From Conditional Independence to Parallel Execution in Hierarchical Models

Balazs Nemeth[1(✉)], Tom Haber[1,2], Jori Liesenborgs[1], and Wim Lamotte[1]

[1] Hasselt University - tUL, Expertise Centre for Digital Media, Martelarenlaan 42, 3500 Hasselt, Belgium
{balazs.nemeth,tom.haber,wim.lamotte}@uhasselt.be
[2] Exascience Lab, Imec, Kapeldreef 75, 3001 Leuven, Belgium

Abstract. Hierarchical models describe phenomena by grouping data into multiple levels. Due to the size of these models, parallel execution is required to avoid prohibitively long computing time. While it is occasionally possible to specify some of these models using parallel building blocks, this limits expressivity. Therefore, a more general generative specification is preferred. To leverage parallel computing capacity, these specifications can be annotated, but doing so effectively assumes that the modeler has expertise from computer science. This paper outlines how to identify parallel parts automatically by leveraging the conditional independence property in the graphical model extracted from the dataflow graph of model specifications. Computation related to random variables with the same depth in the graphical model are identified as candidates for parallel execution. Since subsequent proposals in the parameter space exploration of the model are clustered together, the results show that the well known longest processing time scheduling heuristic deals adequately with load imbalance. The proposed parallelization is evaluated on two pharmacometrics models, a domain where hierarchical models with load imbalance are common due to the numeric simulation of pharmacokinetics and pharmacodynamics of human subjects. The varying number of measurements taken per subject further exacerbates load imbalance.

Keywords: High performance computing · Descriptive language · Probabilistic modelling · Automatic parallelization · Dataflow · Hierarchical models

1 Introduction

In recent years, the physical limits that would otherwise prohibit Moore's Law's predicted performance increase have been circumvented by the trend towards more parallel systems [7]. The flip-side of this explicit form of parallelism is that it puts more of the burden on the software developers, or even the users. It is far from straightforward to leverage all the compute power available in parallel systems [10,13], but the complexity of the models precludes fitting a model on a single processor since it is too time intensive in practice. In the context of computational modeling, there are two prominent strategies for parallelization.

© Springer Nature Switzerland AG 2020
V. V. Krzhizhanovskaya et al. (Eds.): ICCS 2020, LNCS 12137, pp. 161–174, 2020.
https://doi.org/10.1007/978-3-030-50371-0_12

First, in some cases, as models are fit using an iterative optimization routine, multiple processors can be kept busy within each iteration. Computation fits into the Bulk Synchronous Parallel (BSP) model of parallel computing [16] with multiple candidate parameters evaluated concurrently. While this approach allows hiding the parallel constructs within the routine, improving the usability of these routines for scientists from other domains requires the optimization routine to be designed to run in parallel, which might not be feasible.

Second, depending on the model, a single candidate parameter can be evaluated in parallel. This strategy is suitable both for more sequential optimization routines as well as parallel optimization routines where it can further improve performance. Even if a task can be decomposed into smaller concurrently executable tasks, doing so manually is tedious and error-prone even when armed with the right parallel computing background. Arguably, the scientists concerned with building these models are in an even worse position; their expertise is probably not in parallel computing and a more automated approach, like the one explored here, is preferable.

The focus here is on the parallelization of hierarchical models composed of multiple interconnected levels. Computational tasks required for each model evaluation are typically spread across relatively few layers. Consequently, this brings with it the opportunity to execute each level in parallel. While it might not be the optimal parallelization, it turns out that it performs well in practice. It can even be used in conjunction with other methods that search for more fine-grained parallelism [20]. The main contribution is to show how to extract the graphical model representation from the dataflow graph of the model and how to map parallelism from the former to the latter.

When the number of tasks exceeds the number of processors in a layer, some processors will inevitably execute more than one task. Depending on the variability of execution times between these tasks and the ratio between the number of tasks and processors, neglecting the scheduling problem can result in inefficient use of the underlying hardware. The parallelization approach is augmented with the well-known Longest Processing Time (LPT) static scheduling heuristic [9], where independent jobs with varying execution time are scheduled on p identical processors.

