Fostering Reuse in Scientific Computing with Embedded Components

Application to high-performance Bayesian inference for bioinformatics

Vincent Lanore

Univ. Lyon, Université Claude Bernard Lyon 1, CNRS, LBBE UMR 5558,
F-69622, Villeurbanne, France

vincent.lanore@univ-lyon1.fr

Component-based programming is a programming paradigm which eases software reuse but has yet to be widely adopted in scientific computing. We propose to embed component frameworks inside high-performance languages directly to improve flexibility compared to the literature. We present this approach through the example of a high-performance Bayesian inference application.

Developing high-performance code is a challenging balancing act between performance and productivity concerns. Optimizing performance is crucial for large-scale applications but can be extremely time-consuming and lead to code that is both complex and hard to maintain. On the other hand, productivity is important to reduce development cost, particularly in academia where developer time is scarce.[1]

Because they need fine control over implementation, and because they need high performance, many developers of computationally-intensive applications write them from scratch using low-level languages like C++.[2] Such hand-written codes are costly to develop and maintain, even more so if several versions must be maintained at the same time. In addition, this approach discourages reuse from third parties, by third parties, and between application variants. Indeed, good practices for software reuse (e.g., hiding implementation details, modular structure) tend to get in the way of low-level optimizations and fast prototyping of new ideas.

One possible approach to try address these reuse problems is component-based programming. This programming paradigm consists in building applications by assembling reusable pieces of software called components. This approach has been well-studied: it is known to greatly ease software reuse, improve separation of concerns, and provide a high-level representation of an application, making its structure easier to reason about. However, component-based techniques have yet to be widely adopted in scientific computing.[2]

We argue that existing component-based technologies for scientific computing suffer from their reliance on dedicated languages and external tools. These tools impose a strict separation between low-level business code and high-level structural concerns, which adds complexity and can get in the way of low-level optimization and fast prototyping.

Instead, we propose to embed component frameworks directly into low-level languages. Having everything in the same language makes it easier to implement changes that bridge abstraction levels, which is useful for low-level optimization and fast prototyping. Moreover, this approach makes component assemblies mere objects, which opens introspection and static optimization possibilities.

We have developed such an embedded component-based framework, called tinycompo, in C++ and used it to build a Bayesian inference library. We find the approach to be particularly suited to Bayesian inference and see software engineering benefits.

This paper starts with a general presentation of software components for readers which are not familiar with the paradigm. Next, we review related works and present our own C++ framework. The framework is then used, in a detailed step-by-step example, to build a high-performance Bayesian inference application. Finally, we present a performance and software engineering evaluation of the approach on real-life bioinformatics applications.

Software Components

Principle

At the end of the 1960s, computer scientists proposed to copy the practices of other industries and to start building software from components produced by third parties.

Components are different from traditional libraries in the sense that they are built to interoperate with components from unrelated third parties. For example, most implementations of the MPI standard – a widely-used message passing technology in high-performance computing – are traditional libraries: they provide a functionality to programmers through an API and can be configured though a library-specific interface. On the contrary, the OpenMPI implementation[3] is component-based and allows the integration of third-party code directly in the library implementation, provided the code in question follows some interoperability rules.

Rules for interoperability between components are called *component models*. Just like in other industries, some degree of standardization is required so that components from different manufacturers can be used together. For example, in the music industry the MIDI standard makes it possible to connect audio and music-related devices from various manufacturers. A component model must define what constitutes a *software* component – that is, the unit of code reuse – and how to use several components together.

Component-Based Software Engineering is the domain that deals with designing and implementing component models. Many component models have been proposed over the years both in the industry – such as COM (Microsoft), EJBs or XPCOM (Mozilla) – and in academia.

Component models in practice

The nature of the components themselves – that is, the unit of reusable code – can take various forms depending on the model. Many modern component models use objects as components. More recently, component models have been developed for cloud applications, in which components can be things like web services or databases.

A central concern in the design of component models is that of *interfaces*. The interface of a component is a declaration of what it does, provided by the component designer. When using a component, a user can assume exactly as much about its behavior as is specified in its interface. Components are said to be black boxes in the sense that they hide implementation details. This is important as it eases *separation of concerns* - that is, how easy it is to work on one part of the code without understanding the rest.

Like electronic components, software components often have *ports*, that is interaction points which can be connected to the ports of other components. Ports can represent things like data input/output, provided/required services, object interfaces, and so on. Just like there are port types in electronics – you cannot connect any port with any other port – possible connections between software components are constrained by port types.

