{srd}(d) = (\#p, q : p < q : \mathit{distance}(p, q) \approx d)$$

where p, q are particles, $<$ is an operator which defines an ordering on all particles in the sample, *distance* is a function that computes the (box-)distance between particle pair (p, q) and $\approx$ is a comparator that returns true for a certain (small) range of distances around d.

MRD

Suppose every particle belongs to a certain substance, and set S is the set of all these substances. The MRD experiment (Multi Radial Distribution) has function signature

$$\text{mrd: length} \times (S \times S) \longrightarrow \text{amount of substance}$$

and definition

$$\text{mrd}(d, (s, t)) = (\#p, q : p < q \wedge \mathit{subst}(p, q) = (s, t) : \mathit{distance}(p, q) \approx d)$$

where (s, t) is a pair of substances, and function *subst*() defines the substance pair of a particle pair.

2.3.2 Scattering experiments

Scattering experiments compute spectra, that originate from radiation being diffracted by the particles in a sample.

SFS

The SFS experiment computes a *Structure Factor Spectrum*. It has signature

$$\text{sfs: length}^{-1} \longrightarrow \text{intensity}$$

and its definition is similar to 2.1, with the exception that the formfactors have been omitted:

$$\text{sfs}(q) = \sum_{j=1}^{N} \sum_{k=1}^{N} \frac{\sin\left(q \cdot r_{j,k}\right)}{q \cdot r_{j,k}}$$

As the SFS experiment is defined over particle pairs, but doesn't take their substances into account, it can be computed from the results of an SRD experiment.

SAXS

In the SAXS experiment, or Small Angle X-ray Scattering, a spectrum is computed by using Debye's formula (equation 2.1). Its signature is

$$\text{saxs: length}^{-1} \longrightarrow \text{intensity}$$

As described in section 2.2.1, the computation of this type of experiment can be based on the results of a MRD experiment.

WAXS

Wide Angle X-ray Scattering is performed by the WAXS experiment. It has signature

$$\text{waxs: angle} \longrightarrow \text{intensity}$$

and is defined as

$$\text{waxs}(\theta) = \text{saxs}(\frac{4\pi}{\lambda} \sin(\theta))$$

As this experiment is defined over particle pairs and requires their substance types for the formfactor computation, it can be computed out of a MRD experiment's results.

From the definition of waxs(θ) it can easily be seen that SAXS and WAXS compute very similar functions. The main difference is in the range of the parameters: WAXS computes intensity spectra over a wider range of angles, which results in other characteristic distances to become visible in the spectrum. More detailed information on the theory behind scattering experiments and the analysis of the results can be found in [GF55].

Chapter 3

An object-oriented model for computer experiments

In this chapter, the object-oriented model that will be used for the simulation of scattering experiments will be presented. It is designed, however, to accommodate more general types of experiments as well. The modeling of observations and quantities is inspired by the work of Fowler [Fow97].

3.1 Observations

In an experiment, a number of observations is performed; each observation contains measurements of the values (both numerical values and units) for a number of quantities. If experiments are modeled by functions, i.e., one quantity is considered to be functionally dependent on other quantities, an observation consists of zero or more argument quantities and exactly one result quantity. For each of these quantities, the observation holds a value.

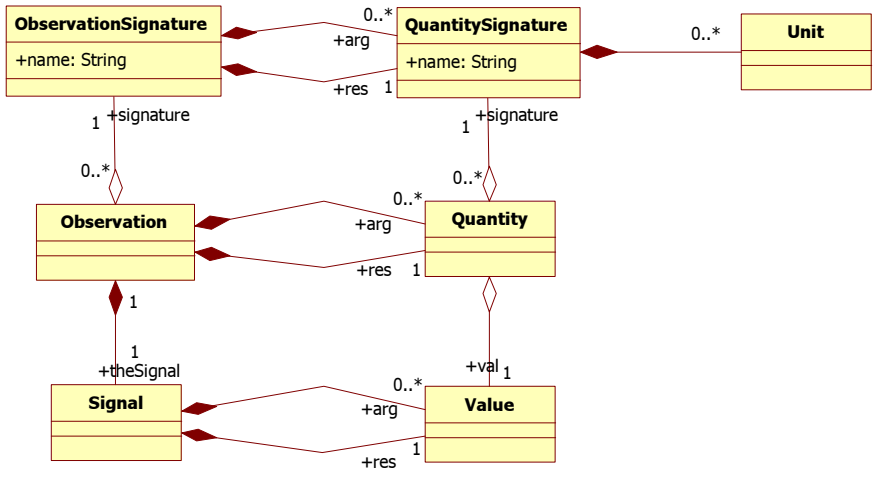

Figure 3.1: the relations between observation and quantity classes

In Figure 3.1 it is shown how observations are modeled in *OOSA*. The basic class is the Observation which consist of zero or more argument quantities and exactly one result quantity. Each Quantity object consists of two parts: the numeric Value of the quantity and its QuantitySignature, which describes the name and the units in which its values are expressed.

This separation of value and signature also exists at the Observation level: a Signal aggregates the numeric values for all arguments and the result, the ObservationSignature is composed of the individual quantity signatures (function signatures).

3.1.1 Signatures

In *OOSA*, signatures are used to describe quantities; for each Quantity, its corresponding QuantitySignature holds the name and the unit its value is expressed in. Each Unit has a name and a symbol; for example an area could be expressed in squared millimeters with symbol mm^2.

Currently, the only responsibility of a unit is to return a textual representation of its name and/or symbol. Therefore, units will constructed from a BaseUnit, which can be one of the basic SI units [Bur06] or a custom defined one. By means of decorators, these basic units can be extended with a prefix or a exponent; a decorator simply extends the textual representation returned by the unit it decorates. To illustrate this, the example unit from the previous paragraph would have to be constructed by decorating a Length instance of BaseUnit with a magnitude of -3 (signifying 10^{-3}, the milli prefix) and an exponent of 2.

Derived units are constructed by adding them separately to a QuantitySignature; for example, a quantity that represents a moment, would have two units, of type Force (Newton) and Length (meter), in its signature.

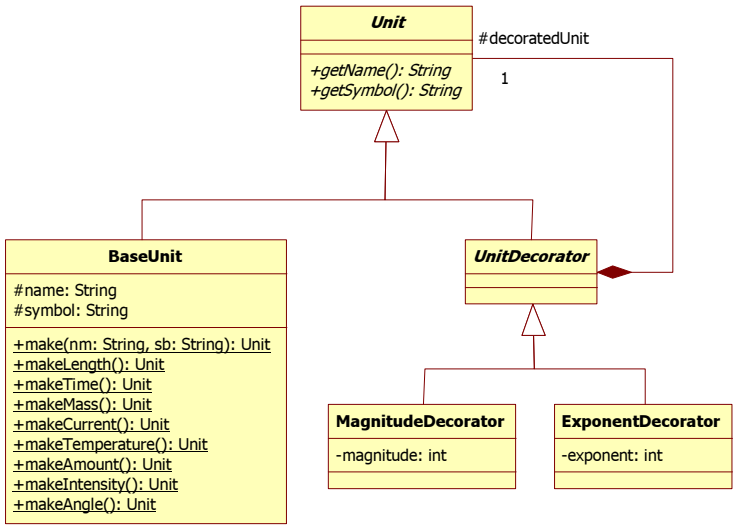

Figure 3.2: Units

In principle every quantity has its own signature. In *OOSA* however, where each observation is computed by the same function over the same argument

types, the signature of the quantities is the same for each observation, and can therefore be shared. This explains why the relations between Observation and Quantity and their corresponding signatures (see Figure 3.1) are modeled as aggregations instead of compositions.

3.1.2 Values

The numeric value of a Quantity is stored in a Value object. The responsibility of the Value class is to store values and —upon request— write them to output.

Quantities assume values from a certain domain (the type of the values); in *OOSA* the two main types are integers (for discrete variables) and reals (for continuous variables). Values from different types of domains are stored in objects of the subclasses of Value:

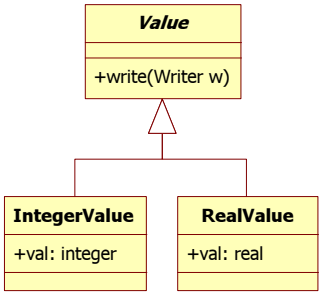

Figure 3.3: The abstract Value and its subclasses

The exact type of the primitive data types can be specified at compile-time. In general, a larger number of bits to store values in (e.g. using `double` instead of `float`) will increase the precision of the result, but at the cost of performance and increased memory use.

To abstract the code from the actual data type chosen to store the value in, the `val` attribute of the various Value subclasses has been given public visibility, to allow code to directly access the value without specifying a type; the exact type of the `val` field will be resolved by the compiler. If access to the value would only be possible by `get` and `set` methods, the return value and parameter would have to be expressed in this type; a change in precision of a value type would then also require modifications in code, instead of only changing the type of a single attribute.

To be able to determine this primitive type, the compiler will have to know what subclass of value is being referenced. *It will therefore be the responsibility of classes that operate on values, to have knowledge of the type of the corresponding quantity;* this possibly requires a (static) cast of an abstract Value to one of its concrete subclasses.

In OOSA*, this requirement comes down to experiments having to know the types of their argument and result quantities*, i.e., whether a certain quantity assumes values from a discrete or a continuous domain.

Note: in programming languages that feature a preprocessor facility, one can introduce a new type by means of a typedef, and have the preprocessor replace it by an actual type at compile-time. The approach above has been chosen here because *PROOSA* will be implemented in *Java*, which lacks a preprocessor.

3.2 Performing observations: Experiments

Performing observations is the responsibility of the Experiment class. Using a mathematical function which describes the relation between the argument and result quantities, an experiment computes the numeric value of the result for a tuple of argument values.

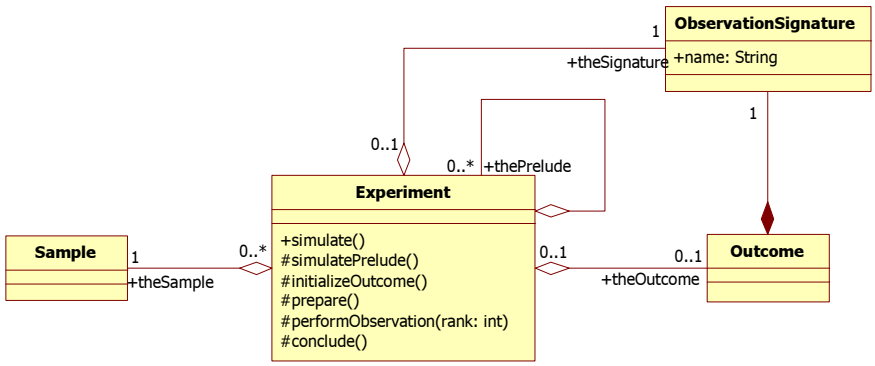

Figure 3.4: The Experiment class

Figure 3.4 shows the model for Experiment and its related classes. The Outcome of an experiment is a container class which holds the set of observations performed by the experiment. Details of how observations are stored in outcomes can be found in section 3.3.

All experiments in *OOSA* operate on samples, i.e. they compute functions defined over a (physical) sample. In this respect the Sample can be seen as an argument of the function, albeit an argument with only one (i.e. constant) value. Due to its special nature, Sample will be modelled separately; its structure is described in section 3.4.

As some experiment computations can be based on the results of other experiments (e.g. computing an intensity spectrum from a radial distribution), a prelude relation is defined between experiments. An experiment e will reference the outcomes of the experiments in its prelude, therefore it requires those outcomes to be known; all prelude experiments should have been simulated before e can start performing observations.

3.2.1 Experiment simulation

To perform the set of observations for an experiment e, a user (or an experiment which has e in its prelude) invokes e's `simulate`-method. This leads to a number of steps which are shown in Figure 3.5.

As the outcomes of the prelude experiments are required for the current computation, the first step is to ensure that all of them have been computed. To establish this, `simulatePrelude()` queries all of its prelude experiments whether their outcomes have been computed; if not, their `simulate` method will be invoked.

In the second phase, an experiment initializes its outcome. The method

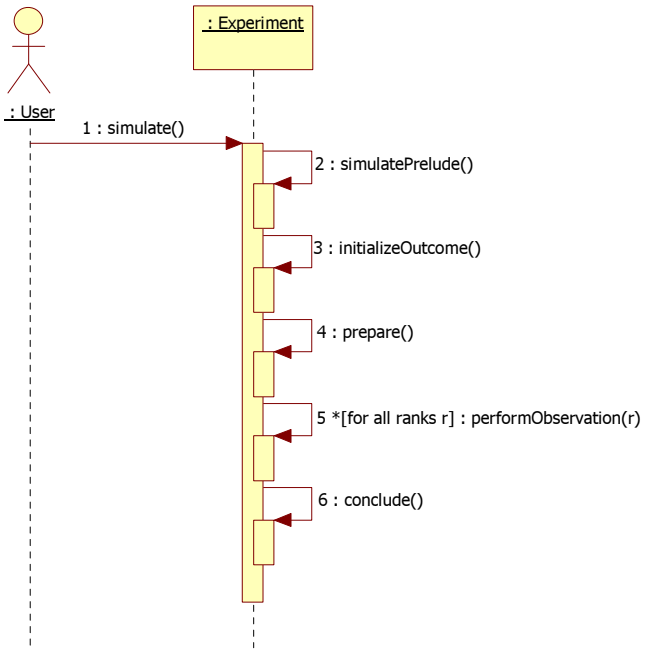

Figure 3.5: Simulation of an experiment

`initializeOutcome()` allocates memory for storing the observations and initializes the results to a suitable value.

Sometimes additional data structures can be used to obtain a more efficient computation: the classical tradeoff between space and time. This could be the computation of expressions that are shared between multiple observations, an example is a table of values that replaces a compute-intensive function. These structures are created in the `prepare()` method. The motivation for introducing these structures is the same as for introducing most prelude experiments. The latter option is preferred when its computed data structure is general enough to be reused by other experiments simulations, which —as experiment outcomes will be persistent— don't necessarily have to be simulated in the same run.

Note that, as `initializeOutcome()` and `prepare()` operate on separate data structures, they can be executed in parallel; the sequential ordering in Figure 3.5 is only one possible execution order and could also have been reversed.

The main step is the computation of all observations. If all observations are uniquely identified by a rank (more details on ranks can be found in section 3.3), `performObservation(r)` will be invoked for each rank. As shown in Figure 3.6, this results in the computation of the result value for the observation with rank `r` and its storage in the outcome.

The final step in a simulation is the invocation of `conclude()`, which performs post-simulation cleanup or finalizations.

3.2.2 Concrete experiments

The abstract Experiment class defines the basic structure for an experiment (simulation). To implement a concrete experiment, i.e. an experiment that

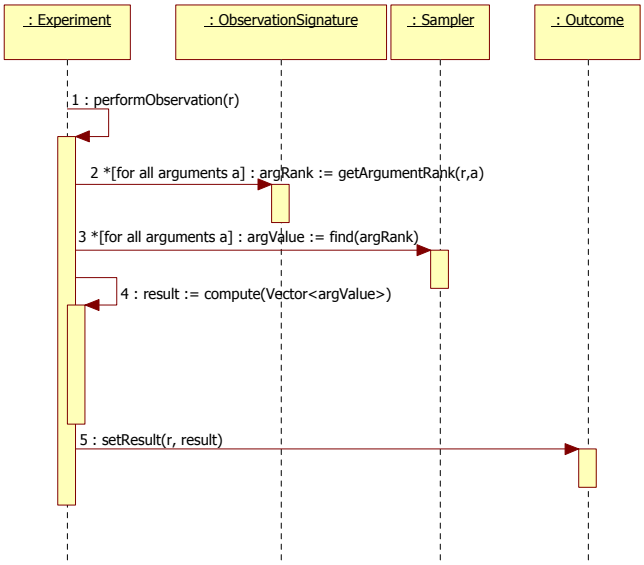

Figure 3.6: Performing an observation

computes a certain function, one has to create a subclass of Experiment and override some of its methods.

In *PROOSA*, five experiments have been defined (for an overview of the function each computes, see section 2.3). As can be seen in Figure 3.7, a separate subclass of Experiment has been created for each of these five experiments.

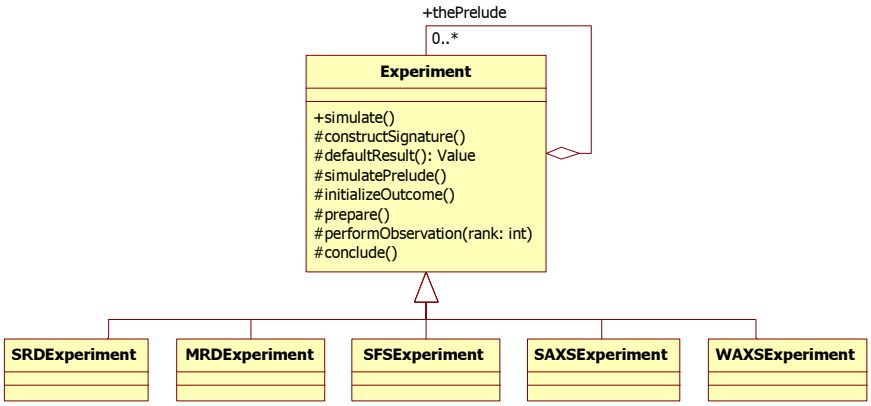

Figure 3.7: The abstract experiment and its concrete subclasses

The abstract methods of Experiment that will have to be overridden in subclasses are

`performObservation` which defines the computed function, i.e. how the value of the result quantity can be computed from the values of the argument quantities;

`constructSignature`, a factory method that constructs the (observation-)signature

for the function computed in the experiment;

`defaultResult`, which returns the initial Value for the result quantities.

3.2.3 Incorporating variations

Different functions will be computed in separate experiment classes; but even a single function can often be computed in a number of ways (this could be the result of the application of a certain approximation technique, in order to obtain higher performance).

When these variations in the computation are incorporated by the creation of Experiment-specializations for each variation, this can result in an explosion of subclasses, as the variations often are orthogonal to each other.

A better approach is to isolate the differing behavior in a separate class, as is done in the *Strategy* design pattern from [GHJV95]. For each possible variation a separate Strategy interface is defined; note that these interfaces can be very different as the variations, isolated in these Strategies, are orthogonal. The subclasses of Strategy implement the various alternative algorithms for the corresponding variation.

This approach is used in *PROOSA* for the computation of the sinc()-function (defined as $\mathrm{sinc}(x) = \sin(x)/x$), which is used in the computation of scattering experiments. As sin() is a relatively costly operation, higher performance can possibly be obtained by replacing the computation of this function by lookup of the result in a precomputed table.

This is demonstrated in Figure 3.8: the computation of sinc() is performed by the abstract method `calculate(double x)` of class SincStrategy. During an experiment simulation, the implementation in a subclass of SincStrategy, which could either be SincFunction or SincTable, will be used.