The reachable efficiency is model-dependent; in general, the more compute-intensive tasks are available at each level of the hierarchy, the better performance will scale. Therefore, two different models are considered for evaluation: one containing only a few tasks and another with many more compute-intensive tasks. While parallelization adds overhead introduced by inter-processor communication, overall run time decreases in both cases.

The remainder of this paper is structured as follows. Section 2 references related work. Section 3 discusses hierarchical models, their structure in the dataflow graph representation and the relationship with conditional independence. Section 4 describes the parallelization approach. Section 5 discusses performance results. Section 6 provides future work directions and concludes the paper.

2 Related Work

The input to the optimization routines or sampling algorithms is a function that evaluates a model and returns a score that reflects the quality of the parameters. In this paper, the input is a model description specified similarly to the probabilistic languages used in Turing [11], Stan [5] and WinBUGS [19].

The Turing system [11] relies on explicit vectorization syntax to gain performance. The presented approach relies on the message passing model [16] for parallelism and vectorization is an extension that is left as future work.

Stan [5] is a platform for statistical modeling and high-performance statistical computation. Recently, an extension to its modeling language has been proposed for parallelization [22], but use requires changing the model description. In contrast, the parallelization outlined below does not require the user to specify additional input signifying how computation should be scheduled on the hardware, but the downside is that it can be too aggressive causing performance to degrade in some cases.

Gibbs sampling [6] draws samples from the marginal target distribution by combining samples taken from conditional distributions. The concept of a graphical model is fundamental for Bayesian inference Using Gibbs Sampling (BUGS), implemented in WinBUGS [19]. MultiBUGS [12] has added parallel execution to WinBUGS by working directly on the graphical model from which conditionally independent parts are identified and scheduled to parallel processors only when deemed beneficial by a heuristic. Execution of Gibbs Sampling requires synchronization between phases more closely resembling the BSP model. The difference with the work presented below is that the graphical model is used indirectly to detect parallel parts of the dataflow graph. Since the posterior is evaluated as a whole with less synchronization instead of being separated into smaller conditional densities, the applicability is not limited to Gibbs sampling. Another difference is that MultiBUGS ignores load imbalance by explicitly assuming that tasks have the same running time.

Even if the outlined approach is applied in a Gibbs setting, the parallelization within a single phase is different. For example, given a posterior $p(\theta|\mathcal{D})$, if $p(\theta_i|\ldots)$ and $p(\theta_j|\ldots)$ are assigned to one Gibbs phase, computation shared between these two conditional distributions can be executed only once even without blocking, a technique that affects convergence properties of Gibbs sampling [25].

Nemeth et al. [20] uses an Evolutionary Algorithm (EA) to parallelize the evaluation of probabilistic models by optimizing schedules through simulation of a parallel system with communication overhead. The downside is that searching for a schedule can become prohibitively slow, even though, at least in theory, the optimal schedule could be found. In contrast, using the graphical model is a simpler strategy as only tasks assigned to phases can be executed in parallel. However, it turns out that such an approach already yields well-performing schedules. Another difference is that the EA approach yields a static schedule in which both the execution order and the assignment of tasks to processors are

fixed while the tasks that have been identified from the graphical model can be re-assigned depending on load imbalance changes.

An extensive survey for the well researched task graph scheduling problem is provided by Yu-Kwong et al. [17]. The main difference with conventional scheduling approaches is that the target domain is rather specific. The dataflow graph of a generative model specification always obeys a specific template. From this observation, a mapping can be formulated from which the parallelism is extracted directly.

3 Hierarchical Models and Conditional Independence

The main goal of this paper is to show how model descriptions can be parallelized by relying on information from the graphical model. This section introduces the notion of a model description, its dataflow graph, and its graphical model. To distinguish between the structure of the two representations, "layers" refers to candidates for parallelism in the former and "levels" refers to the depth of variables in the latter.