Just like one might use a variety of cables and adapters to connect electronic components, some component models allow non-trivial connections between ports using *connectors*. For example, two ports of incompatible types could be connected with an adapter connector that performs a type conversion. For example, two arrays could be connected by an n-ary “array connection” connector that performs many element-by-element connections. Connectors are a useful tool to encapsulate connection logic, to keep it outside of the components themselves.

In addition to defining what components and their interfaces are, most component models specify how they should be composed to form applications. A composition of components is often called a *component assembly*. Many component models provide dedicated languages to describe component assemblies. Since components often need to be deployed – that is instantiated and/or allocated somewhere – most component models provide automated deployment of component assemblies. This means that a component assembly description plus an entry point – that is, a function to run at startup – is effectively a full application that can be run with no extra code.

In addition to being useful to deploy applications, component assemblies provide a high-level view of application structure. Since they are graph-like in nature, assemblies can easily be represented graphically. Examples of graphical representations of assemblies are given further in the article. Such representations are useful for top-down discussion about application design.

A last commonly-found feature of component models is so-called *composite components*. A composite component is a collection of components that can be seen as a single component, or, to put it the other way around, it is a component that is implemented by a component assembly. Components inside composites can themselves be composites, forming a hierarchy of nested composites. Composites are a useful tool to organize application structure into meaningful sub-parts.

Proposal: an embedded component model

Related works

CCA[4] (for Common Component Architecture) and L2C[5] are two component models specifically designed for scientific computing. CCA focuses on multi-language compatibility (C/C++/Fortran/python), distributed deployment, and some design patterns commonly found in high-performance computing. L2C is based on distributed C++ objects that communicate either locally or using MPI. L2C assemblies are described using XML. Both L2C and CCA are designed to have minimal performance impact and to ease the deployment of large component assemblies on distributed architectures.

Component models from closely-related domains, such as grid or cloud computing, might also be of interest to high-performance programmers. For example, the Grid Component Model (GCM)[6], focuses on distributed deployment and control of components in computing grids.

Other works have investigated applying the component approach to specific application classes. For example, Aumage et al.[7] applied a mix of component-based and task-based paradigms to a large gyrokinetic simulation, and Bigot et al.[8] used component assemblies as the backend of a domain specific language for stencil applications. Another interesting occurrence of components in high-performance computing is the already-mentioned use of a component model for the implementation of OpenMPI.[3]

Although the aforementioned component models have been successfully used in specific applications, adoption in the scientific community at large is limited. These frameworks are multi-layered and use dedicated languages for component interface declaration and component assembly. We first argue that these models are hampered by a high barrier to entry, as they are difficult to use for non-computer scientists and difficult to adapt to existing code. Their multi-language nature leads to complex tool suites and requires handling additional dedicated languages. We also argue that low-level optimization and fast prototyping, which are staples of scientific computing, are hampered by the strict separation of concerns imposed by the multiple layers of existing models.

Our proposed model

Instead, we propose to embed all layers of typical component-based frameworks in a single language. This approach is conceptually and technologically simpler, lowering the barrier to adoption. While a more relaxed separation between the different layers (component programming, declaration and assembly) might weaken separation of concerns, it is also more flexible which can be useful for rapid prototyping. Finally, embedding component assembly into a general-purpose language provides opportunities for introspection and automatic assembly generation.

We have thus devised an embedded component model which implements common component-based features such as composites and connectors. Figure 1 gives a conceptual overview (in the form of a metamodel) of the structure of our component model. A user wiling to build a component assembly must declare a model of the assembly; this model contains components which implement C++ classes and have addresses; components are connected with connectors which are implemented by functors (C++ callable classes) and target addresses; finally, the model can contain composites which have an address and are implemented by an assembly model.