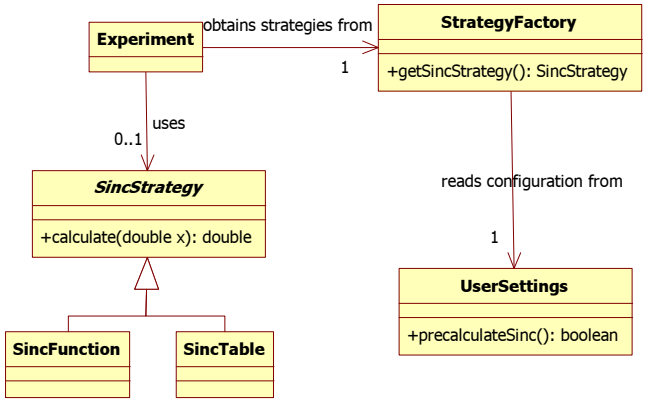

Figure 3.8: Experiments and strategies

Upon construction, an Experiment which makes use of a certain strategy interface has to be configured with a concrete object implementing it. By setting a configuration variable, the user controls which implementation should be used. The actual instantiation will not be performed by the experiment itself, but be delegated to a factory class, as this

- lowers the dependency of the experiment on the strategy: experiments only need to know how to use the (abstract) interface for the strategy, not which implementations are available and how they are controlled by user variables. Adding a new implementation only requires the modification of a single factory class.
- allows reuse of strategy objects: if a second experiment requests a strategy which has no internal state, the factory class can decide to return an earlier constructed object. For example, in the case of a SincTable this would save memory (only one table will be stored) and computation time (the time needed to compute the table).

Note that the strategy design pattern can also be used to unify the five experiments implemented in *PROOSA*:

- the difference between the two radial distribution experiments is whether or not the type of substance plays a role in selecting the bin to store the distance in. This different behaviour could be isolated in a strategy that performs the bin selection.
- the SFS and SAXS/WAXS experiments compute similar functions, except that in the former formfactors are absent. One approach to this is a strategy that either multiplies a term by a calculated formfactor, or by 1 (the identity element).
- SAXS and WAXS differ in their type of argument. A strategy could be employed which converts to scattering numbers. For SAXS this conversion would be the identity function, for WAXS the strategy converts angles to scattering numbers.

With three simple strategies like these, the number of Experiment implementations can be reduced to only two (one basic radial distribution experiment and one basic scattering analysis experiment). This approach, however, will not be followed in *PROOSA* because the strategy objects introduce an indirection which causes overhead, as the (virtual) methods of the strategy will frequently be invoked.

3.3 Storing the results of experiments: Outcomes

The set of observations performed by an experiment is stored in its Outcome object. The structure of the Outcome class is shown in Figure 3.9. Basically, an Outcome object is composed of a set of R observations; each of these is uniquely identified by an integer $rank$ (where $0 \leq rank < R$).

The most notable difference with the class diagram for observations (Figure 3.1), is that each argument Quantity is replaced by a SampledQuantity. This distinction is made as these quantities assume a predetermined set of values obtained by enumerating or sampling a domain. The Sampler object, which determines these values, is stored as part of the SampledQuantitySignaturee.

The number of observations R (the size of the outcome) is determined by the number of values taken from the domain of each argument (the size of each

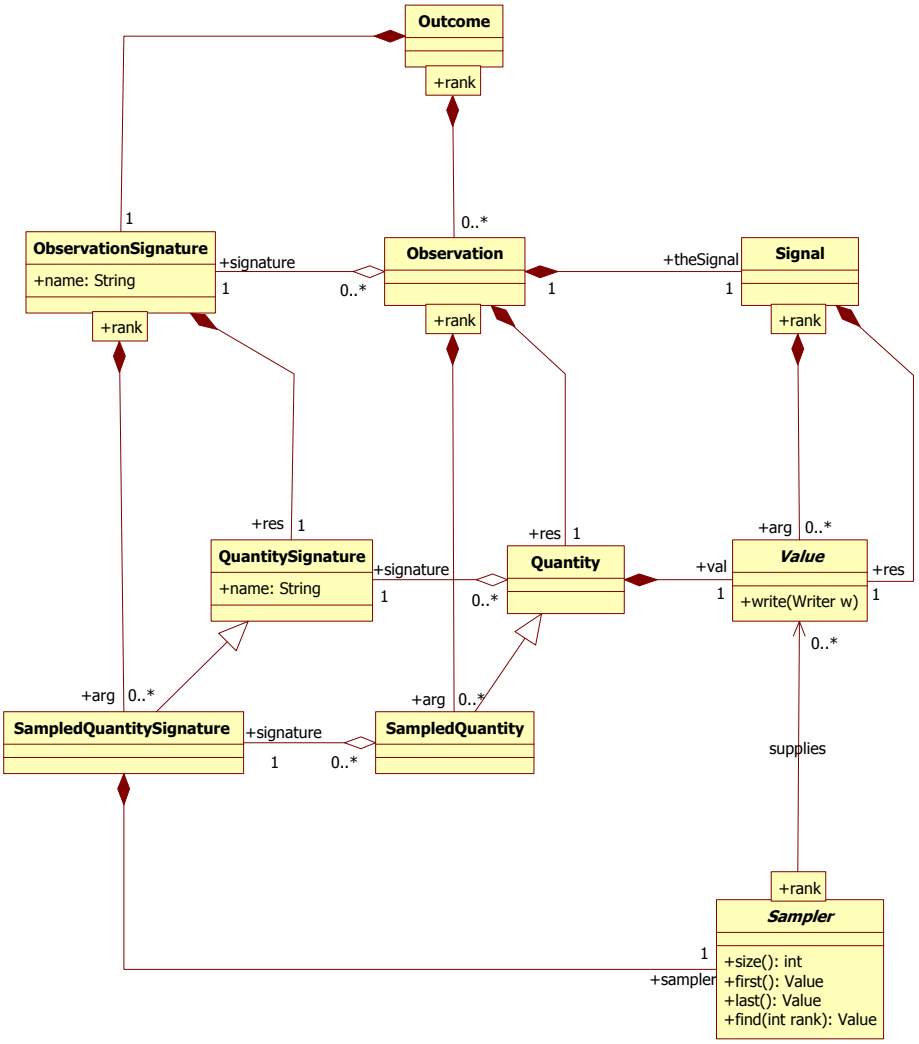

Figure 3.9: Outcome structure

SampledQuantity): the value of the result quantity will be computed for each possible combination of argument values. If $|A_i|$ is defined as the size of the i^{th} argument, and the number of argument quantities is N (which is the arity of the observation), then $R = \prod_{i=0}^{N-1} |A_i|$.

Similarly, the rank of an observation is determined by the ranks of the values of its SampledQuantities. Each of these values has an integer argument rank, which determines the values' position in the list of values returned by the corresponding Sampler. The functions, which perform the conversion from argument ranks to observation ranks and vice versa, are encapsulated in the ObservationSignature; they are used by method `performObservation(r)` of class Experiment, in order to determine the values of the arguments for the observation with rank `r`.

3.3.1 Sampling

Values, which sample a certain domain, are provided by the Sampler class. The interface for this class is shown in Figure 3.10: The abstract Sampler defines methods to obtain the first, the i^{th} or the last Value (where $0 \leq i < sampler.size$) from the sequence of value samples.

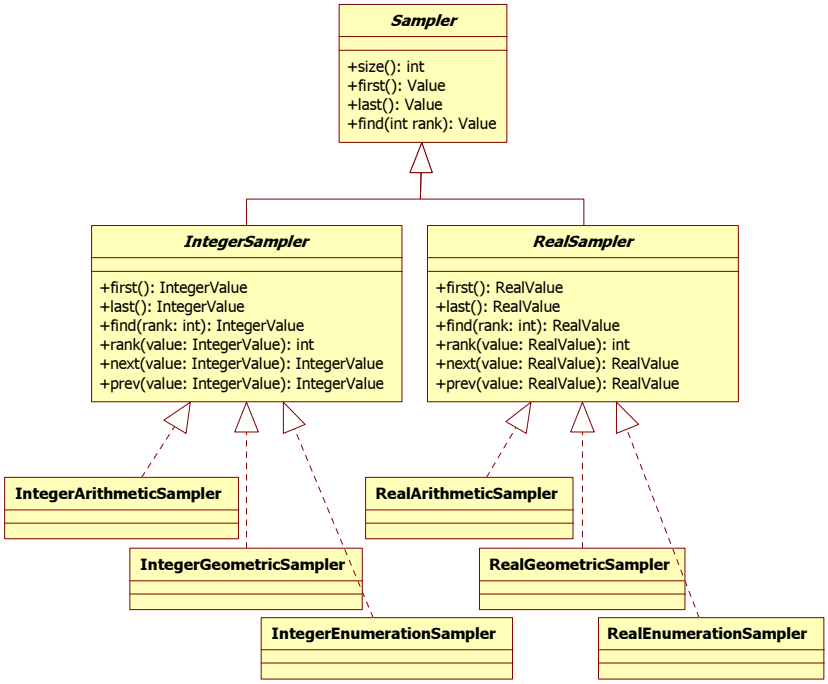

Figure 3.10: Sampler class hierarchy

For each concrete subtype of value, a Sampler subclass is defined which redefines the above methods (to return the more specific subtype), and introduces new methods which take parameters of the Value subtype: `prev` and `next` to obtain the sampled value that precedes or follows after the supplied parameter in rank, and a method to find the rank of a sampled value. In this way, samplers cannot be supplied with parameters of an unknown value subtype.

The *sampling scheme* defines which values from a certain domain are selected. In *OOSA*, three schemes have been defined and each is implemented in a separate subclass:

- EnumerationSampler objects simply enumerate the values in a list supplied to their constructors;
- ArithmeticSampler objects specify an arithmetic series. They are defined by a *start*-value, a *step*-size and a sampler *size* (i.e., the number of samples taken) and return samples $start + step \times rank$, where $0 \leq rank < size - 1$;
- GeometricSampler objects, similar to the above, define a a geometric series of values: $start \times step^{rank}$.

This approach does lead to a high amount of Sampler subclasses: for (almost) every combination of Value subtype and sampling scheme, a separate subclass is implemented. There are ways to avoid this explosion of subclasses:

- in programming languages that support templates and operator overloading, the sampler schemes can be implemented by operations on a template parameter t; upon compilation this type parameter will be replaced by the actual types used in *OOSA*.

- in languages that don't support the above, the approach can be mirrored by application of the *Bridge* design pattern [GHJV95]. The classes are separated into two smaller class hierarchies, one which defines the sampling schemes in terms of methods of the abstract Value, and another hierarchy which defines the value subclasses and implements the methods for each type.

In *PROOSA* however, every combination of value type and sampling scheme will be implemented in its own class. The first approach is not possible in *Java* (as it does not allow templates over primitive types, nor operator overloading); the second can be implemented, but would impact performance (as every arithmetic operation would require a method invocation). Apart from this, the number of subclasses is fairly limited and the hierarchy is not expected to grow much in the future.

3.3.2 Representations in memory

In the class diagrams for outcomes, Figure 3.9, it can be seen that an observation contains redundant information:

- the Values of an observation are referenced by both (Sampled-)Quantity and Signal. The Quantity doesn't have to be stored, as it can always be reconstructed from the numeric Value and (Sampled-)QuantitySignature (stored in the Signal and the ObservationSignature);

- the argument values are stored in the signal, but can also be obtained from the Sampler objects.

Therefore, three subclasses of Outcome will be implemented, which offer different tradeoffs between speed and memory usage:

- FullObservationsOutcome, which stores all information and thus contains redundancy;

- SignalsOnlyOutcome, which stores the Signal part only. In case one of its observations (or quantities) is accessed, it will be constructed upon this request;

- ResultsOnlyOutcome, which only stores the result values and thus has a minimal memory footprint. Signals or observations will only be constructed when externally accessed.

3.4 Modeling physical samples

The classes presented this far constitute the *OOSA* model, i.e. the simulation of experiments that are modeled as functions.

The experiments in *PROOSA* are based on functions that have a (physical) sample of particles as one of their (constant valued) arguments. For example, the computation of a scattering experiment is based on the inter-particle distances and the scattering behavior of those particles. How Samples are modeled is shown in figure 3.11.

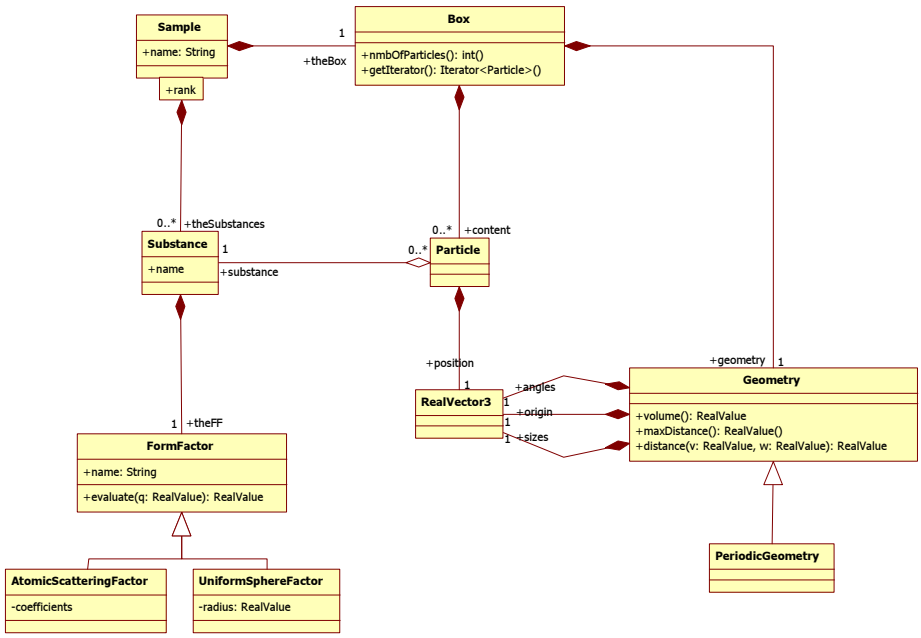

Figure 3.11: Sample and its constituent classes

The class Sample is the main class in the model. It consists of a Box, which defines the space and the particles contained in it. This definition of the space is encapsulated in a Geometry object; its responsibilities are the computation of distances within the simulation box. For simulations with periodic boundary conditions, a subclass PeriodicGeometry exists which overrides the distance calculation.

The Particles contained in the Box have various properties. Properties that are unique to a certain particle (e.g. its position) are stored as an attribute of the Particle class. Properties shared between a number of particles are stored as attributes in their common Substance; in the current model an example is such a property is the Formfactor, which defines the scattering behaviour for a certain Particle.

3.4.1 External representation

The Outcome and Sample classes in *OOSA* require their instances to support some form of persistency; this way their objects can be reinstated later, allowing them to be referenced by (future) experiment simulations.

In the current version of *OOSA*, these objects will be stored in external files, whose exact format is defined by a formal grammar in Backus-Naur Form (BNF). To reinstate such an object, a separate scanner and parser class will read such a file and reconstruct the object described by it. To illustrate this, appendix A contains the grammar that is used to represent a Sample object —one of the more complex structures in the *OOSA* model— in an external file. The SampleParser class is a recursive descent parser that recognizes this grammar and can be used to reconstruct a Sample from a valid file.

Chapter 4

Introduction to Tuple Spaces

The parallel computation version of *OOSA*, which will be introduced in the following chapters, will make use of a *Tuple Space* for the communication between the cooperating processes. This chapter will introduce the concept of tuple spaces, followed by a description of a few implementations of this communication model.

4.1 The Tuple Space model

The concept of tuple spaces (or shared dataspaces), was first introduced in the language *Linda* [Gel85]. In this model, asynchronous distributed processes communicate by storing and removing tuples from a common tuple space; this is also called *generative communication*, as the generated tuples independently exist as entities until some process decides to remove them.

4.1.1 Tuples and Tuple Spaces

A Tuple is, similar to a mathematical tuple, a collection of data values. For example, a tuple with type

$String \times Integer$

could have an instance

$(\text{"}ParticleCount\text{"}, 2^{10})$.

The tuple space is an abstract data type, distributed over a set of nodes P, where processes (residing on one of the nodes from P) can store persistant tuples in, or read tuples from by an associative matching mechanism.

4.1.2 Tuple Space operations

There are three basic operations that a process can perform on a tuple space (TS):

- put(t) to add a tuple t to TS,
- read(t) to read the values of a certain tuple from TS, and
- take(t) to remove a certain tuple from TS.

The atomicity of these operations is guaranteed by the TS implementation.

Put(t)

This operation adds the tuple supplied as parameter to the tuple space, enabling it to be read and/or taken by other processes. Note that multiple tuples of the same type and with the same values can exist in a single tuple space.

Read(t)

To select a certain tuple, the read operation is supplied with a *template*; if a tuple that matches the template can be found in TS, a copy of its values is returned; should multiple tuples match the template, one of them is selected arbitrarily. When no tuple can be found, this method will block until a matching tuple is available.

Take(t)

Similar to the Read(t)-operation, this operation removes one matching tuple from TS and returns it to the process that invoked the operation. Note that this operation can also result in blocking when no matching tuple is available.

Templates and matching

A *template* is a tuple with zero or more of its values unspecified. For a tuple of type $String \times Integer$, possible templates are (here $*$ is used a wildcard symbol):

$(*, *)$
$(\text{"}ParticleCount\text{"}, *)$
$(*, 2^{10})$
$(\text{"}a\text{"}, 0)$

A template t is defined *to match* a tuple v when all of t's specified values equal the corresponding values of v. For example, any of the templates $(\text{"}ParticleCount\text{"}, 2^{10})$, $(\text{"}ParticleCount\text{"}, *)$, $(*, 2^{10})$ and $(*, *)$ match the tuple from the example in 4.1.1.

4.1.3 Properties of Tuple Spaces

Tuple spaces have some properties that distinguish them from other communication models. The most notable are

- spatial decoupling: when two process communicate by putting and taking a tuple from TS, they have no information about each others identities.

- temporal decoupling: when a tuple is put in TS, it persistently stays in there until it is removed by some process. The process that puts a tuple in TS, may even have ceased to exist before the tuple is taken by some other process.

These properties lead to loosely coupled systems. Because they are spatially decoupled, processes have no information who read their produced tuples, or who produced the tuples that they read. This allows for scalability: in a producer/consumer situation, the producer needs no information which consumers read its tuples. Furthermore, more consumers can easily be added without any modification to the producer.

Temporal decoupling results in asynchronous communication: sender and receiver don't need to be active at the same time, which avoids waiting times when one process needs to wait for another.

Another property of the tuple space model is that communication is orthogonal to the programming language in which the individual processes are defined. The above three simple operations on a TS —which sometimes is referred to as a *coordination language*— can be added to any programming language to turn it into a parallel programming language.

4.2 Tuple Space implementations

There are many design decisions taken in the implementation of a tuple space system. The most important choice is how to store the tuples.

4.2.1 JavaSpaces (JS)

JavaSpaces, an implementation of the tuple space model for the *Java* language, takes a simple approach to implementing a TS: it is a non-distributed design where one process m manages the storage and retrieval of tuples, other processes communicate through a local proxy that forward the performed operations to m. It is a relatively easy option to implement a TS, as it avoids many synchronization and consistency problems, but this design introduces a bottleneck as all nodes depend on the single process m for their communication (both in performance and robustness).

Tuples in JS

Tuples in JS are ordinary objects, with the requirement that they descend from a class named Entry. All tuple values have to be defined as attributes with public visibility (this is required for the matching mechanism to be able to read their values) and have to be of an object-type (to enable the value *null* to function as a wildcard). For example the tuple type, mentioned in section 4.1.1, is implemented in JS as:

```
public class NamedIntegerTuple extends Entry {
  public String name;
  public Integer value;
```

```
  public NamedIntegerTuple();
}
```

JS operations

In addition to the three operations that were described in section 4.1.2, *JavaSpaces* also supplies non-blocking variants of the two operations that retrieve tuples and a method to register for notification when a certain tuple gets written in JS:

- `readIfExists(Entry e)`, which returns a copy from a single tuple that matches template e. Should no such tuple exist, this operation does not block, but returns control to the process that performed the operation;
- `takeIfExist(Entry e)`, which is the same as the above operation, except that it also removes the matched tuple from TS;
- `notify(Entry e, RemoteEventListener listener)`, which notifies the listener object when entries are written that match the template e.

Tuple matching in JS

A template e in JS is a tuple object where zero or more variables have unspecified values (for this, null is used as a wildcard value). When supplied to one of the tuple-returning operations, a tuple t is returned that

- is of the same type as e, i.e. a (sub-)class of e, and
- has the same values for e's specified variables.

When the subclass of the returned tuple t is unknown to the process that reads or takes it, the class file (which defines the class) is requested from a so-called *class server*. This allows for a very general implementation of consumers for a certain type of tuple: as tuple are objects, the code that is used by consumers to process it, can be defined in the class itself.

Communication costs

Performing tuple space operations can be time consuming. When a process p performs a tuple space operation, the tuple parameter has to be sent from p's *JavaSpaces*-proxy to the *JavaSpaces*-kernel managed by some process q. To transfer objects over a network from p to q, they first have to *serialized*, i.e., converted from an object to a binary stream of data, followed by *deserialization* in process q. Next, for operations that retrieve tuples, q will need time to search the stored tuples for one matching the template; if no such tuple is available and the operation is blocking, process p will be idle waiting for a tuple to become available. Finally, matching tuples will have to returned from q to p, which again requires time for (de-)serialization and transfer of data.

Therefore, the costs of performing tuple space operations consist of the costs for serialization and deserialization, data transfer between nodes and, depending on the type of operation, search time for finding matching tuples (which can include idle time).

4.2.2 GSpace (GS)

Another implementation of the TS model is *GSpace* [RCvS04, Rus06]; this system is truly distributed over a set of nodes, where each node has a local *slice* of the tuple space in which tuples can be stored. Not only does this remove the bottleneck as described in 4.2.1, it also allows for different strategies for the division of tuples over the local slices. GSpace will be used as the TS implementation in *PROOSA*.

Policies: customizable distribution

Instead of implementing a single strategy for the distribution of tuples over the local slices of the nodes (that communicate through the same TS), GSpace recognizes the fact that the best performing strategy is dependent on the access patterns for a certain type of tuple. As the application programmer defines these patterns in his software design, he or she is also better equipped to specify the distribution strategy (in GSpace terminology: *distribution policy*), whose performance characteristics best match the tuple's usage pattern.

This policy can be specified per type of tuple (i.e., per subclass of a general Tuple class). The eight basic policies available in GSpace are:

Store locally New tuples are stored in the local slice only. Finding a matching tuple is first tried on the local slice, if none can be found the other nodes will be queried to search their slices for a tuple matching the template.

Pull caching Tuples are written in the local slice. Tuples read from other nodes are cached in the local slice. When a cached tuple is read again, the node that owns the original tuple is contacted to verify whether the local copy is still valid.

Push caching Copies of tuples read from other nodes are again cached locally for faster (repeated) reading. When a tuple is taken from GS, all nodes with cached copies are notified of the invalidation of their copies.

Full replication Each node's local slice contains all the tuples stored in GS. Read operations can be performed locally, take and write operations must be forwarded to all other nodes in order to maintain consistency.

Fixed replication Similar to full replication, all tuples are replicated to the local slices of a group of nodes G, a subset of all nodes that is allowed to grow to a fixed size. Nodes that are not in G forward their operations to one of the groups' members.

Static replication The group of nodes G, in which the tuples are replicated, is static and determined before the start of the application.

Producer replication The nodes that produce tuples form a group G, all tuples are replicated over the local slices of the nodes in G.

Consumer replication In this policy the group of nodes, which replicate all tuples in their local slices, is formed by nodes that read and take tuples.

GSpace operations

GSpace supports the three basic tuple space operations of `put`, `read` and `take`. Of these three, the `read` and `take` operations are blocking, which means they only terminate when a matching tuple is available; in this respect, the operation can be regarded as synchronous communication, as the receiver (the process performing a read or take) cannot continue before the data from a sender (the process putting a suitable tuple) is available.

These blocking operations can be very useful for synchronization (e.g., when waiting for exclusive access to a tuple that contains a shared variable, or for a tuple that implements a semaphore) which may be required for the correctness of a program. In all other cases however, the best performance can only be obtained when there is no unnecessary idle time while waiting for a blocked operation; the processes should be kept as busy as possible.

Another possible problem is that it can be difficult for a receiver to determine whether a sender will be sending data (in TS: to guarantee the availability of a matching tuple); it is unsafe to just perform another receive operation, as executing it could result in being blocked indefinitely. This can occur when, as in tuple spaces, communication isn't one-to-one and a message may already been received by another receiver.

Of course these issues are not unique here, but arise in other systems as well, and several solutions have been proposed.

The first issue, idle waiting, can be avoided by performing the blocking operation in a separate thread, allowing the program to continue with other actions in its main thread. In some systems, separate nonblocking communication primitives are available as these may be able to attain even higher performance than the threading solution. An example of this can be found in the MPI library [For94], where nonblocking `send start` and `receive start` calls initiate a communication action, leading to data transfer that is performed concurrently with the actions following the call; to complete the communication, these calls should later be followed by a `send complete` and `receive complete`, which verify that the communication action was completed.

The second issue, i.e., to assert whether is safe to execute a receive operation so that it will not get blocked indefinitely, is less common. In tuple spaces the problem can occur because of the spatial decoupling: a receiver generally has no information about the number of producers (and other receivers) for a certain type of tuple. With only blocking communication operations available, a protocol will be required to resolve this issue.

Example: consider a set of nodes W (workers) that together process the tuples from a set T (tasks). Each node tries to remove as many tuples of T as possible and once T is emptied, all nodes should terminate. In the case of blocking operations, a take-operation can only be performed when it is certain that there is still a matching tuple available, as otherwise the process performing the operation will get blocked indefinitely, preventing the node to terminate. The only way to certify availability of a tuple from T is to introduce a tuple $SizeT$ which holds the number of tuples remaining in set T. When a node wants to remove a tuple from T, it first has to obtain exclusive access to tuple $SizeT$, decrement its value, and write it back. This scheme introduces a bottleneck,

as for each take operation, nodes need to obtain the $SizeT$ tuple and update it before they can safely take another tuple from T. Furthermore, the system isn't robust as all nodes in W will get blocked when one of W's members fails to put $SizeT$ back in tuple space.

The protocol described in the example above, requires two additional space operations (a take followed by a put of $SizeT$, as the value needs to be updated) per element of T. An more efficient alternative would be to have a communication primitive that asserts whether a matching tuple is available. This is similar to the *probe* primitive as proposed in [Mar85], which asserts whether a communication channel is ready for input. However, in a situation where there are multiple receivers (as in the example above), this condition isn't stable as it can be falsified by a communication action from a different receiver.

A solution is to perform `probe(t)` and `take(t)` as one atomic action. This combined statement has the same semantics as the `takeIfExists(t)` that is available in *JavaSpaces*. With these nonblocking operations available, the workers in the example can simply perform `takeIfExists()` repeatedly, until it no longer returns a tuple. In that case, set T has been emptied.

In conclusion, the three operations provided by *GSpace* are general enough and can be used to implement any TS-based program, but it can be useful — for performance considerations and ease of programming— to have nonblocking read and take operations available as well. It is therefore decided to extend the GSpace API with implementations of these two operations.

To illustrate the work required for this extension, appendix B lists the implemention of the blocking and nonblocking `take`-operations for the *Store Locally* distribution policy.

Chapter 5

Distributed computation of experiment outcomes

To simulate experiments in parallel, the computations will have to be partitioned and distributed over a number of concurrent processes. This chapter describes the general approach to this distribution in the *OOSA* model, and how it is applied to the specific experiments.

5.1 Partitioning the set of observations

In *PROOSA*, the parallel computation version of *OOSA*, multiple processes cooperate to compute the outcome of an experiment simulation. Since observations are independent of each other (i.e. each result can be computed in isolation), a simple approach to distribute the computational work over the nodes is to partition the set of observations —which comprise the outcome— into P subsets (where P is the number of nodes) and have each node compute the results for the observations in one partition. A simple union of all sets of results forms the outcome for the experiment simulation.

This approach has some consequences:

- the size of the outcome $|O|$ (i.e. the number of observations) can be a limiting factor to the number of nodes P that can be efficiently used. In general, $|O|$ should be several orders of magnitude higher than P: if it is smaller some nodes will remain unused, if they are of the same order the (relative) differences in size of subsets can be large which results in imbalanced loads between nodes.

- for some experiments it can be inefficient to compute only a subset of observations. A good example of this are the radial distribution experiments, where all particle pair distances need to be evaluated to obtain the result of a single observation. For these types of experiments, incrementally computing small parts of observations that ultimately form a complete observation is necessary in order to obtain an efficient simulation.

- the reason for introducing parallelism is the large computational effort required to compute observations: performing an observation often involves

the computation of an operation over a (very) large set (e.g. in *OOSA* a physical sample or the outcome of a previous experiment simulation). Each node will need access to this complete set, leading to high communication costs.

In conclusion, a complete observation can be a too coarse-grained unit of parallelism. Having nodes compute partial observations will increase the number of observation computations, can circumvent the properties that make computing complete observations inefficient and can reduce data dependencies (to only a subset of the large set an experiment is defined over).

5.2 Partial observations

5.2.1 Subdividing

In order to reduce the grain size of observation computations, each observation is divided into multiple smaller ones that are independent of each other and depend on only a subset of the data. When the results of these smaller observations have been computed, they need be combined to obtain the result for the original observation. This approach can be applied —without impacting the *OOSA* model for observations— by adding arguments to observations.

Example: a quantity Q is defined as a function of another quantity A. Suppose, for each sampled value a from A, an observation $Q(a)$ is defined as

$$Q(a) = \bigotimes_{b \in B} f(a, b)$$

where $\otimes$ is an associative and commutative operator and B is some (large) set. By introducing an element of B as an argument to Q, the partial observation $Q(a, b)$ is defined as

$$Q(a, b) = f(a, b)$$

Accumulation over all values b of the introduced argument yields the result of the original observation $Q(a)$:

$$\bigotimes_{b \in B} Q(a, b) = Q(a)$$

This way the number of observations is multiplied by a factor $|B|$ which can be very large. The associativity and commutativity of the $\otimes$ operator can be used to extend the number of observations in a controlled way; by partitioning set B in N disjoint subsets B_n and adding subsets instead of individual values as argument to Q, the equations become:

$$Q(a, B_n) = \bigotimes_{b \in B_n} f(a, b)$$

$$\bigotimes_{0 \le n < N} Q(a, B_n) = \bigotimes_{0 \le n < N} \bigotimes_{b \in B_n} f(a, b) = \bigotimes_{b \in B} f(a, b) = Q(a)$$

The number of observations is now increased by a (controllable) factor N and each partial observation depends on only a subset of the possibly very large set B.

5.2.2 Accumulation

When all partial observations have been computed, they need to be accumulated to obtain the result of the original observation.

$$\bigotimes_{0 \le n < N} Q(a, B_n) = Q(a)$$

With set B partitioned into N subsets, operator $\otimes$ needs to be applied $N-1$ times to N observations $Q(a, B_n)$ which all reside on different nodes. This involves the transfer of $N-1$ partial results to obtain the final result for $Q(a)$.

5.3 Application to *OOSA*'s experiments

5.3.1 General approach

For a general experiment which defines the application of a commutative and associative operator $\bigotimes$ over a large set B (the data structure an experiment is defined on, e.g. a sample or the outcome of a previous experiment) and P nodes available to perform computations on, the basic approach in *PROOSA* will be to partition set B into P subsets and have each node compute the results of the observations that depend on its subset.

Note that *the partitioning of B can (and will) be performed dynamically*: each node n starts with an empty set B_n, repeatedly removes one or more elements from set B and processes these, continuing until set B is empty. *This approach will balance the workloads between the nodes*, which can reduce the total time required for a simulation.

Initialization: each node initializes an outcome (i.e. an ordered list of observations) to value e, the identity element of operator $\otimes$.

Repetition: each node is supplied with iterators for the experiment's arguments and an iterator for a subset of B. For each combination of argument values, it calculates the terms of the partial observation that are based on the subset.

For balancing purposes, it is possible split the argument ranges into more (and thus smaller) sets of iterators than there are nodes available; now each node processes multiple sets of iterators until all have been processed.

Finalization: following an accumulation strategy, partial observations are transferred to one or more nodes, in order to compute the original observations defined by the experiment.

5.3.2 SRD Experiments

For SRD experiments, the function to be computed is

$$\mathrm{srd}(d) = (\#p, q : p < q : \mathit{distance}(p, q) \approx d)$$

where p and q are particles from a sample S, operator $<$ defines an ordering on all particles in S, $distance()$ a function that computes the (box) distance between two particles and $\approx$ a comparator that returns true for a certain (small) range of distances around d.

Defining S as the set of unique particle pairs, $S = \{(p, q)|0 \leq p < q < N\}$, the function becomes

$$\text{srd}(d) = (\#s : s \in S : distance(s) \approx d)$$

Note that d values must be chosen such that $(\forall s :: (\exists d :: distance(s) \approx d))$, in order to obtain a good approximation of the radial distribution.

As is it very inefficient to perform this computation for individual distances d (this would involve evaluating all particle pairs for each observation), the computation will only be partitioned over set S and not over the set of distances d.

5.3.3 SFS Experiments

Based on the outcome of a SRD experiment, SFS experiments compute the function

$$S(q) = N + 2 \cdot \sum_{d} \text{srd}(d) \cdot \frac{\sin(q \cdot d)}{q \cdot d}$$

Both a division in the range of q-values as one over the SRD outcome is a possibility. Subdivisions of the SRD outcome decrease the data dependencies for a process, but increase the number of partial observations that need to be accumulated into a full observation; subdivisions of argument q have the opposite effect (faster accumulation but more processes need access to the same data). Measuring performance for several possible subdivisions will be used as a guide to select the best performing option.

Chapter 6

Extending the *OOSA* model for parallel computation

To support the concept of partial observations, some changes to the Experiment class are necessary that allow for more fine-grained control over the computation; this control will be exerted by a Simulator class that defines how the experiment is simulated.

6.1 The Experiment class

The public interface of class Experiment is extended with four methods; most of these methods are the same as in the *OOSA* version of the experiment class, except that they now have public visibility. This is required as they will be invoked by external classes.

```
public void initializeOutcome()
```

This method is responsible for the allocation and initialization of a new Outcome object, that is used to store computed results in.

```
public void prepare()
```

Any other preparations necessary to perform (partial) observation computations, e.g. the computation of a $sinc(x)$-table, are performed in this method.

```
public void performObservations(Iterator<Arg0Type> arg0Iterator,
Iterator<Arg1Type> arg1Iterator, ...)
```

The actual computations take place in this method, where the supplied iterators define which (partial) observations to compute.

As the number and types of the arguments differ between the various types of experiments, the signatures of the `performObservations`-methods are different in each experiment class. This approach has been chosen to enforce type correctness of the parameters. As demonstrated later in this chapter, most classes interacting with experiments do not have to know the specific signature of this method, as its invocations can be isolated in a limited amount of classes that already are dependent on the signature of the experiment.

```
public void conclude()
```

Invoking this method performs the final steps required to obtain the outcome, which could be as simple as multiplication and/or addition of a constant to each observation.

The `simulate` method, which was present in the *OOSA* version of the Experiment class, is now obsolete as its responsibility will be delegated to a separate class that defines how to simulate an experiment.

6.2 Experiment Simulators

The Experiment class, as defined above, defines what function to compute, but no longer how its simulation will proceed. By applying the *Strategy* design pattern [GHJV95], the old implementation of the `simulate()`-method —which performed sequential simulations— was moved to a separate class SequentialSimulator. This allows different types of simulation to be specified by their own classes, which share a common Simulator interface (see Figure 6.1).

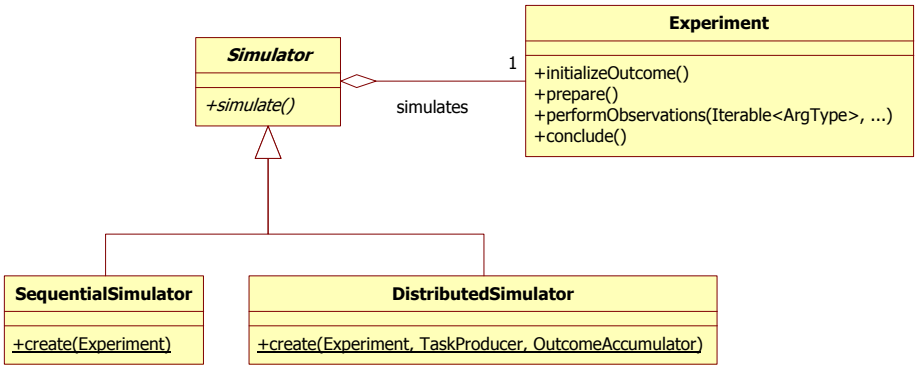

Figure 6.1: Simulator classes

Simulators are instantiated with an Experiment to be simulated; invoking method `simulate` will perform the simulation (i.e., compute all observations).

6.2.1 The SequentialSimulator class

To replace the sequential version *OOSA*, the class SequentialSimulator is introduced; its `simulate` method performs the same steps as the original `simulate` method as it was defined for class Experiment.

The only difference is that, instead of performing the observation for all ranks, it uses iterators —obtained from the Experiment itself— that sample values for all arguments.

6.2.2 The DistributedSimulator class

In the newly introduced DistributedSimulator class, the experiment's computation is distributed over a number of concurrent processes that perform the observations in parallel.