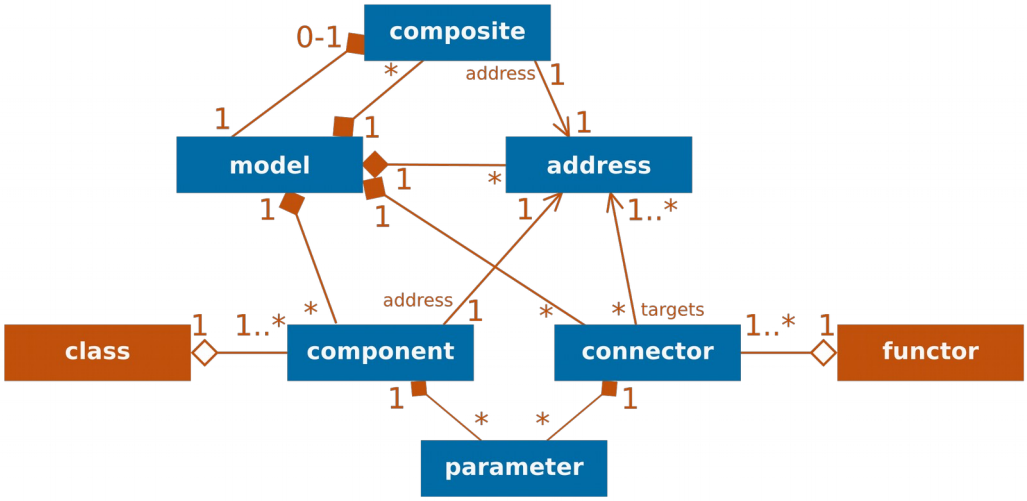

*Figure 1. Metamodel of our embedded component model from the point of view of a user using UML notation. Orange rectangles are user-provided C++ objects while blue rectangles are user-declared component-related entities. Edges denote relationships between entities and are labelled by their multiplicity (for example, a model can be associated to any number of components, denoted by *, but a component is associated to a single model). Diamonds denote whole/part relationships (e.g., components are part of a model), with filled diamonds denoting that the part cannot exist independently of the whole. Arrows denote other directed relationships.*

We have implemented this model in C++ in the form of a framework called tinycompo. The name is inspired by the well-known *tinyxml*, to which it is similar in the sense that it is a small (~2k lines of code) C++ header-only library that one needs only include in their project to use. Instead of having dedicated languages for interface and assembly declaration, these are done directly in C++, leveraging C+11 features to provide a declaration syntax as close as possible to what one might find in a dedicated language.

Particular care was put to minimize the overhead of the model. In practice, *tinycompo* components are normal C++ objects which are either connected directly using pointers or which communicate through MPI. The only computation time overhead brought by the implementation is the systematic use of virtual calls between components, which could only be a problem in very fine-grained applications.

In practice, to use *tinycompo*, one has to:

- write components (which are objects which inherit from tc::Component) and connectors (which are functions that perform connections);
- build a Model object (that follows the structure given in Figure 1) that is a representation of the desired assembly;
- instantiate the assembly by building an Assembly object from the model;
- use the instantiated assembly, for example by calling component methods.

A minimal Hello World looks like this:

```
#include "tinycompo.hpp"

class HelloComponent : public tc::Component {
  public:
    /* declare a port in the constructor */
```

```
    HelloComponent() { port("hello", &HelloComponent::hello); }
    void hello() { std::cout << "Hello world\n"; }
}

int main() {
    tc::Model model; /* declaring an empty assembly model */
    model.component<HelloComponent>("mycompo"); /* adding our hello
component */

    tc::Assembly assembly(model); /* instantiating assembly */
    assembly.call("hello", "mycompo"); /* calling the hello method */
}
```

Having an abstract *Model* object that represents a yet-uninstantiated assembly means that the assembly can be modified before instantiation. For example, static optimization of a user-provided assembly could be performed automatically. This opens possibilities of static analysis and optimization which can help bring existing automation from high-level Bayesian frameworks.

The implementation is also capable of automatically generating graphical representation of *tc::Model* objects in the .dot format, using a representation that is close to the UML notation – a common graphical convention for components.

A step-by-step bioinformatics example

Thanks to New Generation Sequencing techniques, sequencing the genome of an organism – in the form of DNA – has become cheap and efficient. This has opened new research opportunities to understand the role of the genome. Of particular interest in the genome of organisms are *genes* – parts of the genome that code for functional molecules. *Gene expression* – that is, how many molecules are produced from a given gene – can be measured. Patterns of gene expression levels can then be used to locate genes of interest.

Gene expression measurements are, however, imprecise, which makes differentiating noise from unusual patterns difficult. This can be solved by using Bayesian inference to provide estimates of real gene expression levels from noisy observed data. Bayesian Inference is a mathematical technique that consists in computing the probability of hypotheses depending on observed data. Such distributions are called *posterior distributions*; statistical tests can be performed on them to locate genes of interest.

To perform Bayesian inference, one must provide a hypothesis – in the form of a probabilistic model – of how the observed data have been produced. This model can take the form of a Directed Acyclic Graph (DAG) where each node is associated to a probability distribution, and each vertex from A to B denotes that B uses A as a parameter for its associated distribution.

A very simple gene expression model is presented in graphical form in Figure 2. This model is limited to the expression of a single gene across individuals and experiments. Given individuals indexed by i, the real gene expression of individual i is given by λ_i. The value of λ_i may vary from one individual to another, and is assumed to follow a gamma distribution described by its mean α and a dispersion parameter μ. Those two parameters are assumed to have exponential distributions. Several experiments have been performed in which the gene expression $K_{i,j}$ of every

individual has been measured. We expect these discrete measurements to be distributed according to a Poisson distribution of mean λ – which adequately models the measurement noise.