In pseudo-code, its `simulate` method performs the following steps

```
public void simulate() {
    constructTasks();
    distributeTasks();
    collectOutcome();
}
```

The above three steps will be delegated to separate classes. The reasons for doing so are:

- isolation of experiment-type dependent code: as task structure depends on the signature of the experiment (i.e. the amount and types of its arguments), confining the construction (and distribution) of tasks to separate classes avoids having to create a DistributedSimulator for each type of experiment.

- to allow multiple implementations, but avoid an explosion of subclasses: different strategies, both for distribution of the computation over the nodes as for the collection and accumulation of the outcome, are implemented in different subclasses. A single implementation of the DistributedSimulator class can now choose between multiple distribution and accumulation strategies by instantiating one of these subclasses.

These classes will be introduced in the following sections.

6.2.3 Further options for class Simulator

Isolating the simulation in a separate subclass allows to add new types of simulations later on. Examples are distributed simulations for a different communication layer between the nodes, or a Multi-threaded implementation on a single node to exploit the availability of multiple processor cores.

The Simulator class also allows hybrid simulations of multiple experiments (that belong together) by different Simulator implementations; it could well be possible that for certain experiments, the overhead for distribution and accumulation exceeds the performance gained by simulating in parallel. In this case is beneficial to perform these experiments by sequential simulation on a single node, and the others distributedly.

6.3 Distribution strategies

There are many ways to distribute an experiment computation over a set of processes; there a different strategies (e.g. which arguments to partition, whether or not to compute partial observations, which process(es) to do the accumulation) and for each of these strategies a number of parameters can be chosen (e.g. the size/amount of partitions). As it is very difficult to choose an optimal strategy in advance, these strategic choices are isolated to separate classes. Subclasses implement different strategies, parameter values can be selected by the user in a configuration file.

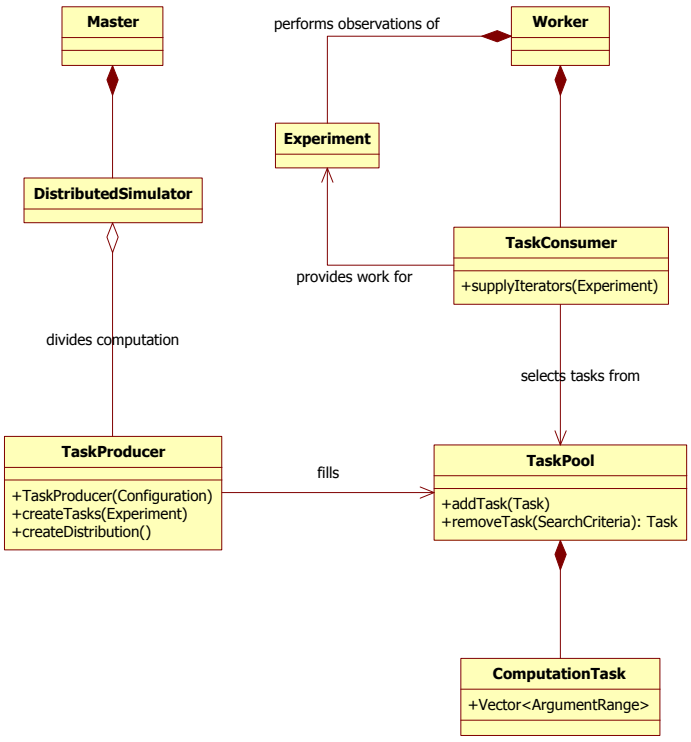

Figure 6.2: Distribution classes

6.3.1 Workload distribution

Unit of work

To represent the unit of work for a worker to do, the ComputationTask class is introduced: it defines the argument ranges to be used for a single invocation of the `performObservations` method of class Experiment. As sampler-objects will be supplying the iterators over the argument values, the ranges are specified by a starting rank and number of consecutive values to be sampled.

Note that, since the ranges are dependent on the signature of the experiment (i.e., amount and types of arguments), each concrete Experiment-subclass has a corresponding ComputationTask-subclass.

TaskProducer

The creation of tasks, i.e. the partitioning of the arguments into smaller ranges, is the responsibility of the TaskProducer class. Construction of the various subclasses of ComputationTask are implemented in separate TaskProducer-subclasses.

Invoking method `createTasks` will create a pool of (a certain type of) tasks, according to a certain distribution strategy. This strategy defines which argument(s) to partition; configuration parameters can be used to control the sizes of these partitions.

The experiment parameter is used to obtain the size of argument ranges from the experiment signature.

TaskConsumer

The allocation of tasks to the worker nodes will be done dynamically; after a worker process completes a task, it tries to obtain a new one, until all have been computed. This approach strives for a load-balanced distribution, where the faster nodes perform more tasks, such that every node completes computation around the same time.

The method `supplyIterators` searches for a available task and calls the experiment's `performObservations` method with argument ranges as specified by the task; this is repeated until the complete pool of tasks for the experiment is emptied.

Different strategies for searching for new tasks exists; a basic strategy could be to just randomly select a new task until all have been removed. More advanced schemes could make use of the fact that data, which has already been used for earlier computations, may be available faster because of caching mechanisms; strategies like these select tasks based on their data dependencies.

TaskPool

The task pool, which is represented as a class in Figure 6.2, will in *PROOSA* be implemented by the tuple space. The TaskProducer and TaskConsumer isolate the code for interaction with the tuple space from the Simulator and Experiment classes.

6.3.2 Outcome Accumulation

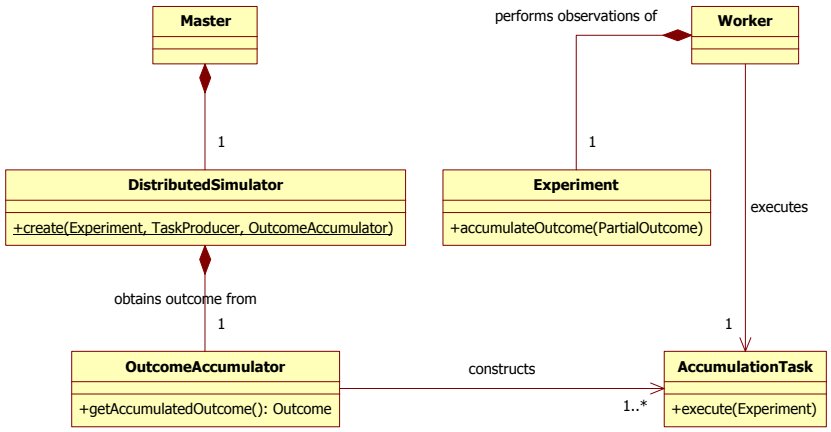

Figure 6.3: Accumulation classes

After all computations have been completed, each node has a part of the outcome, which have to be combined to form the outcome for the complete experiment. This accumulation step is comparable to performing a *reduce* operation of the MPI library [For94], where the sendbuffers on each node contain the partial observations to which operator $\otimes$ is applied.

This accumulation can also be accomplished in multiple ways. A simple strategy would be for each worker process to return its outcome to the master

that initiated the experiment, which then has to combine all results; more advanced methods use multiple process to do the accumulation in parallel (e.g., to have each process accumulate a subset of the observations, or a tree algorithm where each node accumulates the partial outcomes of its child nodes). Each of these strategies can be implemented in a specialization of the OutcomeAccumulator class.

In its single method `getAccumulatedOutcome`, for each worker process an AccumulationTask is created, which instructs a worker what to do with its computed partial outcome.

Note that the operator $\otimes$, which is applied to combine partial observations, is dependent on the specific type of experiment. By extending the interface of the Experiment class with a method to accumulate a partial outcome to its outcome object, class OutcomeAccumulator remains independent of the experiment types.

6.4 Other extensions to the Object-oriented model

6.4.1 Distributed data structures

The data structures used in the experiments of *PROOSA* tend to be very large in size; a sample can easily consist of millions of particles and outcomes are sampled for many argument values to obtain a good approximation.

As the computation of a partial observations requires only a subset of these data structures, these structures' interfaces are extended to allow processes to access subsets of their data. In the implementation of these new methods, only the requested subset of data will have to be transferred to the process accessing it.

To store these structures in tuple space, their data is partitioned into subsets of a certain fixed size; each of these subsets is stored in a separate tuple. The size of these partitions has an impact on the performance of *PROOSA*:

- the smaller the partitions, the higher the communication costs will be, as processes will have to execute more read/take operations to access a certain amount of data.
- the larger the partitions, the larger the task size will be, possibly resulting in more unbalanced loads.

Also, in the case that concurrent write access by multiple processes is required:

- the larger the partitions, the less parallel write access is possible; once a partition is taken out of tuple space by a process to update its value(s), no other processes can access any of the values stored in it.

For these reasons, the partition size will be a tunable parameter; increasing it leads to more efficient communication, but a decrease allows for more parallelism.

General approach

The *OOSA* version of the classes —that represent these large structures— provide access to their elements by supplying iterators. In the *PROOSA* implementation, the general approach is to extend those classes to implement the following interface:

```
public interface Partitioned<T> {
  public int numberOfPartitions();
  public Iterable<T> getPartition(int partitionIndex);
}
```

This interface defines that classes implementing it supply iterators over an indexed set of partitions. For structures that store their partitions in tuples, this interface hides the tuple space access from interacting classes.

In the basic implementation of these methods, the data contained in a partition will only have to be transferred when a process requests it. An alternative implementation is to cache (a number of the) read partitions in local memory, to provide faster repeated access to the data (at he expense of larger memory consumption). The advantage of this over caching at the distribution policy level, is that substructures will only have to be transferred and deserialized once.

Sample

As sample data is read-only, exclusive access to (a part of) it is not necessary. If each process used the complete sample, communication costs would be minimal by storing the sample in a single tuple (only one read access). However, as computation of partial observations requires processes to access only a subset of the data D_p, communication costs are minimal when only D_p is transferred. This also allows samples to be used that are too large to fit in the memory of a single process.

Figure 6.4 shows how the general approach is applied in *PROOSA*. As before, the set of particles in a sample are stored in the Box class (see section 3.4), which provides access to them by means of an iterator. This class can be exploited to hide the existence of a tuple space to classes depending on it. To that end, Box is turned into an abstract class, for which two implementations, SimpleBox and DistributedBox, are provided. SimpleBox contains the old Box code, DistributedBox changes this behavior by storing subsets of particles in tuples and by implementing Partioned<Particle> to provide access to its subsets.

Classes interacting with particles now only need to know the ParticlesContainer interface, not knowing whether the sample is distributed or not. This knowledge is only required by the classes that supply the ParticleContainers to them.

Sample construction is performed by the `SampleParser` class. With two implementations available (i.e. distributed or not), the SampleParser should not be dependent on the actual type of sample it is constructing. These classes are be decoupled by applying the *Abstract Factory* design pattern [GHJV95]: the sample and all its constituting classes are instantiated by a `SampleFactory` class, which hides the actual types of the classes to the parser.

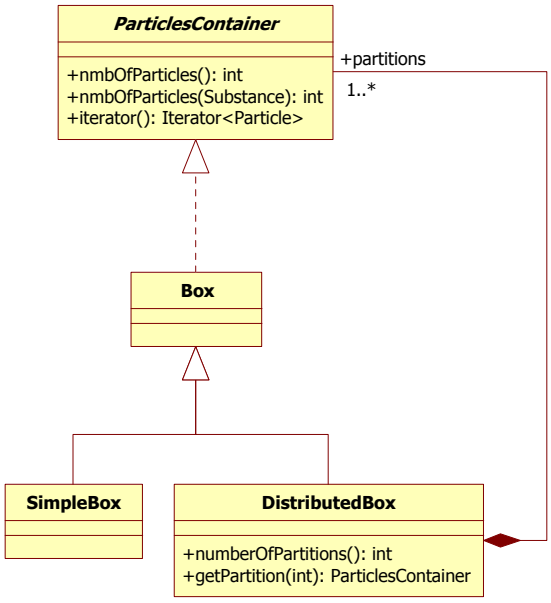

Figure 6.4: Distributed Sample access

Outcome

There are two types of outcomes in PROOSA:

- the (partial) outcome of the experiment that is currently being simulated (e.g. by a SequentialSimulator or a worker process that performs a subset of the partial observations);
- the outcome from a previous experiment simulation, that is used as a prelude to one or more experiments that follow it.

The first type of outcome reuses the *OOSA* implementation, as it is local to a single process. For the second type, partitioning is useful to lower communication costs, as only a subset of the prelude's observations may be needed to compute a (set of) partial observation(s).

A newly introduced DistributedOutcome class extends *OOSA's* (abstract) outcome by implementing the Partitioned<SampledValue> interface (where SampledValue, the type that is iterated over, will be introduced in the next section).

In the implementation of this method, each partition of sampled values will be stored in a separate tuple, and will be read from tuple space by a DistributedOutcome when a process requires one of its partitions.

Outcome construction When a process creates an outcome, it is always a non-partitioned one that is either used for computation of observations, or has been reconstructed from a previously stored outcome by a parser. To construct a distributed version, `DistributedOutcome` is equipped with a constructor which takes an Outcome object O as a parameter. The observations from O will then be partitioned and stored in separate tuples.

Representing partial outcomes In the accumulation phase of an experiment, partial outcomes are transferred from one process to another in order to combine them. Instead of transferring a complete outcome object, a class `PartialOutcome` is introduced, which only implements the `Iterable<SampledValue>` interface. Its implementation tries to keep the memory footprint as small as possible, in order to minimize communication costs (which was the reason for introducing a separate class for this type of outcome).

6.4.2 Iteration over ranges

For the computation of partial observations, iteration over a range of sampled values is required. A sampled value will be represented by the following interface

```
public interface SampledValue<V extends Value> {
  public int getRank();
  public V getValue();
```

Classes supporting iteration over sampled values, do so by implementing the standard *Java* Iterable-interface.

Iteration over samplers

The primary providers of sampled values are the Sampler-objects contained in the experiment's signature. Their interfaces are extended with the following method:

```
public class IterableSampler extends Sampler {
  public Iterable<SampledValue> getRange(int start, int numSamples);
```

A call to `getRange` returns an object which provides iterators over sampled values with ranks $start, start + 1, \ldots, start + numSamples - 1$.

Iteration over prelude outcomes

As described in a previous section, outcomes from one or more prelude experiments are stored in a distributed fashion. Their partitions implement the Iterable<SampledValue> interface, which provides iterators over the (consecutive) range of observations contained in a partition (see section 6.4.1). This common interface allows uniform access to prelude outcome partitions and ranges of a sampler by experiment instances.

This approach limits iteration over distributed outcomes to ranges contained in a single partition. Therefore, computation tasks will specify ranges of observations from prelude experiments by their partition index. This design has been chosen for efficiency, to ensure that all transferred data (when accessing a partition) is used by the process.

Iteration over Particle pairs

For the computation of radial distributions, iteration over particle pairs is required. As the particles in a Sample are partitioned (see section 6.4.1), two classes are introduced that return iterators over particles pairs contained in a ParticleContainer:

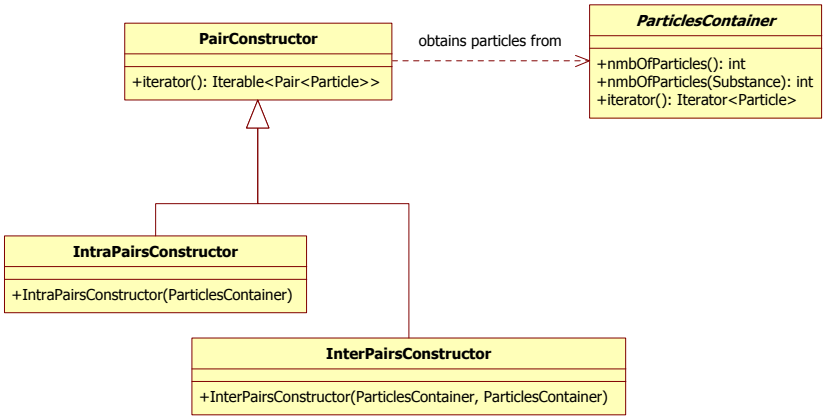

Figure 6.5: Particle pair iteration

The classes in Figure 6.5 return the unique pairs of particles contained in one or two ParticleContainer objects: for a partition size of N, class `IntraPairsConstructor` returns all $\frac{1}{2}(N^2-N)$ unique pairs in a single partition, and `InterPairsConstructor` returns all N^2 pairs from two partitions.

As for outcomes, the possible ranges for iteration over particles are limited to those particles contained in a single partition. Not only does this make it easier to specify ranges (and particles don't need globally unique identifiers), but it also is more efficient as iterators return exactly all particles in a single partition (which ensures that the transferred data is optimally used).

6.5 Overview

In this section it is demonstrated how the classes, introduced and extended in the previous sections, cooperate in order to calculate the outcomes of experiments.

6.5.1 Distributed simulation

In a distributed simulation, a master process and a group of worker processes cooperate to compute the outcome of an experiment: the master distributes the work over smaller tasks, which are consumed by the workers who perform the (partial) observations computations prescribed by the task.

Master process

The master process performs experiment simulations by construction and simulation of a DistributedSimulator.

The first step in a distributed simulation is the construction of tasks, which define how the computation is divided in smaller parts; this is the responsibility of the TaskProducer-class.

The next step is the distribution of tasks over the set of worker processes. For load balancing purposes, this allocation of tasks is performed dynamically by the workers themselves. To hide the used search strategy for finding tasks from the workers, this functionality is encapsulated in the TaskConsumer-class.

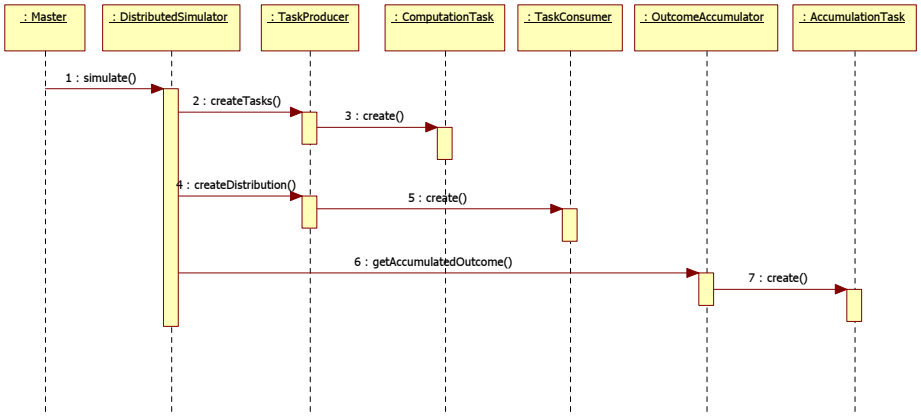

Figure 6.6: Distributed simulation of Experiments by a Master process

The final step is the accumulation of the partial outcomes (owned by each process) into the outcome for the experiment. The OutcomeAccumulator class instructs the workers how to perform the accumulation by providing each of them an AccumulationTask.

Note the absence of `conclude`, in general this method will be called by the process that performs the final accumulation step (which, depending on the accumulation strategy, can be the master itself or one of the worker processes).

Worker process

The actual observation computations are performed by worker processes.

Note: in this section, the worker process interacts with a number of objects that were created by a master process. How the worker obtains these objects will be detailed in the next chapter.

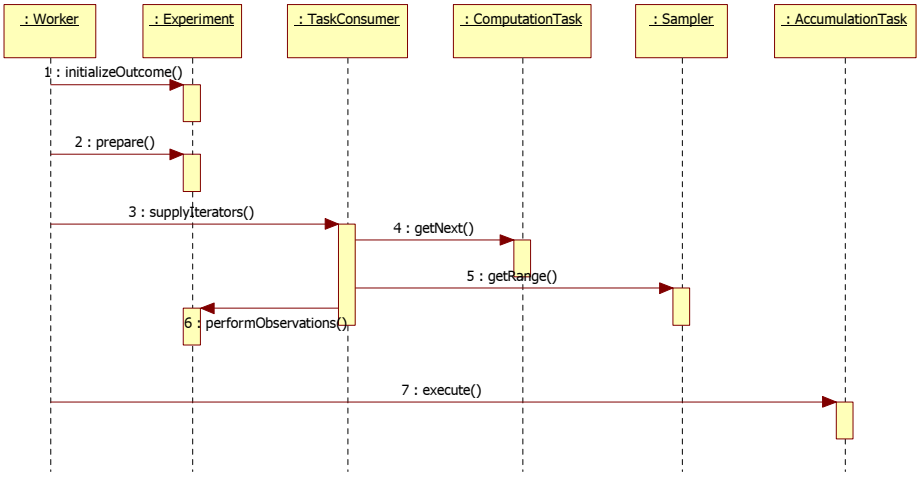

Figure 6.7: Distributed simulation of Experiments by a Worker process

When a worker wants to perform observations for an experiment, it first

performs the steps of initializing an outcome (for storing partial results) and performing any other preparations necessary for observation computations.

Next, it instructs its TaskConsumer to provide work to the experiment: the TaskConsumer will repeatedly search for a new task, obtain iterators for the ranges specified in that task and provide those iterators to the Experiment's `performObservations`-method.

Finally, the worker executes a single `AccumulationTask`, which instructs it what to do with its partial outcome.

6.5.2 Sequential simulation

For comparison, an overview of sequential simulation will also be given here.

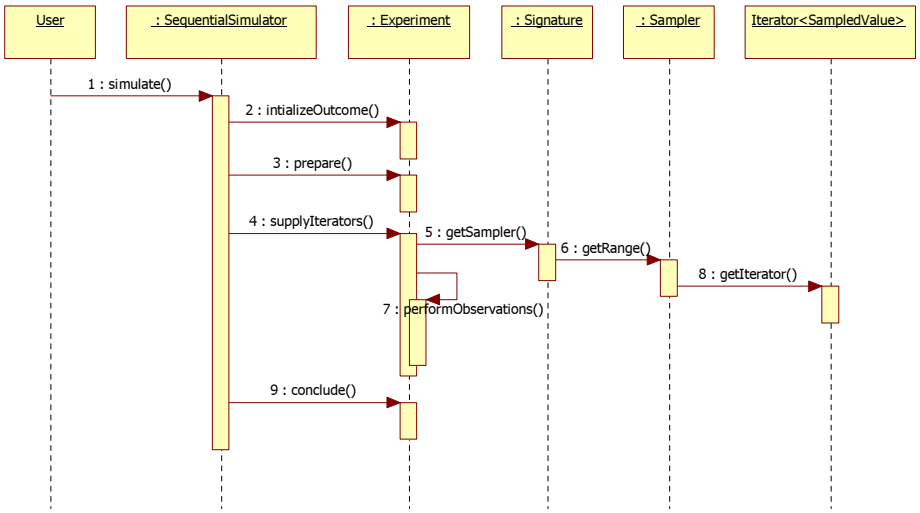

Figure 6.8: Sequential simulation of Experiments

Simulation is initiated by the user, instructing the SequentialSimulator to start the computation of the outcome. First, the experiment is initialized by constructing a new outcome to store results in, and secondly by performing other preparations needed to perform computations (e.g. initialization of a table that replaces the computation of a costly function).

The next phase is the computation of the observations. Instead of obtaining iterators for the argument variable(s) from a TaskConsumer, they are now supplied by the Experiment itself; the experiment obtains the iterators —for the complete range of argument values— from its Sampler objects and uses them as parameters in the invocation of its `PerformObservations` method.

In the final phase of the simulation, the experiment performs any operations defined in its `conclude` method.

Chapter 7

Tuple Space Implementation

7.1 System architecture

PROOSA is implemented on a network of nodes, that can communicate by reading and writing tuples from *GSpace*, a tuple space (TS) implementation. A network node can run any of two types of *PROOSA* processes:

- masters, who have one or more experiments to be simulated, and
- workers, who perform the computation of experiments.

7.1.1 Master nodes

To perform an experiment simulation, a user creates a configuration file which specifies one or more experiments to simulate and starts a master process on one of the nodes. For each of these experiments, this master constructs a DistributedSimulator and calls its `simulate`-method. After all experiments have been performed, the outcomes are stored and the process terminates.

7.1.2 Worker nodes

Worker processes offer the service to perform experiment computations for masters; running a worker process on a node makes it available for a master to use its service.

After being started, a worker will wait for an opportunity to join a new simulation; after having joined one, it computes tasks for each of the experiments specified for the simulation.

In the current implementation, a master expects the number of workers to remain constant for the duration of the simulation. Therefore, once a worker has joined a simulation, it has the obligation to stay in it until all the experiments are completed.

7.2 Communication protocol

Nodes in the network can only communicate by reading, taking or writing tuples to the TS. The communication protocol specifies the various types of tuples and how they are used in the communication between master(s) and workers.

7.2.1 Space Initialization

A TS that is to be used for *PROOSA* simulations, should be initialized with exactly one instance of the following two tuple types:

tuple name	**field**	**type**	**initial value**
UniqueID	nextAvailable	long	0
AvailableWorkers	amount	int	0

As there could be multiple simulations running on the same network, the tuples that belong to a certain simulation will be marked with an ID. To obtain an ID that is unique to this shared dataspace, processes can take the UniqueID tuple and get an unused ID from it. The AvailableWorkers tuple is used by masters to discover the number of available worker processes (see next section).

7.2.2 Starting a new simulation

At the beginning of a new simulation, a master first has to discover the number of available workers; it can then gather (a subset of) workers and use these to perform its computations.

Workers: announcing availability

When a new worker is started, it signals its availability to join a simulation by taking the AvailableWorkers-tuple out of the TS, incrementing its amount by one and writing it back.

If a node no longer is available to join a simulation, it has to access the AvailableWorkers tuple and decrease its amount; only if the amount can be decremented to a non-negative number, a worker is allowed to terminate, as an amount of 0 signifies that a master is already depending on this worker to join its simulation.

Master: discovering available nodes

When a new master is started, its first objective is to gather a set of worker nodes that join in to compute the outcome of its experiment(s). For this, it takes the AvailableWorkers-tuple from TS, selects an number of nodes $P (0 \leq P \leq AvailableWorkers.amount)$, decreases the tuple's amount by P and writes it back.

Having determined the number of worker nodes P, an experiment writes out exactly P Join-tuples. As these tuples will be consumed by the workers, only P workers will be able to obtain one and join this simulation.

tuple name	field	type	initial value
Join	simulationID	long	unspecified
	numberOfExperiments	int	unspecified

To obtain an unused ID for its simulationID, a master can query the UniqueID tuple.

Worker: joining a simulation

Available workers await the availability of a Join-tuple to be taken from TS, by executing a blocking `take` operation. As soon as one is available, the worker has joined the simulation with the specified ID.

7.2.3 Simulation of experiments

After having stored the Join-tuples, which will guarantee that P workers join the simulation, the master can start simulation of its experiments.

Master: starting a new experiment simulation

Based on the user-specified configuration, the master creates the experiment object that is to be simulated next and stores it in tuple space:

tuple name	field	type	initial value
ExperimentTuple	simulationID	long	unspecified
	experimentID	long	unspecified
	theExperiment	Experiment	unspecified

It then creates a DistributedSimulator for the experiment and invokes its `simulate` method.

Worker: initializing a new simulation

After having joined a simulation, the worker awaits the availability of an ExperimentTuple (with the specified simulationID) to be read from TS.

For the experiment contained in it, the worker invokes the `initializeOutcome` and `prepare` methods; this can be done in separate threads, as both methods operate on different data structures.

Master: creating a distribution of the computation

The first two steps in a distributed simulation is the construction and distribution of tasks over the workers (see section 6.2.2).

Construction of the task pool is performed by the TaskProducer of the DistributedSimulator. After invocation of its `createTasks` method, it will create —based on a distribution strategy— a number of Task tuples and writes these to TS:

tuple name	field	type	initial value
TaskTuple	experimentID	long	unspecified
	argumentRanges	unspecified (dependent of experiment signature)	

As tasks are dependent on the (signature of) the experiment, each Experiment has its own type of Task tuples.

Distribution of tasks takes place dynamically: with a pool of Tasks available, workers continuously search for Tasks to remove and compute, until none are left. This search strategy is encapsulated in the TaskConsumer class, which hides the type of tasks and search strategy being used for the workers. A master constructs P (possibly different!) TaskConsumerTuples and stores them in TS:

tuple name	**field**	**type**	**initial value**
TaskConsumerTuple	experimentID	long	unspecified
	theConsumer	TaskConsumer	unspecified

Important notice: in the current design, the TaskConsumers are written to TS only after all Tasks have been stored in it; this is done to ensure that, when a TaskConsumer cannot take any more Tasks from tuple space, the assumption that all tasks have been allocated to workers is valid.

Worker: performing computation tasks

After the experiment to be simulated has been initialized, the worker is ready for computation of tasks. These tasks are provided by a TaskConsumerTuple. First the worker takes one of these tuples with a matching experimentID from TS; next, for each task provided by its TaskConsumer, the experiment computes the (partial) observations for the argument range(s) specified in the task.

Master: accumulating the results

For a master, the final step in the simulation is to obtain the accumulated outcome. The strategy used to accumulate the partial outcomes is encapsulated in the OutcomeAccumulator class. To instruct the workers what to do with their computed outcomes, `getAccumulatedOutcome` writes P AccumulationTaskTuples to TS:

tuple name	**field**	**type**	**initial value**
AccumulationTaskTuple	experimentID	long	unspecified
	theTask	AccumulationTask	unspecified

Worker: outcome accumulation

When the task pool of computation tasks has been exhausted, workers take a single AccumulationTask from TS and execute it on their outcomes. This completes the participation of the worker to the current experiment, allowing it to dispose of the objects that where instantiated for it.

7.2.4 Termination of a simulation

Master

The above steps, are repeated for each of the experiments specified in the configuration file of the master. After all experiment outcomes have been computed,

the master writes the results to the filesystem and terminates.

Worker

When a worker has performed computation and accumulation tasks for the number of experiments that were specified in the Join tuple, it first disposes of all remaining objects used for this simulation. Then it either decides to terminate, or to be available for other simulations by incrementing the amount of the AvailableWorkers tuple again.

7.2.5 Summary

The tuple communication protocol, as described in the previous paragraphs, is visualized in the following state diagrams.

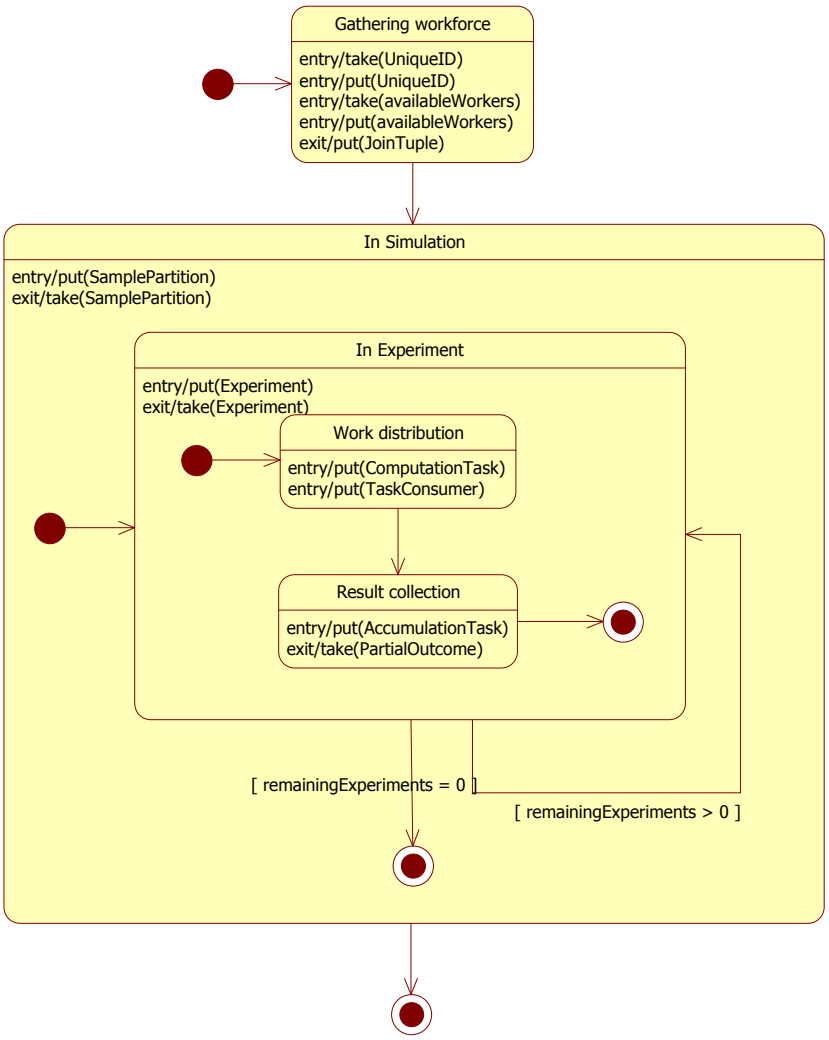

Figure 7.1: Tuple operations performed by Master

Figure 7.1 shows the communication protocol for a *PROOSA* master process, Figure 7.2 does the same for a single worker.

It is clearly to see that the processes visit similar states, where the tuples produced by the one are consumed by the other. The only exception are tuples that contain read-only data and are the same for all workers (currently, these are the experiment and sample partition tuples): the master will store a single copy of these tuples when entering a state and is responsible for removing it when exiting the state.

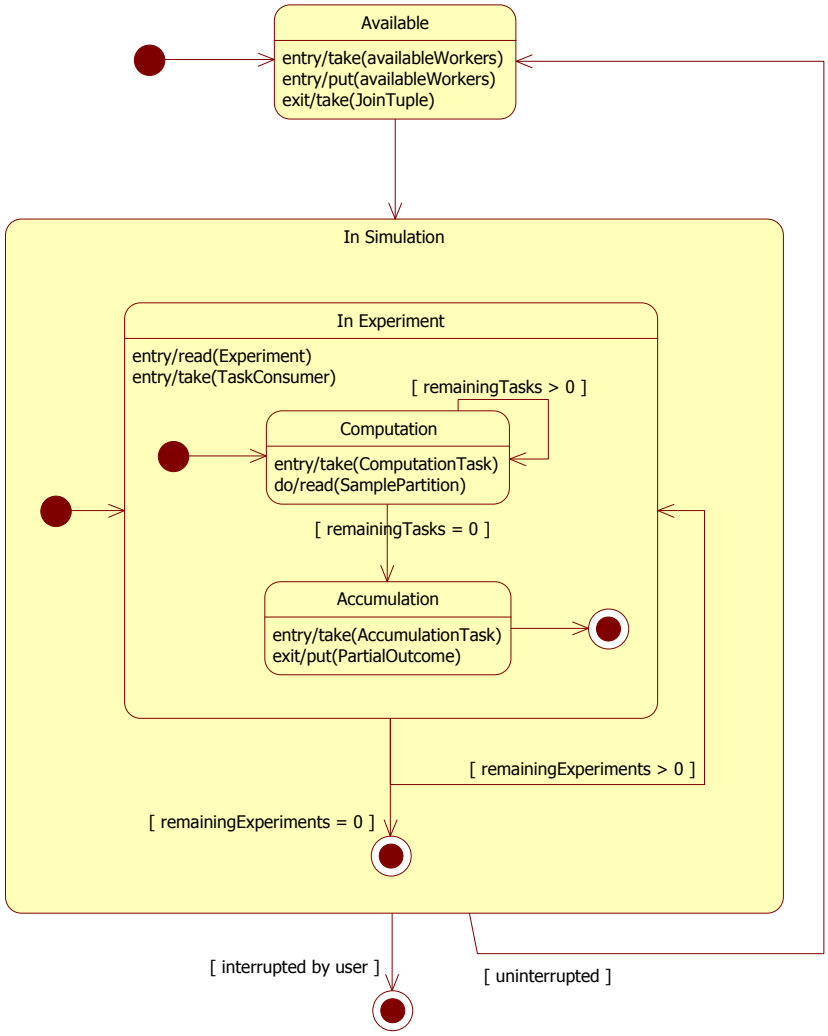

Figure 7.2: Tuple operations performed by Workers

7.3 Distribution policy selection

In *GSpace*, each type of tuple can be assigned its own distribution policy. By selecting a policy whose performance characteristics best match how an appli-

cation uses a particular type of tuple, communication costs can be reduced.

The default distribution policy is *Store Locally (SL)*, in which tuples are stored in the gspace kernel that is running locally to the process executing a `put` operation. The advantages are that tuples can be stored fast and require a minimal amount of memory as only one copy is stored. The downside is that read and take operations —when a matching tuple cannot be found in the local gspace kernel— are costly as other kernels will be queried sequentially whether they contain a matching tuple. This is particularly a problem in the case where the tuple can only be found in the local kernel of one specific process, as processes on other nodes will perform a linear search to locate this kernel, on average requiring half the number of kernels to be queried.

Besides the *SL* distribution policy, *GSpace* offers several other policies (see section 4.2.2 for an overview), which can be divided into two categories. First there are two policies that introduce caching by storing local copies of previously read tuples, which speeds up repeated reading of the same (remotely stored) tuple.

The second category are replication policies that duplicate tuples over a set G of *GSpace* kernels. This duplication increases the cost of put and take operations (as tuples have to be stored and removed from multiple kernels) and requires more memory to store tuples in each kernel of group G; the advantage is that read operations can be performed locally for processes executing on nodes in set G, while nodes outside G will only have to query one of the nodes in G. The various replication policies differ in how the set G is composed.