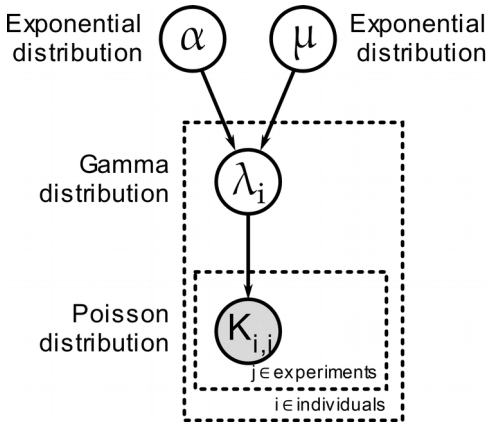

Figure 2. A simple probabilistic model represented by a DAG. Circles are probabilistic variables of the model. The greyed circle corresponds to observed data. The contents of dashed boxes are replicated for every index value.

Given this model, Bayesian inference consists in computing the probability distribution – the so-called posterior distribution – of unobserved nodes (that is, λ, α and μ) given observed data. In practice, the computations should output a set of vectors which collectively form a sample of the posterior distribution.

A widely-used method to obtain this sample is *Markov Chain Monte Carlo* (MCMC). Starting from an arbitrary vector $\theta=(\lambda_0,\lambda_1...\lambda_N,\alpha,\mu)$, MCMC consists in running a Markov chain of θ_k vectors – that is, a chain where every element is computed probabilistically from the previous element. There exist several algorithms to compute from in such a way that the chain samples from the posterior distribution.

One such algorithm is the *Metropolis-Hastings algorithm*, which consists in proposing random changes – called moves – to and accepting them with a probability that depends on the probability of the observed data before and after the move.

Using *tinycompo* as our component model, we have implemented a Bayesian inference library. This section and the next ones present a step-by-step example of how to use such a component-based library to build a simple Bayesian application. While many details are obviously specific to our library, the example illustrates more general design principles and highlights some of the benefits of components.

Basic model structure

From the perspective of the developer of a Bayesian inference application, an important question is the specification of the probabilistic model. In high-level Bayesian inference frameworks, users basically only need to specify their model and ask the framework to run the inference. Graphical representations like the one presented in Figure 2 are familiar to scientists in the domain and are thus a good abstraction to start with.

Assuming we can specify probabilistic models, how do these models intervene in inference? The Metropolis-Hasting algorithms needs to know the probabilistic model in these cases:

- to access the value of every node in order to output the vectors;

- to compute the probability of the model in order to compute acceptance probability when making moves; the probability of every node depends on its associated distribution (the probability density of this distribution is required), on its value, and on the value of its parents;
- to modify the value of nodes to enact accepted moves.

Thus, a good potential data structure to provide all these functionalities would be a graph of "probabilistic node" objects, each associated to a distribution, and each connected to their parents in the model. Each object would provide its value through a "Value" interface, and would provide a "Probability" interface that returns the probability of the node given its value, its associated distribution, and the values of its parents – which it can access through their "Value" interface as it is connected to them.

Our library provides component types that implement probabilistic nodes with various number of parents. These components need to be connected to form a probabilistic model. The library also provides the ability to handle arrays of probabilistic nodes using composite components to represent arrays of components and a collection of connectors to represent many-to-one, one-to-many and many-to-many connections between arrays of varying dimensions.

Declaring our example model using the library looks like this:

```
tc::Model m;
m.component<OrphanExp>("alpha", 1, 1);
m.component<OrphanExp>("mu", 1, 1);

// array lambda of gamma nodes
m.component<Array<Gamma>>("lambda", experiments, 1)
    .connect<ArrayToValue>("a", "alpha")
    .connect<ArrayToValue>("b", "mu");
    // a and b are port names for gamma parameters

// array K of Poisson nodes
m.component<Matrix<Poisson>>("K", experiments, samples, 0)
    .connect<MatrixLinesToValueArray>("a", "lambda")
    .connect<SetMatrix<int>>("x", data);
```

Note the use of the *.connect<C>* method to declare a connection between components using connector C. Our library provides a set of connectors such as *ArrayToValue* which connects components using pointers to interfaces. For example, all components in the "lambda" array will obtain a pointer to the *Value<double>* interface of component "alpha"; this pointer will be stored in an attribute of the "lambda" components associated to port "a".

Although the syntax to declare components is not the simplest due to the constraints of the C++ language, only minimal information is required, and the structure resembles that of existing assembly declaration languages.