Although *PROOSA* currently uses more than ten different types of tuples, they can be categorized in five categories according to their access patterns:

1. global tuples that are updated on every access
2. tuples produced by a master
   (a) taken by a single worker
   (b) read my multiple workers
3. tuples produced by worker(s)
   (a) taken by a master
   (b) taken by a single worker

Tuples of category 1 contain the value of global variables (e.g. the number of workers, or the next unused simulation ID) and processes need to access these only once per simulation. Under the *Store Locally (SL)* policy, processes will —on average— need to query half the *GSpace* kernels to locate the single instance of these tuples. Since the values are updated on every tuple access, caching policies do not improve the situation as any cached copy will always be invalid. With the replication policies, the tuple will always be stored in the kernel of any member of group G; by choosing a group of size 1, processes will be able to locate the tuple directly, while at the same time minimizing the costs of replication. A good policy for this category will therefore be *Fixed Replication* or *Static Replication* on a group of size 1.

Category 2a encloses the tuples that are produced by a master and are consumed by workers to prevent other workers from using the same tuple (e.g. Task tuples). The *SL* policy requires workers to —on average— query half the remote kernels before a take operation is performed in the kernel local to the master process. Caching policies do not offer any improvement as there is no repeated access to these tuples. With a replication policy, processes will only have to query one of the nodes of G; because all operations are `take` actions, they will always be performed on one specific member of G, called the *sequencer*. Therefore it has no advantages to choose a group size larger than one: *Fixed Replication* or *Static Replication* on a group of size 1 are good choices for this access pattern, *Producer Replication* may also be considered when there will only be one master process (the advantage is that the master will be running on the sequencer node, resulting in local storage of tuples).

The tuples categorized under 2b are *read-only* tuples, which for example contain an experiment or a partition of the sample. Not only will the *SL* policy require processes to query multiple kernels for each read operation in order to locate the kernel m where the master process stores its tuples, it may also introduce a bottleneck as these tuples typically are large and m is responsible for serving the requests from all worker process. Although a process only incidentally needs repeated access to one of these tuples, the caching policies still reduce communication costs: read operations not only have to be served by m, but may also be handled by other kernels which contain a cached copy of the requested tuple. The resulting behaviour is similar to the data distribution in peer-to-peer file sharing networks. *Push caching* will be the best performing caching policy, as this will avoid validity checks of cached tuples, which of course are not necessary for read-only tuples. Another way to reduce search time and avoid the bottleneck is by using a replication policy, as worker processes will only have to query one of the members of group G; the performance will be optimal when the kernels local to the worker nodes are all members of G: the *Full Replication* and *Consumer Replication* policies are therefore the best choices.

Category 3 represents tuples created by workers, containing the results of computations. If one node runs a master process and W nodes run worker processes then, under the *SL* policy, the master process collecting the results will on average have to query $\frac{1}{2}W$ remote nodes to obtain a single result; to collect all W results, $\frac{1}{2}W^2$ remote queries are required. Caching policies do not improve this situation as there is no repeated read access to these tuples. Replication can be used to reduce the search times as only one node of the group G, that replicate the tuples, will have to be queried. Since only one process is interested in a certain tuple, groups of sizes larger than 1 offer no advantages. Note that replication with a group of only one node g may introduce a bottleneck, as this node will have to handle the storage and retrieval of typically large tuples; performance tests will have to prove whether this weighs up to the reduced search times. Policies to consider for 3b are therefore *Fixed Replication* and *Static Replication* with a group size of 1. For category 3a, performance will be optimal when kernel g is local to the master process: when it is known that there will only be one master process, *Consumer Replication* will be the optimal replication policy.

In the current *GSpace* implementation, the *Store Locally* performs a linear search over all remote kernels when a matching tuple cannot be found in the

local kernel. Because space based programs often have a producer/consumer design, the probability is high that consecutive `take` operations by consumers can be serviced by the same remote kernel where a producer stores its tuples. It would therefore make the *SL* policy much more efficient if its implementation was changed to start each linear search in the remote kernel where the last successful `read/take` took place.

Chapter 8

Results

To assess the performance of the proosa implementation, a number of experiment simulations were performed. Their results and an analysis form the content of this chapter.

8.1 Test environment

All simulations were performed on a Beowulf cluster consisting of 17 identical machines. Each of these has a single Pentium 4 CPU (clocked at 3.06 GHz, with HyperThreading support) and 2 Gigabytes of RAM. All machines are connected via a private, local gigabit ethernet by means of a 24 port 3COM switch.

The machines are running the Gentoo Linux operating system. All Java program code was executed in the server virtual machine of the Sun Java 2 Platform (Standard Edition version 5.0).

Simulation times were measured by performing 11 simulation runs in sequence. The time of the first run was discarded (its purpose was to "warm up" the Java Virtual Machine, i.e., to make sure the most frequently executed methods were compiled into native machine code); the reported times are the average time of the 10 remaining runs.

In the analysis of simulations, the time spent in fragments of the program was measured and classified as either computation, communication and idle time. Because it was often hard to discern communication from idle time (this would have to be measured within the *GSpace* kernel), the following approximation was used: when a node performs a blocking read/take for a certain tuple, it was classified as idle time (thus masking communication time as idle time), any consecutive read/take operations were classified as communication time (possibly masking idle time as communication time).

8.2 Sequential simulation

As a reference, the simulations were first performed by the sequential version of the simulation program; these values will be used as baselines to compare the performance of the parallel version to.

The simulation setup consists of two experiments: a *SRD* experiment followed by a *SFS*. The sample used is a NaCl crystal lattice, consisting of N particles. In the distance computations, periodic boundary conditions were used.

For various sizes of N, the average times over 10 simulations runs (in seconds) are reported in Table 8.1:

	16k	32k	64k	128k	256k
T_{seq}	13.0	49.5	201.1	830.2	3282.8

Table 8.1: sequential simulation (average time in sec)

Since the computational effort required to compute a radial distribution is $\mathcal{O}(N^2)$, and just $\mathcal{O}(N)$ for a scattering experiment, the *SRD* experiment is expected to dominate the simulation time. As the number of unique particle pairs to evaluate is $\frac{N(N-1)}{2}$, increasing the sample size by a factor 2 results in a factor 4 increase of particles pairs:

$$\frac{2N(2N-1)}{2} = \frac{4N-2}{N-1} \cdot \frac{N(N-1)}{2}$$

Therefore, doubling the sample size is expected to lead to approximately a factor 4 increase in simulation time. This factor is clearly visible in Table 8.1.

8.3 Parallel simulation

8.3.1 Simulation 0

The first parallel simulation, will perform the same *SRD* and *SFS* experiment as in the sequential simulation on a sample size of $N = 128k$ (i.e., a sample consisting of 131072 particles). There will be a single master process M and $W = 16$ worker processes, each running on a separate node with its own local *GSpace* kernel.

For all strategies the most basic ones are chosen:

- tasks are assigned to the workers randomly: each worker removes available tasks until all have been performed,
- accumulation is performed by the master: all workers write their partial outcomes to the tuple space, which are collected by the master and combined into the outcome of the experiment.
- the sample is partitioned into W sets, resulting in $\frac{W(W-1)}{2} = 120$ inter- and $W = 16$ intra-tasks for the SRD experiment. The SFS experiment will not partition the sampled range of q-values: each worker will compute partial observations for all q-values over a subset of the SRD bins.

Finally, the *store locally* policy is used for all types of tuples.

Theoretical analysis

For the SRD and SFS simulation on a sample of N particles and performed by W worker nodes, the execution time required for each phase is characterized in the following table:

phase	**action(s)**	**time required**
initialization	gather workers	$\mathcal{O}(W)$
	parse/store sample	$\mathcal{O}(N)$
srd	create experiment	$\mathcal{O}(1)$
	create tasks	$\mathcal{O}(W^2)$
	distribute tasks	$\mathcal{O}(W)$
	compute outcome	$\mathcal{O}(N^2 \div W)$
	accumulate outcome	$\mathcal{O}(N \times W)$
	finalize	$\mathcal{O}(1)$
sfs	store srd data	$\mathcal{O}(N)$
	create experiment	$\mathcal{O}(1)$
	create tasks	$\mathcal{O}(1)$
	distribute tasks	$\mathcal{O}(W)$
	compute outcome	$\mathcal{O}(N \div W)$
	accumulate outcome	$\mathcal{O}(N \times W)$
	finalize	$\mathcal{O}(1)$
finalization	remove sample	$\mathcal{O}(N)$
	store outcome(s)	$\mathcal{O}(N)$

In typical simulations, where $N \gg W$, the computation phase of the SRD experiment is expected to dominate the total execution time.

Observed performance

The results of simulation 0 are visualized in figure 8.1. For each phase of the simulation, shades of gray are used to visualize how the processes spend their time: the light gray color is used to show computation time, medium gray color stands for communication and the dark gray signifies idle time (waiting for synchronization, i.e., awaiting the availability of a tuple in tuple space). The small arrows signify the moments where tuples are stored or retrieved on which the processes synchronize: J stands for Join-, E for Experiment-, T for TaskConsumer- and O for Outcome-tuples.

The figure clearly shows that most of the simulation time is spent by the master, in the accumulation phase of the SRD experiment (409.04 seconds), which was characterized as $\mathcal{O}(N \times W)$. The large communication time that is needed to read W partial outcomes from tuple space is caused by

- absence of parallelism: a single node, the master, sequentially reads all partial outcomes from tuple space and accumulates them to its local outcome. This leads to the multiplication by the factor W.
- high costs per read/take operation: which is mainly caused by the large size, $\mathcal{O}(N)$, of the outcome tuples read.

The total execution time of the master process, which is the total time a user needs to wait for the outcome of the experiments to be available, will be used

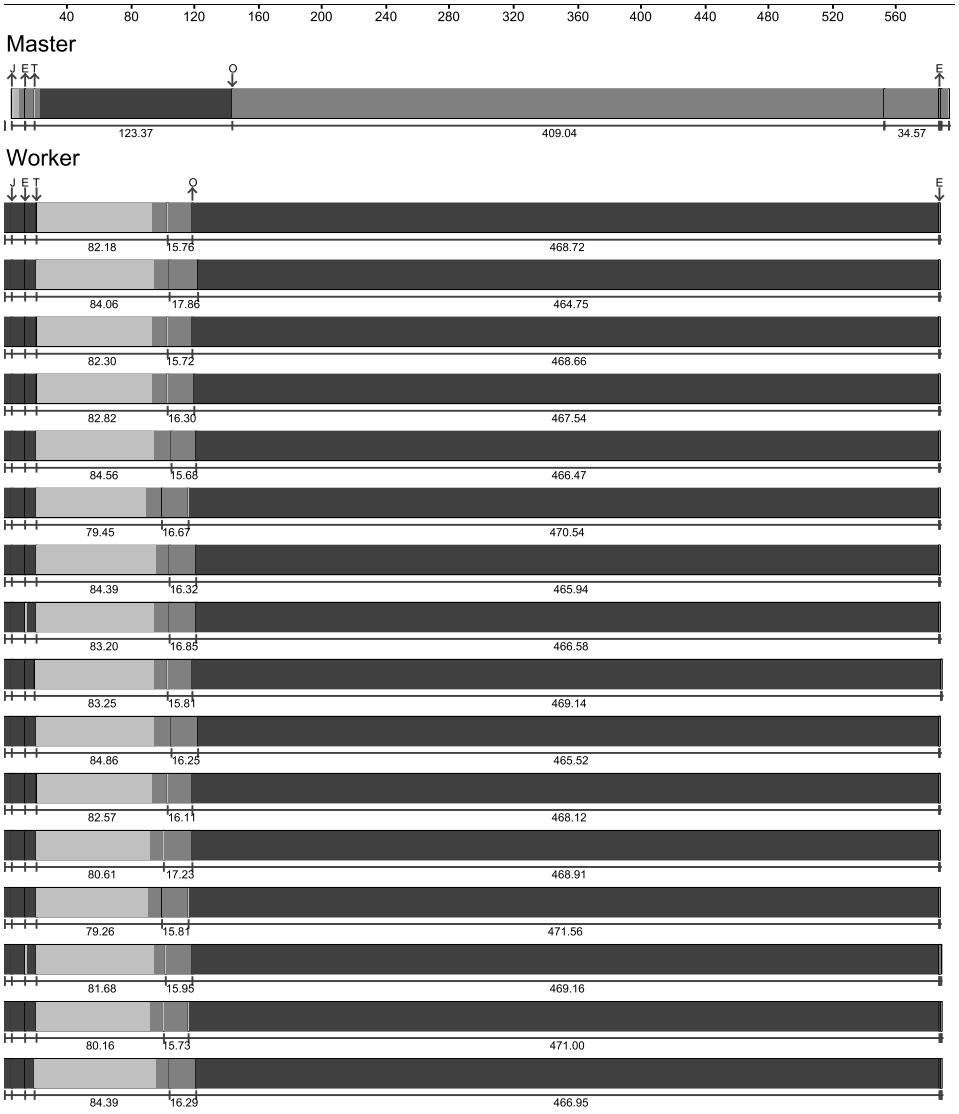

Figure 8.1: random task distribution, outcomes accumulated by master (Simulation 0)

to quantify the performance of the parallel program. For this run of simulation 0, the total execution time was 587.8 seconds.

Because there is hardly any improvement in performance over the sequential simulation, the experiment setup will be modified in the following simulations, to try to tackle the problems identified above.

8.3.2 Simulation 1

This simulation strives to introduce parallelism in the accumulation phase of the SRD experiment, to reduce the bottleneck formed by the master.

There are several possible approaches to introduce parallelism in the accumulation of outcomes. The strategy used in this simulation is targeted towards maximum parallelism: with W workers available, the outcome is partitioned into W sets. Each worker is assigned one of these outcome partitions to accumulate. For this, it needs to read $W - 1$ partial outcome partitions of the other workers and integrate them with its own partial outcome partition.

All workers require approximately the same amount of time, T_{acc}, for the accumulation of their assigned part, and no worker can complete before all others have made their partial outcomes available to the other workers. Therefore, all workers should start their accumulation phase at around the same time to keep the accumulation time as small as possible.

This may require better load balancing of the observation computations over the workers. Should this not be possible, another approach is to use less than W workers in the accumulation phase, so the worker that last completes its (partial) observation computations, does not require T_{acc}, which would delay the computation considerably, but only the time need for writing its partial outcome partitions to tuple space.

Theoretical analysis

As described in section 8.3.1, the introduced parallelism should decrease the effect of factor $\mathcal{O}(N \times W)$ in the SRD outcome accumulation; by using W workers to perform the accumulation on, it is expected that this factor changes to $\mathcal{O}(N)$.

Observed performance

The result of distributing the accumulation over the workers is displayed —for a typical run— in Figure 8.2; the total execution time of the simulation decreased from 587.8 seconds to 362.8 seconds.

Introducing parallelism into the SRD outcome accumulation had a large impact on the performance, but this phase still requires a considerable amount of time on the workers: it covers more than half of the total simulation time. For example, the first worker spent 205.94 seconds on it. Therefore, the next step will be to try to speedup the reading and writing of partial outcome partitions.

8.3.3 Simulation 2

The goal of this simulation is to lower the time required for storage and retrieval of (partial) outcome tuples. The time needed to obtain a tuple from tuple space, which matches with a template t, consists of

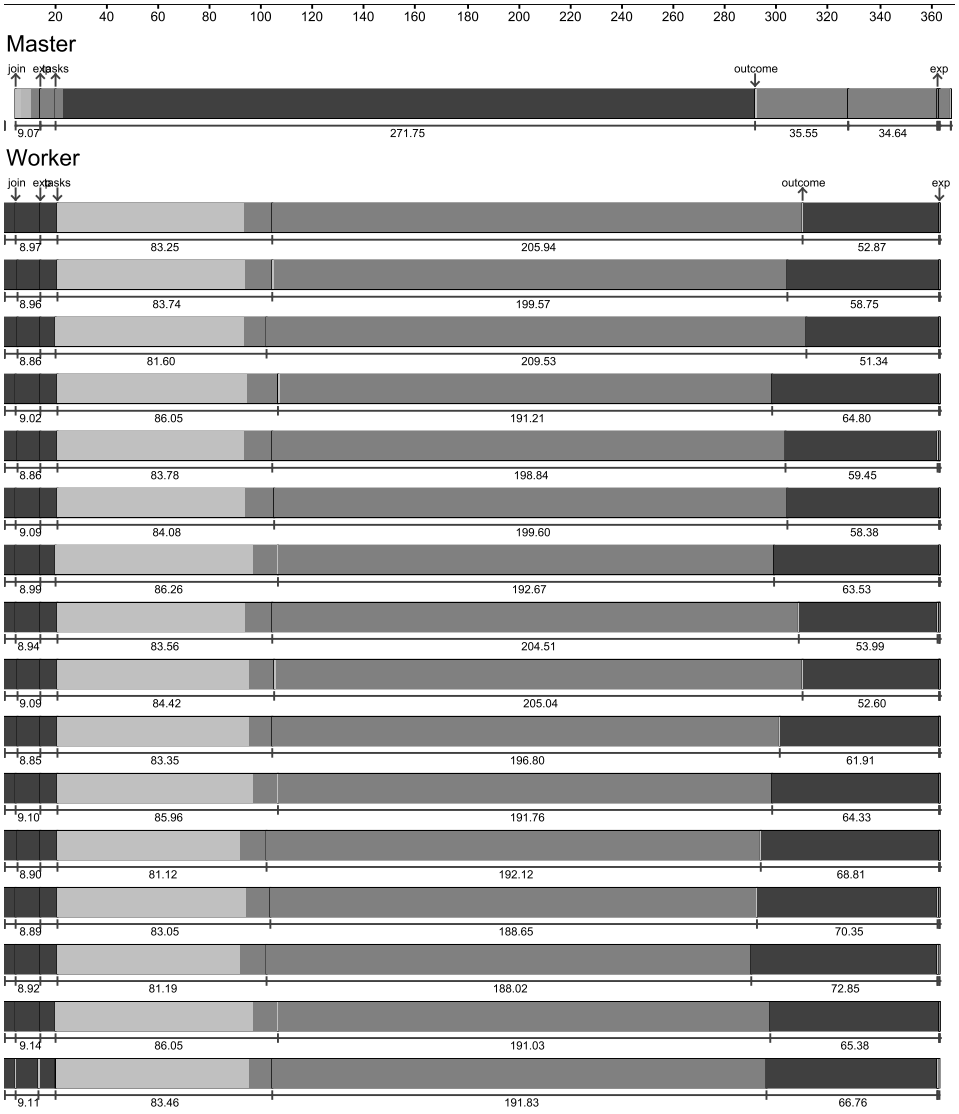

Figure 8.2: random task distribution, outcomes accumulated by workers (Simulation 1)

(i) search time: the distribution policy for the type of t determines where matching tuples can be found. In the store locally and caching policies, each node can possibly contain a matching tuple in its local slice of the tuple space; searching time will therefore be of order $\mathcal{O}(P)$ (where P is the total number of tuple space kernels, usually equal to the number of nodes). For replication policies, the search time will be $\mathcal{O}(1)$, as only one of the nodes in the replicating group will have to be queried.

(ii) waiting time: when no matching tuple is available, the operation blocks until a tuple, added by another process, can be obtained by the blocked process.