Metropolis-Hastings moves

Now that we have a data structure that represents our probabilistic model, we need to add the Metropolis-Hastings algorithm itself. The basic algorithm is as follows:

1. initialize observed nodes from actual data and unobserved nodes arbitrarily;
2. propose a move, *i.e.*, a random change to unobserved node values;

3. compute acceptance probability using model probability before and after proposal;
4. decide to accept or reject move using acceptance probability; change the value of nodes according to proposal only if the move is accepted;
5. start over at step 2.

In practice, not all unobserved nodes are moved at the same time. It is often more efficient to move a few parameters at a time and cycle between moves that change different parts of the model. For example, in our example model, one move could change the value of only α, another the value of μ, and so on. As long as every unobserved node is covered by at least one move, the algorithm will converge to the correct distribution.

In this context, writing the vector $\theta_k = (\lambda_0, \lambda_1 \ldots \lambda_N, \alpha, \mu)$ after every move is not useful. Instead, the vector should be written to disk after every unobserved node has been the target of a move at least once. We call the set of moves that are performed between two outputs an *iteration*.

To add Metropolis-Hastings moves to our implementation, we need a way to represent them with components. A component implementing a move would need to have the following interface:

- have a target, an unobserved node in the model that it will try to move (implemented by a "target" port);
- have access to all the nodes in the model whose probability might change when the target's value is changed ("logprobs" port);
- provide a way to start the move from outside the component.

Using such a component to add a move on would look like this (assuming we have already declared our probabilistic model data structure):

```
m.component<MHMove<Scale>>("move_alpha")
    .connect<UseValue>("target", "alpha")
    .connect<Use<LogProb>>("logprobs", "alpha")
    .connect<OneToMany<Use<LogProb>>>("logprobs", "lambda");
```

While the move component itself neatly encapsulates move logic, its connection to the probabilistic model data structure is non-trivial. In particular, the "logprobs" port must be connected to the correct parts of the model data structure, the ones whose probability is affected by the move.

Now that moves have been added to the assembly, everything is in place to run the actual Metropolis-Hastings algorithm. The only thing missing is a *main* function that loops over iterations and calls the moves.

Automation using model introspection

Declaring moves and writing the main MCMC loop as presented in the previous sections could easily become verbose in more complex examples. Existing Bayesian inference frameworks usually try to automate such tasks instead of leaving them to the user. To do this, they rely on representations of the probabilistic model, and run various algorithms on these representations.

While our library does not have a "graphical model" object per se, the *tc::Model* object representing the assembly provides introspection methods which can be used to gain insight on the graphical model. For example, it is possible to get a Directed Acyclic Graph (DAG) of the components and binary connections; it is also possible to test the type of a given component. Contrary to existing HPC component models, these introspection functionalities are directly accessible in the C++ code as well as methods to modify the assembly.

For example, we have automated the connection of moves to the correct logprob ports in the assembly. The set of nodes affected by a given move – which is called *Markov blanket* in the literature – is easy to compute given a set of modified nodes in a probabilistic model. This algorithm needs only the DAG of probabilistic nodes of the model, which can be obtained with the introspection methods.

We have implemented this Markov blanket calculation in our library in a MCMC object which provides a declaration interface for moves and automatically adds them to the assembly with optimal logprob connections. Because this MCMC object is aware of the list of moves, it can provide other functionalities such as configuring the trace to show only variables which are affected by moves and running the main loop. We have also implemented an optimization from the literature called *sufficient statistics* which consists in using proxies for the logprob calculation of large arrays of probabilistic nodes. Moves affected by sufficient statistics should be connected to relevant sufficient statistics objects instead of directly connected to the logprob of the probabilistic nodes; this is another task which can be automated by our MCMC object.

Using this new object, the move declarations and main loop section of our example becames as simple as:

```
MCMC mcmc(m, "model"); /* declaring MCMC object */
/* giving it the tc::Model and the address of the graphical model */

mcmc.move("alpha", scale); /* declaring moves */
mcmc.move("mu", scale);
mcmc.move("lambda", scale);
mcmc.suffstat("lambda", {"alpha", "mu"}, gamma_ss); /* sufficient
statistic */
mcmc.declare_moves(); /* add all the moves to the component assembly */

mcmc.go(50000, 10); /* main MCMC loop with 50000 iterations */
```

Distributed version with MPI

The Metropolis-Hastings algorithm can become computationally intensive for large model sizes. While our example model may seem simple at first glance, its size is actually ~O(#individuals x #experiments), which may be significant for large datasets.