(iii) transfer time: when a matching tuple is found in a remote kernel, it has to be transferred to the machine on which the searching process is running. The required time, consisting of time for serialization, data transport over the network and deserialization, are determined by the size of the matching tuple.

The costs for putting a tuple in tuple space, are determined by

(iv) transfer time: which, as for reading/taking tuples, is determined by the tuple size.

(v) distribution policy: if the policy specifies a tuple to be replicated in multiple slices of the tuple space, these transfer times are additional to the transfer time of putting the tuple in the local slice of the process that executes the `put()` operation.

Therefore simulation 2 will incorporate the following changes to its setup:

- a custom implementation of the serialization/deserialization for the objects involved: the advantage over Java's default serialization mechanism are shorter (de-)serialization times and smaller amounts of binary data.
- replacing the outcome tuples by sparse tuples, i.e., only the values that differ from a certain default value are included and transferred. For SRD outcomes only the nonempty bins will have to be transferred. This filtering of values does require more computation time on the process that writes the tuple, but can have a big impact on the amount of data to transfer.

Theoretical analysis

The time the master has to wait for the SRD outcome to be available consist of the time needed for (a) the observation computations, (b) the distributed accumulation and (c) the transfer costs to the master, or:

$$\mathcal{O}(N^2 \div W) + \mathcal{O}(N) + \mathcal{O}(N)$$

The effects of the above changes do not change this characterization, i.e., it does influence terms $\mathcal{O}(N)$ but the relation remains linear with N. It is however hard to predict by how much the times will decrease.

Observed performance

The modifications for this simulation had their intended effect: the total execution time of the master process decreased to 147.1 seconds.

The main reason for this decrease is the introduction of sparse outcome tuples. To obtain a good approximation, the number of bins for typical simulations will be large; this often results in a large number of empty bins. In the current simulation, where the sample is a regular crystal lattice, the number of distinct distances is even more limited, increasing the empty bin count even further.

Looking at the visualization of the execution (Figure 8.3), it easy to see where the most time can be won: prior to each experiment, the master stores the required data in tuple space. For the second experiment, this is the outcome of the first experiment; in the visualized simulation run, for example, this phase required 36.39 seconds.

These large times could be reduced by applying the same modifications as was done for the accumulation phase (i.e., customizing the serialization process and reducing the amount of data by only storing non-empty bins). They can however be circumvented altogether by changing the program design: the master should not remove the outcome partitions in the accumulation phase of the first experiment, but only read each one exactly once. In the second experiment, the workers can then consume the outcome partitions.

Since this is such a small modification, it has been applied to the current simulation. A run of this modified simulation is visualized in figure 8.4; the total execution time has decreased to 110.0 seconds.

The main problem now is that the workers spend a considerable amount of idle time before they can start computing.

8.3.4 Simulation 3

In the final simulation setup, it is first tried to reduce the idle waiting time of the workers at the start of the simulation. This idle time consists of waiting for

1. a master process to be started,
2. the sample particle partitions to be stored in tuple space,
3. the storage of tasks.

Of these three items, the first is beyond control of the application and only the second and third can be influenced.

Item two is caused by the large time required to parse the sample from file: an $\mathcal{O}(N)$ operation. In the previous simulation setups however, the master process first parsed a block of particles from the sample file and consecutively stored the block in a tuple; as both these operations involve a lot of waiting for I/O, performing them in separate threads should lead to an interleaving that reduces the total time required. To easiest way to achieve this behavior is by enabling the multirequest setting in *GSpace*: with this setting, tuple operations performed by a process will each be serviced in its own thread.

The large time required for storing task tuples, item three, is a result of the current *GSpace* implementation: when a tuple is inserted, all kernels (sequentially) are informed of this event, which allows them to retry `read/take`

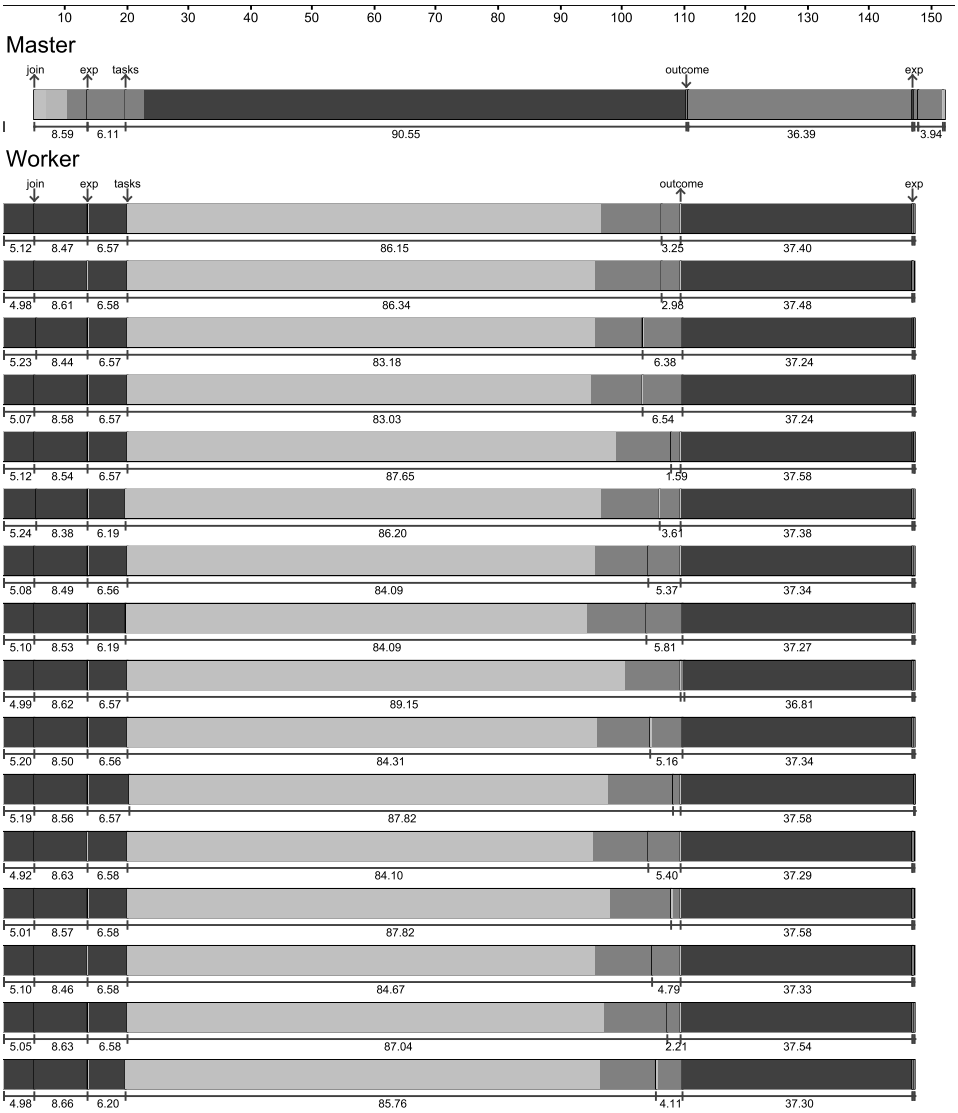

Figure 8.3: customized serialization and reduced data (Simulation 2)

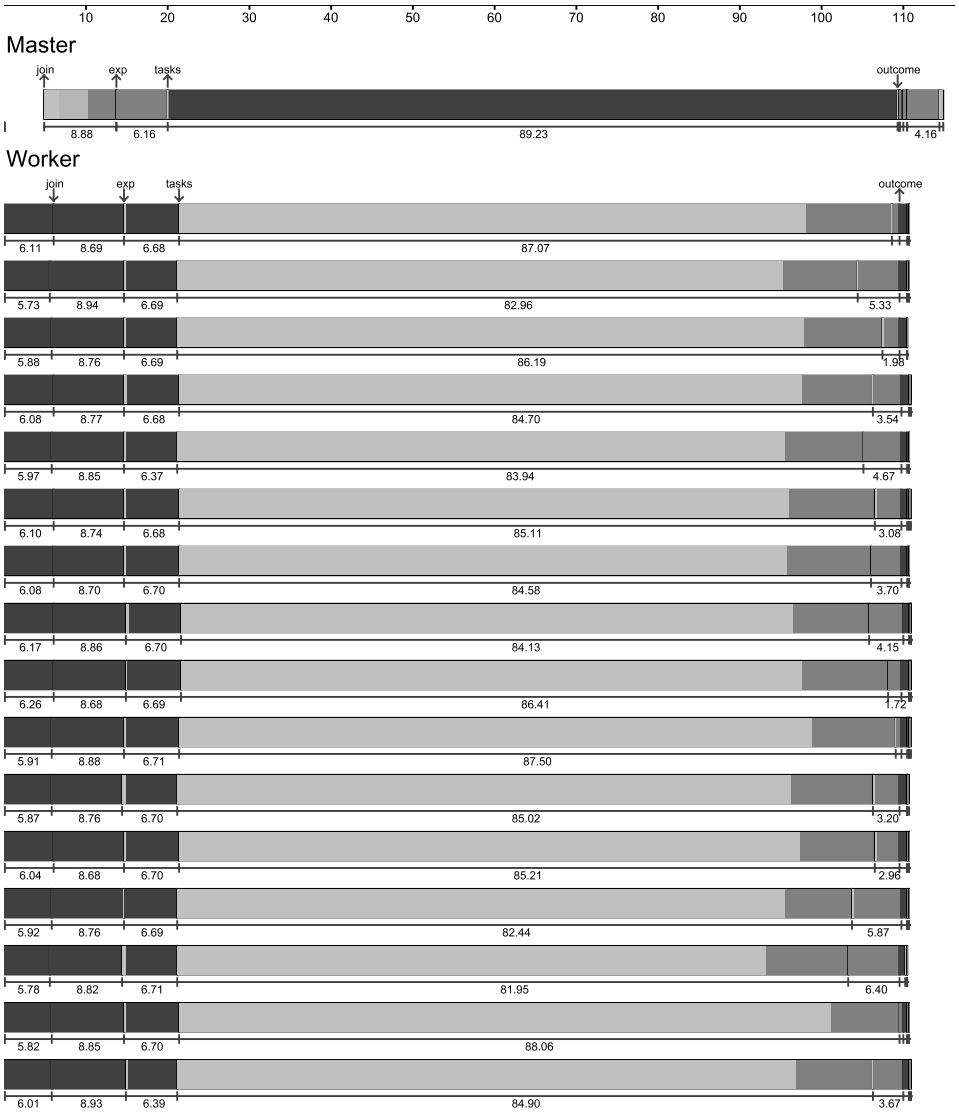

Figure 8.4: master process redesign (Simulation 2.1)

operations that were blocked on the type of tuple inserted. This $\mathcal{O}(W)$ operation is repeated for each task tuple that is inserted. As the number of tasks was $\mathcal{O}(W^2)$, there are $\mathcal{O}(W^3)$ communication actions performed to inform a single kernel of the insertion of one task tuple. Resolving this issue requires changes to the *GSpace* implementation, which is beyond the scope of this work. One possible approach would be to introduce a `mput(List<Tuple>)` operation, which can be used to put multiple tuples of the same type, requiring only one insertion notification to be sent to all remote kernels.

A second change to the simulation setup is the assignment of SRD tasks to the worker processes. In the previous simulations this assignment was random: workers selected a random task from tuple space, retrieved the particle partition(s) on which that task was based and then computed the particle pair distances. Since these tuples were governed by the Store Locally policy, repeated access to the same particle partition didn't bring any advantages in communication costs. A different distribution policy could be selected to reduce communication costs by locally available cached or replicated copies, but as the set of tasks has size $\mathcal{O}(W^2)$, the probability of repeated access is low. Therefore, the random task assignment will be replaced by a different strategy: each worker process is assigned one partition p of the sample and tries to process as many tasks as possible that are dependent on the particles contained in p. When no more of these tasks exists, the worker will continue processing other (randomly selected) tasks, which will balance the computation when one or more other workers are slower in processing their (preferred) set of tasks.

Theoretical analysis

The proposed changes do not transform the program into a more efficient design, but they should lead to more efficient usage of resources: multithreading will lead to interleaving of file and network I/O, the new task allocation will maximize reuse of previously read data.

Observed performance

Figure 8.5 shows that the changes did result in better performance: the workers' idle time is shortened and they can start computing earlier, the SRD computations are performed faster because less particle partitions have to be read. For the visualized run the simulation time (on the master) decreased to 99.9 seconds.

8.3.5 Evaluation of distribution policies

Until now, all performed simulations used the basic *Store Locally* distribution policy for storage and retrieval of tuples. In section 7.3, the other available distribution policies were evaluated for each of the tuple access patterns in *PROOSA*.

It was intended to measure the performance improvements (that were expected as a result of) the other recommended policies here, however, the implementation of *GSpace* that was used in this thesis prohibited this as

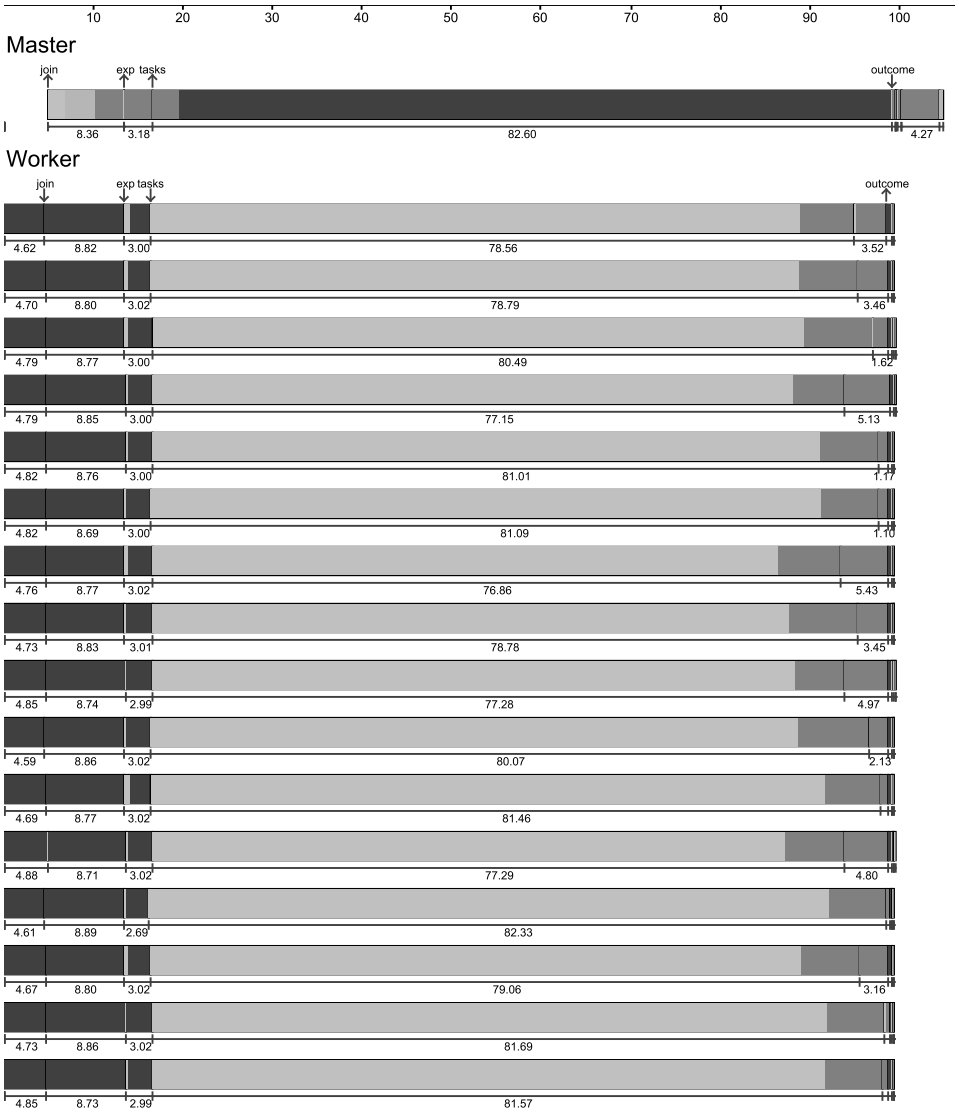

Figure 8.5: optimized task allocation, multithreaded `put` (Simulation 3)

- it was not yet able to handle the large (particle partition) tuples in the replication policies, which use multicasting to replicate the tuples;

- the method to remove cached or replicated tuples is very inefficient, decreasing the total performance: when a tuple t is removed that is governed by a caching or replication policy, the sequencer or master node that controls t will send a message —containing t— to the other kernels, with the instruction to remove it. For large tuples, which are common in *PROOSA*, this not only results in a considerable amount of data transfer, but also in a deep compare of t with the tuples stored in the kernel that received the message.

8.4 Scalability

To determine how well the program scales, a number of simulations have been performed with the last used setup (see section 8.3.4); the only difference is that the heap size of the Java Virtual Machine was restricted to 64 MB, which proved to lead to shorter absolute simulation times and smaller variance between the runs.

The results, for various sample sizes and workers amounts, are shown in Table 8.2, which lists the average simulation times over 10 consecutive runs.

	16k	32k	64k	128k	256k
T_{seq}	13.0	49.5	201.1	830.2	3282.8
$T_{par(1)}$	17.0	62.9	229.7	853.4	3418.7
$T_{par(2)}$	12.8	43.5	156.8	618.1	2277.9
$T_{par(4)}$	7.7	23.8	84.4	315.5	1149.0
$T_{par(8)}$	5.9	14.8	47.9	154.6	583.6
$T_{par(16)}$	11.5	15.2	31.6	92.3	310.6

Table 8.2: absolute simulation time (in seconds)

As was argumented for the sequential case in section 8.2, doubling the sample size leads to roughly a factor 4 increase in simulation time. In Table 8.2, this factor is visible is the simulation times of the larger samples; the sample needs to be of a large enough size before the time needed for the $\mathcal{O}(N^2)$ SRD computation dominates the other $\mathcal{O}(N)$ phases (see section 8.3.1).

	16k	32k	64k	128k	256k
$S(1, N)$	0.77	0.79	0.88	0.97	0.96
$S(2, N)$	1.02	1.14	1.28	1.34	1.44
$S(4, N)$	1.69	2.08	2.38	2.63	2.86
$S(8, N)$	2.22	3.33	4.20	5.37	5.63
$S(16, N)$	1.14	3.26	6.36	8.99	10.57

Table 8.3: speedup

The effect of the number of workers on the performance is more clearly visible in Table 8.3, which lists the speedup $S(P, N)$ that is defined as:

$$S(P,N) = \frac{T_{seq}(N)}{T_{par(P)}(N)}$$

In the ideal case, doubling the number of parallel cuts simulation time in half or doubles the speedup. In *PROOSA*, this factor is less than 2 because some phases, mainly the creation and distribution of tasks, were designed to be of $\mathcal{O}(W^2)$ in order to allow a balanced load distribution. Furthermore, the cost of tuple retrieval under *GSpace's Store Locally* policy is also of $\mathcal{O}(W)$.