The Metropolis-Hastings algorithm is intrinsically sequential in the sense that an iteration cannot be started before the previous one has finished lest the convergence of the algorithm be compromised. It is, however, possible to perform certain moves in parallel inside iterations. Indeed, moves with disjoint Markov blankets are functionally independent.

With our example model, moves on individual λ_i nodes are independent of each other and can be performed in parallel, as opposed to moves on α and μ. A possible parallelization of the Metropolis-Hastings algorithm is as follows:

1. Have a “master” process which performs moves α and μ while the λ_i nodes remain untouched.
6. Divide the set of individuals into subsets and have a “slave” process for each subset. Once the master has finished moving α and μ, each slave can perform moves on the λ_i in its subset of individuals while the master waits.

7. Once the slaves have finished performing moves on the nodes, the iteration ends, the vector is written to disk and the next iteration can start with the master.

One way to implement parallel algorithms in a component-based context is to have different assemblies on different processes. In our case, each process needs only access to a subset of the full model – for example, the master does not need access to the nodes $K_{i,j}$ – and could thus hold only the relevant part of the model to save memory. Given the components we have so far, the only missing part is a way to have processes wait on each other and exchange data.

The question of synchronization and communication can be answered by having Bcast and Gather component which transmit data from the master to slaves and from slaves to the master respectively. These components would be connected to the same target nodes on both sides – on one side to read the values and on the other side to set it. They would need to provide an acquire interface to wait for the other processes that modifies their targets and gets their value, and a release interface to signify that they no longer need to modify their targets and that other processes can modify them again. Since data is transmitted between one master and N slaves, collective communications – such as *MPI_Gatherv* and *MPI_Bcast* – can be used for maximum performance and encapsulated in the components.

In practice, to get a parallel version of our MCMC example, the following changes are needed: 1) add an if in the model declaration to not declare K on the master; 2) partition individuals between slaves; 3) declare moves α and μ only on the master and moves on λ_i only on slaves; 4) declare communication components on all processes as follows:

```
m.component<Bcast>("alpha_mu_proxy")
    .connect<UseValue>("target", Address("model", "alpha"))
    .connect<UseValue>("target", Address("model", "mu"));

m.component<Gather>("lambda_proxy", experiment_partition)
    .connect<OneToMany<UseValue>>("target", Address("model", "lambda"));
```

And 5) on both slaves and master, call the acquire method of proxies before computing and the release method afterwards – this can be accomplished automatically with a modified MCMC class for MPI (which we have implemented and is called *MpiMCMC*). Note that proxy declaration is the same on master and slaves, *Bcast* and *Gather* hide the master/slave difference inside their implementation; this global declaration of proxies is consistent with how global operations are usually declared in MPI.

Overall, these modifications are fairly small, which makes parallelization of Bayesian applications using our library very easy.

EVALUATION

To evaluate our library, we have implemented a real-life probabilistic model instead of the simplistic model presented in the step-by-step example. This model aims to detect genes with specific expression patterns. We have implemented this model (called M3) with our library, and made a parallel version on the same model as the one presented in the previous section.

This use case was motivating for this work, as it posed both programmability and performance issues. The size of the dataset was large enough that a high-performance implementation would be needed, but custom probabilistic models would need to be developed and iterated upon.

To evaluate our component-based implementation, we compared it to JAGS[9], a widely-used program for Bayesian inference. JAGS has a large community built around it and is widely recognized and used in various applied science communities. JAGS provides an easy-to-use dedicated language to declare graphical models, and implements many MCMC optimizations which give it good sequential performance.

Expressivity

The model we have presented in the step-by-step example (see Figure 2) can be written using the JAGS dedicated language. This dedicated language is used to declare the model itself and must be used through a library in a general-purpose language, typically R.

To compare the expressivity of our approach to JAGS, we have implemented models M1 and M2 which are simpler preliminary versions of the M3 model. Having a collection of models allows us to see that our library is flexible enough to accommodate different models.

Table 1 gives the size of these implementations for JAGS (in R using rjags) and *tinycompo* (in C++). The sizes are given in lines of code, as computed by the cloc program. In addition to M1-M3, we added M0, the model we used for the step-by-step example. We see that, overall, code sizes are comparable. Both implementations use the same underlying abstraction (graphical models) to declare the models themselves. The main change with our library – apart from the added complexity of C++ syntax – is that the MCMC moves must be declared, but we see that the impact on overall code size is minimal.

We see that parallel version of M0 and M3 are not much longer than their non-parallel version, which is an important advantage of our approach.

Table 1. Code size (lines of code) comparison between JAGS and tinycompo.

		M0	**M0-MPI**	**M1**	**M2**	**M3**	**M3-MPI**
Total code	tinycompo	32	43	31	47	61	86
	JAGS	*	*	48	55	61	*
Model only	tinycompo	13	15	10	23	34	44
	JAGS	10	*	13	19	24	*
*: not implemented							

Sequential performance

To evaluate sequential performance, we once again compare our library to JAGS. While JAGS uses different algorithms than we do, it still performs MCMC and produces the same output: a trace which is a sample of the posterior distributions of interest. For a given model, the traces produced by both implementations can be compared using Effective Sample Size (ESS), which is a measure of how much information a trace contains about the posterior distribution of a variable. Two traces with comparable ESS are thus equally useful for a MCMC end user.

For each of the models M1-M3, we have run JAGS and our implementations on a small subset of real data for enough iterations that the chains have reached their stationary distribution. Then we

have computed the mean ESS for N iterations of our implementation, and adjusted N until the mean ESS was higher than for 2000 iterations of JAGS.

Results are that our M3 implementation needs to compute 6500 iterations to have as much information as 2000 JAGS iterations, which it does in 10.458s while JAGS needs 2.337s to compute 2000 iterations. Given that our implementation uses a much simpler algorithm, the fact that its overall speed is not further away from JAGS is indicative of a reasonably efficient implementation. Also, the implementations for M1-M3 use default tunings for the MCMC moves; a better-tuned implementation could probably give much better results.

Parallel performance

So far, we have seen that our approach is slightly worse than JAGS in terms of expressivity and sequential performance, but contrary to JAGS, our code is easily parallelizable.

To evaluate the scalability of our parallel implementation, we have performed two experiments on the tier-0 supercomputer Occigen using our parallel M3 implementation. A first experiment was conducted on a real dataset (with 7124 genes) with an increasing number of 24-core computing nodes, ranging from 4 nodes (96 cores) to 128 nodes (3072 cores). For each node count, the genes were divided as equally as possible between the slave processes. This so-called strong scaling experiment shows how much computation can be sped up for a fixed dataset by adding more resources. A second experiment was conducted with a large simulated dataset. Contrary to the first experiment, each process is given a fixed amount of data (8 genes), which means the size of the dataset grows with the number of processes. This so-called weak scaling experiment shows how well the code can handle datasets of increasing size given more resources.

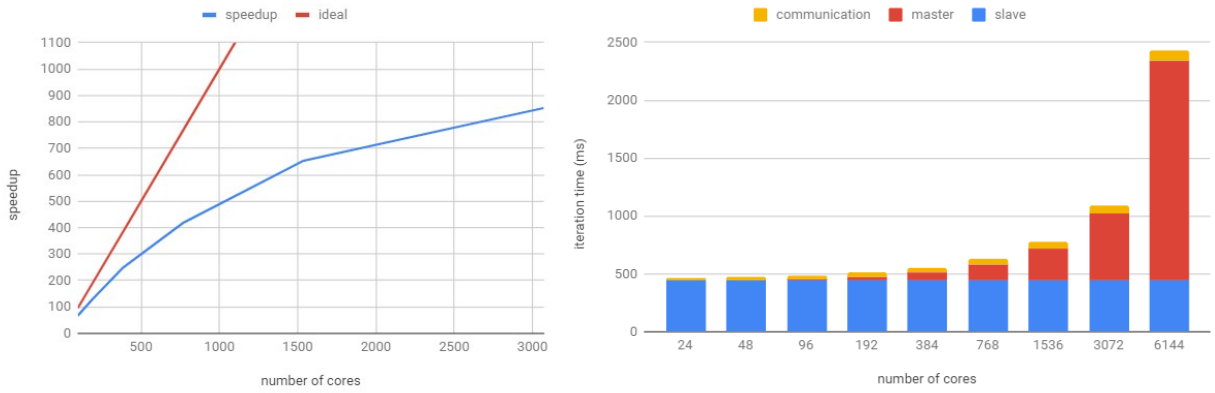

Figure 3. Strong scaling (left) and weak scaling (right) results on the Occigen supercomputer. All computing nodes have two 12-core Intel E5-2690V3 processors at 2.6GHz with 64Go of memory; network uses InfiniBand. Strong scaling consists in runs of the M3 model parallel implementation on real data with 7124 genes and 87 samples. Speedup is computed by dividing a projected sequential iteration time (time measured on 4 genes with a single slave multiplied by 7124/4) by the measured iteration time. Weak scaling consists in runs of the M3 model parallel implementation on simulated data with 8 genes per core. Iteration time is divided into the time spent on slaves (in blue), the time spent on the master (in red) and the different between the total iteration time and the sum of master and slave times (in orange) which is interpreted as synchronization/communication time.