	16k	32k	64k	128k	256k
$E(1,N)$	0.77	0.79	0.88	0.97	0.96
$E(2,N)$	0.51	0.57	0.64	0.67	0.72
$E(4,N)$	0.42	0.52	0.60	0.66	0.71
$E(8,N)$	0.28	0.42	0.52	0.67	0.70
$E(16,N)$	0.07	0.20	0.40	0.56	0.66

Table 8.4: efficiency

However, the increase of overhead dependent on the size of W as described above, does not explain the poor speedup which is experienced when switching from 1 to 2 workers. It is more clearly discernable in Table 8.4, which shows the efficiency, defined as:

$$E(P,N) = \frac{S(P,N)}{P}$$

Examination of the execution of the simulations for $N = 128k$ and $P = \{1,2\}$ learns that there is a lot of deviation in the time needed for processing SRD tasks. The times needed for computing the distances within a single sample partition (in this case, a partition of size $64k$) ranged from 201.2 seconds (cf. $T_{seq}(64k)$) up to 315.4 seconds. Similarly, computing the distances between two sample partitions showed times between 456.5 and 623.2 seconds. Note that these measurements only include the computation time, as at the start of the measurement the required data was already available in the workers' local memory and no actions were performed that include waiting or blocking. Also, the loads in each run were perfectly balanced, where both workers computed $N^2/4$ particle pair distances. During simulation, the *GSpace* kernel was the only other running process of importance, but the external requests it receives during in the SRD computation phase, cannot possibly require a large amount of CPU time to be causing this.

At the time of this writing it is still unclear what is the cause of this deterioration in performance, but it is likely to have to be sought in the Java virtual machine that was used to run the simulations in.

8.5 Further suggestions for improving performance

The total execution time of a program, T_{exec} consists of several terms:

$$T_{exec} = T_{comp} + T_{comm} + T_{idle}$$

where

- T_{comp} is the time spent in computations,
- T_{comm} is the time spent for communication, and
- T_{idle} is time spent idle, e.g. while waiting for synchronization.

To obtain the best possible speedup in a parallel program, the computation time should be equally divided over all concurrent processes, communication time should be minimized and idle time should be eliminated altogether.

Idle time

Looking at the execution of the last performed simulation, Figure 8.5, it is easily discernable where the most idle time is spent:

1. by the master, while waiting for the SRD outcome to be computed;
2. by the workers, while waiting for the tasks and data to become available before they can start their computations;

and finally, some idle time is masked as (a part of the) communication time

3. by the workers, while waiting partial outcome partitions from other workers.

The first item is not really a problem, as this idle time could be used by a worker process, running on the same node as the master, to perform part of the computations.

The second item however, has a large impact on the simulation performance. Its main cause is the design decision to first store all data and tasks in tuple space, before the workers are allowed to start computing; this approach was chosen as it allowed the workers to assert that all tasks were consumed when no more could be found in tuple space. A better approach is to let the workers start during the master's task writing, which requires the introduction of an additional tuple for the workers to be able to assert the emptiness of the task pool. Because tasks cannot be performed before the required data, i.e. one or two sample partitions, are available, an optimal solution would interleave the writing of sample partitions and tasks: once a partition j is added, the tasks that are dependent on j and on one of the earlier stored partitions i ($0 \leq i < j$), can be safely added to tuple space.

A minor improvement may be to replace the multithreaded handling of tuple space operations, which was used in Simulation 3 to interleave file I/O and network communication, by a single concurrent thread responsible for storing tuples. Tuples to be stored are then added to a blocking queue, instead of being passed to a newly created thread: not only will this reduce the required resources, but also guarantee that the order of writing tuples is maintained, which may be needed for program correctness.

Note that in the current implementation, all experiments are simulated in a strictly sequential order. Although it would make only a slight difference here (as the SRD experiment time dominates the time for the SFS experiment), in general a better approach is that the master, while waiting for the results of the current experiment, already starts preparations for the following experiment: storage of any required data, the experiment itself and the set of tasks.

The third item is caused by the imbalanced workloads, which are here caused by the issue that was identified at the end of the previous section: the large variance in processing time for tasks of the same size. If this issue cannot be solved, a different accumulation strategy may be needed, which allows workers that have completed their computations to do part of the accumulation, while others are still performing observations. This can be as simple as limiting the number of workers, that perform the parallel accumulation to a subset of size N, formed by the workers that first complete their computations. An alternative is a tree-based algorithm, in which an accumulation step can be performed as soon as two partial outcomes become available.

Communication time

As a certain (minimal) amount of communication cannot be avoided, and in the current simulation setup only a minimal amount of tuple space operations are performed (the only exception being the repeated reading of a particle partition that is not the worker's preferred partition, which only occurs incidentally), the possibilities for decreasing T_{comm} seem to be limited. There are however a few things that can be done. In the current *PROOSA* simulation software, communication costs can be lowered by

1. replacing Java's default serialization mechanism by a custom implementation: as *GSpace* uses serialization to send objects (tuples) over network connections, a customized serialization implementation can reduce the time needed for (de-)serialization and the amount of data to be transferred. This is expected to have a big impact on the communication costs of sample partition tuples.

2. using distribution policies with performance characteristics that match a tuple's access pattern: for *PROOSA*, this will require modifications to the current *GSpace* implementation to enable it handling large tuples efficiently.

3. reducing the amount of data to transfer, by compression; this is comparable to the replacement of outcome tuples with sparse tuples (in which only non-default values are stored). An alternative can be easily implemented in *GSpace*, by decorating the streams that communicate over sockets with a (de-)compression stream, which performs data (de-)compression transparently.

Finally, a few simple extensions to the API of *GSpace* can have a big impact as well

4. `mput(Vector<Tuple>)` for the insertion of multiple tuples in a single operation: the advantage being that the other kernels will only have to be

notified a single time of this insertion event (which is required as application threads may currently be blocked on a `read/take` operation who's template matches with one of the newly inserted ones).

5. `remove(t)` which removes all tuples matching with `t`. This operation is useful for post-simulation cleanup: currently, the P sample partitions are removed by performing P `take`-operations which, although the tuples are stored in the local kernel, are costly because serialization over a socket connection is used to transfer tuples between the application process and the local *GSpace* process. Furthermore, the retrieved tuple is not used and will immediately be discarded

Computation time

The computation time will be minimal when (i) all the computational work is perfectly distributed over the available workers and (ii) the most efficient algorithm, i.e. the algorithm with the lowest computational complexity, is used.

Although in the current SRD simulation —where the number of sample partitions is chosen equal to the number of workers— a balanced load distribution exists, and the load balancing strategy is able to achieve this distribution, large variations in the processing time per task prevent the workers from ending their computation phase at around the same time. Should the issue —that causes this variance— not be resolved, increasing the number of tasks (which may require using the proposed `mput` operation to be efficient enough) is expected to lead to more equal computation phases.

The SFS simulation only accounts for a small part of the simulation time, due to its $\mathcal{O}(N)$ computational complexity and the small amount of q value samples and non-empty bins (a result of the regular structure of the sample: a crystal lattice). For problem sizes as used in the current simulations, it will therefore be more efficient to perform the SFS simulation by sequential simulation on the master, as the communication costs for distribution and accumulation are larger than the required computation time.

Chapter 9

Conclusions

In the final chapter, the work described in the earlier chapters will be evaluated, followed by a number a recommendations for future work.

9.1 Discussion

The *(PR)OOSA* model

The basis of the simulation software is formed by the *OOSA* model; when experiments to be simulated are expressed as functions, the *OOSA* model provides an object-oriented domain model of functions, in which it should be easy to define new experiments by only defining the relation between argument and result values and their corresponding quantities.

The *PROOSA* model extends *OOSA* for parallel computation: it proposes a way to subdivide the computation of an observation (i.e., a single result value as a function of one of more argument values), in order to distribute the computational work over a number of concurrent processes.

Of course it is hard to predict how well this model suits other experiment simulations, or even completely different applications, where the work to be performed can be expressed as a functional relation. However, the chosen approach of modeling as functions is very general and should be widely applicable. Furthermore, the resulting domain model should be easy to understand. How well the proposed parallelization works for other simulations, i.e., whether it is general enough to apply and how it performs, will require further research.

Parallel simulation of scattering experiments

The *PROOSA* model was applied to the domain of X-ray scattering experiments; the simulations that were performed achieved a reasonable speedup, while there is still room for improvement (see section 8.5 for a number of suggestions).

On a side node, scattering analysis is a problem that lends itself very well for parallelization. The communication costs are linear with N, the amount of data, while the computational work is $\mathcal{O}(N^2)$. Therefore, increasing the value of N will automatically lead to better performance as the ratio between computation and communication improves. In practice however, problem sizes will be much

larger that those used the simulations presented here (which were limited by time constraints).

Furthermore, the test were performed on a dedicated set of nodes, interconnected by a high speed communication network. It would be interesting to test the performance of the current implementation on a number of (heterogenous) machines, connected via a LAN or even the internet.

Tuple space paradigm

An implementation of the tuple space model, *GSpace*, was used in the parallel simulation software to provide the communication layer between the nodes.

The concept of tuple spaces turned out to be easy to understand and intuitive to use. It is a natural match with the chosen *processor farm* design, where the farmer sends out data and tasks as tuples; workers consume and process the tasks, storing the computed results in tuple space where the master can retrieve them.

The available implementation of *GSpace* was able to achieve reasonable communication performance that scaled linearly with the size of tuples; this allowed good scalability to be achieved in simulations when the problem size exceeded a certain minimum, which should not be a problem in practice.

GSpace's concept of distribution policies, which allow a space based application designer to select a tuple distribution strategy with performance characteristics that best match the access pattern in the application, looks very promising. Unfortunately, issues in the current implementation, mainly with respect to the handling of large tuples —for which it probably was not designed— made it impossible to test the effect of distribution policies on performance.

However, performance issues may occur when the program is scaled up to a larger number of nodes: there is a considerable amount of inter-kernel communication which is linear with the number of kernels (e.g. when searching remote kernels for matching tuples, or when one kernel informs the others of the insertion of a new tuple). Not only performance may be a limiting factor to the number of nodes that can be used effectively: in the replication policies, a lot of the inter-kernel communication is implemented by $\mathcal{O}(1)$ IP multicasting, but this feature may even not be available when switching to a wide area network that is needed to accommodate a large number of nodes.

9.2 Recommendations

Besides improvements to the simulation software and tuple space implementation, of which a number were suggested in the preceding chapter and that would lead to higher simulation performance, the work presented here leaves room for further research in a few directions, hoped to be leading to more usability or extended functionality:

- an inventorisation of typical and widely-used computer experiments, with respect to the types of the function they compute; this will give insight in how well the *OOSA* model and the chosen parallelization method are applicable in general, and what changes might be required to accommodate these experiments as well.

Furthermore, this allows performance testing of experiment simulations where the ratio between computation and communication is different from the $N^2 : N$ ratio in scattering experiments, or simulations where the observations cannot be computed independently but require the workers to coordinate by performing tuple operations.

- automated parallelization: in the current scattering analysis experiments, parameters supplied by a user are required to determine how the work is to be distributed over the workers. Ideally, the user should only have to define the functional relation for the experiment he/she wants to simulate, and the choice of an optimal distribution (based on an analysis of the computed function) should be the responsibility of the simulation framework.

- runtime reconfiguration: when switching to larger problem sizes, with larger running times, or to different environments —for example a LAN/WAN of heterogenous nodes in order to harvest unused resources— the probability of nodes leaving due to failure or a limitation on availability increases. The current system has no provisions to recover from failure, and the current requirement of nodes having to remain available is too limiting: it does not allow to make use of nodes that become available after the simulation is started, or of nodes that are available for a limited time only.

- support for larger scales of simulations, both in problem size and number of nodes. A larger problem size may require more data to be available than can be stored in (the combined) memory: in this case only a subset of the data —and the tasks dependent on it— can be made available at one time. When increasing the number of nodes, the tuple space implementation might be limiting the number of nodes that can be used effectively. If these limitations cannot be circumvented, e.g. when they turn out to be inherent to the tuple space paradigm, alternative methods have to be sought. An example is to use an alternative distribution for the problem data, which in the current scattering simulation is responsible for the most memory usage and communication time, by means of a peer-to-peer file sharing protocol; in such a setting, workers can obtain (parts of) their data from each other, avoiding the bottleneck where data is requested from a single source, and can be an alternative to multicasting which might not always be available.

Appendix A

External representation of class Sample

A.1 BNF Grammar

<SAMPLE>	::=	<SYSTEM> <SUBSTANCES> <GEOMETRY> <CONTENTS>
<SYSTEM>	::=	system <SYSTEM_ID> '\n'
<SYSTEM_ID>	::=	"regular expression for system identifiers"
<SUBSTANCES>	::=	substances '\n' { <SUBSTANCE> '\n' }
<SUBSTANCE>	::=	<SUBST_ID> <FORMFACTOR>
<SUBST_ID>	::=	"regular expression for substance identifiers"
<FORMFACTOR>	::=	<ATOMICSCATTERINGFACTOR> | ...
<ATOMICSCATTERINGFACTOR>	::=	asf { <REAL> }^{9}
<GEOMETRY>	::=	box '\n' <BOX_SIGNATURE> '\n' <SIZES> [<ANGLES>] '\n'
<BOX_SIGNATURE>	::=	{ <LENGTH_QUANTITY> }^{3} '\n' { <LENGTH_UNIT> }^{3}
<SIZES>	::=	<SIZE_X> <SIZE_Y> <SIZE_Z>
<ANGLES>	::=	<ANGLE_α> <ANGLE_β> <ANGLE_γ>
<SIZE_X>, <SIZE_Y>, <SIZE_Z>, <ANGLE_α>, <ANGLE_β>, <ANGLE_γ>	::=	<REAL>
<CONTENTS>	::=	particles <PARTICLE_COUNT> '\n' <POS_SIGNATURE> '\n' { <PARTICLE> '\n' }
<PARTICLE_COUNT>	::=	<INTEGER>
<POS_SIGNATURE>	::=	{ <LENGTH_QUANTITY> }^{3} '\n' { <LENGTH_UNIT> }^{3}
<PARTICLE>	::=	<POS_X> <POS_Y> <POS_Z> [<SUBST_ID>]
<POS_X>, <POS_Y>, <POS_Z>	::=	<REAL>

```
 <LENGTH_QUANTITY>        ::=      length
 <LENGTH_UNIT>            ::=      'E'<EXPONENT> m | Å
 <EXPONENT>               ::=      <INTEGER>
```

A.2 Example of a .sample file

```
system nacl_cube4096
substances
Cl-1__RHF asf +18.2915  +0.0066  +7.2084  +1.1717  +6.5337 +19.5424  +2.3386 +60.4486 -16.3780
Na+1__RHF asf  +3.2565  +2.6671  +3.9362  +6.1153  +1.3998  +0.2001  +1.0032 +14.0390  +0.4040
box
            length          length          length
            E-10 m          E-10 m          E-10 m
        +45.024000      +45.024000      +45.024000
particles 4096
            length          length          length
            E-10 m          E-10 m          E-10 m
         +1.407000       +1.407000       +1.407000       Na+1__RHF
         +1.407000       +1.407000       +4.221000       Cl-1__RHF
         +1.407000       +1.407000       +7.035000       Na+1__RHF
         +1.407000       +1.407000       +9.849000       Cl-1__RHF
         +1.407000       +1.407000      +12.663000       Na+1__RHF
         +1.407000       +1.407000      +15.477000       Cl-1__RHF
         +1.407000       +1.407000      +18.291000       Na+1__RHF
         +1.407000       +1.407000      +21.105000       Cl-1__RHF
               ...             ...             ...             ...
        +43.617000      +43.617000      +23.919000       Na+1__RHF
        +43.617000      +43.617000      +26.733000       Cl-1__RHF
        +43.617000      +43.617000      +29.547000       Na+1__RHF
        +43.617000      +43.617000      +32.361000       Cl-1__RHF
        +43.617000      +43.617000      +35.175000       Na+1__RHF
        +43.617000      +43.617000      +37.989000       Cl-1__RHF
        +43.617000      +43.617000      +40.803000       Na+1__RHF
        +43.617000      +43.617000      +43.617000       Cl-1__RHF
```

Appendix B

Extending *GSpace* with non-blocking operations

To illustrate the minor changes that are required to extend *GSpace* with non-blocking versions of the operations to retrieve tuples, the source code of the blocking `take` and non-blocking `takeIfExists` for the *Store Locally* distribution policy is included here.

Note that the code fragments used for profiling and verbose (debug) output were omitted for clarity.

B.1 Blocking `take(Tuple t)`

```
public Tuple take(Tuple template) {

    Tuple result = null;

    while (true) {

        result = ds.take(template); /* read from local dataspace */
        if (result != null) {/* is the result not null */
            return result; /* yes, return it */
        }

        /* the local result was null, search on the other nodes */
        register(template); /* register for notifications */

        /* return the first non-null instance */
        result = (Tuple) comm.multiInvocationReturnFirst(set,
                            dmClassName, "externalTake", template);
        if (result != null) { /* is the result non-null? */
            return result; /* yes, return it */
        }

        sleep(template); /* otherwise sleep until a tuple is available */
```

```
    }

}
```

B.2 Non-blocking `takeIfExists(t)`

```
public Tuple takeIfExists(Tuple template) {

    Tuple result = null;

    result = ds.take(template); /* read from local dataspace */
    if (result != null) {/* is the result not null */
        return result; /* yes, return it */
    }

    /* the local result was null, search on the other nodes */

    /* return the first non-null instance */
    result = (Tuple) comm.multiInvocationReturnFirst(set,
                        dmClassName, "externalTake", template);
    if (result != null) { /* is the result non-null? */
        return result; /* yes, return it */
    }

    return null; /* no matching tuple found, therefore return null */
}
```

Bibliography

[Bur06] Bureau International des Poids et Mesures. *The International System of Units (SI)*, eight edition, 2006.

[For94] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[Fow97] Martin Fowler. *Analysis patterns: reusable objects models*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[Gel85] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[GF55] Andre Guinier and Gerard Fournet. *Small-angle scattering of x-rays*. Chapman & Hall, London, UK, 1955.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Koo06] Marcel Koonen. Parallel object oriented scattering analysis using a service oriented architecture. Master's thesis, 2006.

[Mar85] Alain J. Martin. The probe: An addition to communication primitives. *Inf. Process. Lett.*, 20(3):125–130, 1985.

[Oxf96] *"experiment" The Oxford Dictionary of Philosophy*. Oxford University Press, 1996.

[RCvS04] Giovanni Russello, Michel Chaudron, and Maarten van Steen. Exploiting differentiated tuple distribution in shared data spaces. *Lecture Notes in Computer Science*, 3149:579–586, 2004.

[Rus06] Giovanni Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. PhD thesis, Eindhoven University of Technology, 2006.