Results are given in Figure 3. Strong scaling results show that speedup keeps improving even at 3072 cores, although the rate of speedup improvement decreases with higher node counts. This decrease is to be expected because the parallelization relies on global communication (broadcasts and gathers), because worker/slave load ratio increases, and because load balancing quality

decreases as the number of genes per core decreases. Given all these scalability problems, the observed speedup is rather encouraging. The maximum observed speedup (at 3072 cores) is around 850, which means that our parallel code is approximately 850 times faster with this number of cores that it would be on a single core. Even considering that JAGS sequential performance is better, our parallel code can be several hundred times faster on a large enough dataset. Weak scaling experiments show that scalability is good up to 3072 cores but starts to drop at 6144 cores. Communication and synchronization do not seem to be a problem, but the time spent on the master is the main bottleneck. This is because the master performs operations which are in $O(\#genes)$ which are inherently non-scalable. Still, the performance at 3072 cores means that analysing a dataset of size $3072 * N$ with 3072 cores takes only twice the time it would take to analyse a dataset of size N on one core.

Conclusion

In this article, we have presented the *tinycompo* component model, which is novel because it is embedded in the C++ language. We have presented a detailed step-by-step example of how this model can be used to implement high-performance MCMC applications and have evaluated our implementation both in terms of performance and expressivity. Evaluation show that our approach is about as expressive as JAGS, has worse sequential performance, but is able to scale up to several thousands of cores.

We have presented two technologies that we have developed. Our component model *tinycompo* is available on github (http://github.com/vlanore/tinycompo). Our Bayesian inference library is also available on github (http://github.com/vlanore/compoGM). The full code for our example is available on the repository, as well as the more complex codes of the real-life model that we used for performance comparisons.

Our library and *tinycompo* are currently used in several projects inside the lab. Component-based approaches in general (systematic use of composition, explicit structure) are starting to gain traction inside the team, as people start to see the software engineering benefits. In addition, a presentation of our work on *compoGM* was given to the AppliBUGS community (BUGS/JAGS users in France) and got positive feedback; people in this community are very familiar with the graphical model representation and would be interested in parallelization if it was easy to do.

Perspectives for this work include the use of embedded component to refactor legacy application. The extra flexibility offered could help address this issue which has been cited as being a major obstacle to component use in HPC[2]. There is also ongoing work to improve the performance of arrays of components in *tinycompo*, which poses difficult design questions.

Acknowledgments

Thanks to Philippe Veber for providing the JAGS reference implementations, the script to simulate data and the script to compute ESS as well as for his help with the manuscript. Thanks to Bastien Boussau for his help with the manuscript.

This work was funded by: ANR-15-CE32-0005 “Convergenomix”.

This work was granted access to the HPC resources of CINES under the allocation A0040310449 made by GENCI.

References

1. Darriba, Diego, Tomáš Flouri, and Alexandros Stamatakis. "The state of software for evolutionary biology." *Molecular biology and evolution* 35.5 (2018): 1037-1046.
2. Basili, Victor R., et al. "Understanding the high-performance-computing community." *IEEE software* 25.4 (2008): p29-36.
3. Squyres, Jeffrey M., and Andrew Lumsdaine. "The component architecture of open MPI: Enabling third-party collective algorithms." *Component Models and Systems for Grid Applications*. Springer, Boston, MA, 2005. 167-185.
4. Armstrong, Rob, et al. "The CCA component model for high-performance scientific computing." *Concurrency and Computation: Practice and Experience* 18.2 (2006): 215-229.
5. Bigot, Julien, et al. "A low level component model easing performance portability of HPC applications." *Computing* 96.12 (2014): 1115-1130.
6. Baude, Françoise, et al. "GCM: a grid extension to Fractal for autonomous distributed components." *Annals of Telecommunications-annales des télécommunications* 64.1-2 (2009): 5-24.
7. Aumage, Olivier, et al. "Combining both a component model and a task-based model for hpc applications: a feasibility study on gysela." Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on. IEEE, 2017.
8. Bigot, Julien, Hélène Coullon, and Christian Pérez. "From DSL to HPC component-based runtime: a multi-stencil DSL case study." Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing. ACM, 2015.
9. Plummer, Martyn. "JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling." *Proceedings of the 3rd international workshop on distributed statistical computing*. Vol. 124. No. 125.10. 2003.

About the author

Vincent Lanore received his PhD in computer science at ENS de Lyon in 2015 on the topic of component models for HPC. He has been a temporary researcher at CNRS since 2016, located at the LBBE laboratory in the context of the Convergenomix bioinformatics project. His research interests include software engineering for scientific computing, bioinformatics, and Bayesian inference. Contact at vincent.lanore@univ-lyon1.fr.