From a Bayesian perspective [23], a model description defines a posterior $p(\theta|\mathcal{D})$. The numeric value of the posterior determines the quality of a chosen set of parameters θ while taking into account evidence $\mathcal{D}$. In what follows, θ_i denotes a component of the θ vector and $y_i \in \mathcal{D}$ denotes a data entry.

The description consists of likelihood expressions $y_i \sim p(.|\text{pa}(y_i))$ and prior expressions of the form $\theta_i \sim p(.|\text{pa}(\theta_i))$ where $\text{pa}(.)$ is the set of random variables conditioned upon. These expressions will be generalized to $\gamma_i \sim p(.|\text{pa}(\gamma_i))$ for convenience. As an example, consider the model shown by Fig. 1 on the left describing both pharmacokinetics (PK) and pharmacodynamics (PD) of a drug for type-2 diabetes treatment [24].

To convert a model description into an executable function $f(\theta, \mathcal{D})$, prior and likelihood expressions are replaced by probability density function evaluations of the density $p(.|\ldots)$ at γ_i, denoted by a call to `pdf()` to which the distribution and the position are passed. Finally, the product of the resulting probability densities is returned while the remaining expressions are left untouched. The resulting function is then converted into a dataflow graph [1,8]. In contrast to the typical controlflow style reasoning, a dataflow graph is an alternative model of computation where instead of executing operations on data, data flows through operators. This representation of computation lends itself well to parallelization [16]. The dataflow graph $G = (V, E)$ represents the set of computational tasks V and specifies how data flows between the tasks with edges E.

In general, the dataflow graph of a function $f(\theta, \mathcal{D})$ for a hierarchical model has the structure shown in Fig. 2. The inputs θ and $\mathcal{D}$ are shown at the top and the product over densities is shown at the bottom. These are connected with the central portion of the graph, shown by dashed lines. Considering only the part with solid lines, the relationship with the graphical model is revealed. Each level depends on any of the previous levels through density evaluation nodes in V. In the example shown, the connections are less dense; for example, the

```
for i in 1,...,N
  φ_{i,1} ~ Lognormal(μ2, σ1)
  φ_{i,2} ~ Lognormal(μ5, σ2)
  p = h1(μ1, μ4, μ5, φ_{i,1}, φ_{i,2}, pk_i)
  iv = [0.0, 0.0, 0.0, h2(μ)]
  ŷ = int_ode(t_i, 0, iv, dose_i, p)
  for j in 1,...,n_i
    sdv = h3(ŷ_j)
    y_{i,j} ~ N(sdv, σ)
  end
end
```

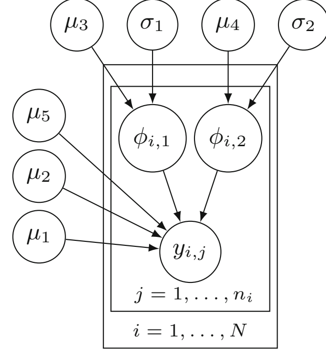

Fig. 1. The Canagliflozin model on the left and its graphical model on the right. A PK/PD model, used to describe the compound concentration over time for N individuals in a population, is numerically integrated by `int_ode`. The number of measurements for the ith individual is given by n_i, and `h1`, `h2` and `h3` are helper functions.

first level is only connected to the second and fourth level and not to the third level. However, it is easy to see how the structure generalizes to any hierarchical model.

The model from Fig. 1 is even less dense. Part of the first layer, μ_3, σ_1, μ_4 and σ_2 are connected with the second layer with variables $\phi_{i,1}$ and $\phi_{i,2}$ and all variables in the second layer together with the remaining part of the first layer are connected with the third layer with variables $y_{i,j}$.

One simplification made here is that an edge in Fig. 2 can represent a sequence of operations that transform random variables between layers or parts of layers like `h1`, `h2`, `h3` and `int_ode` in Fig. 1. It is important to keep this in mind for the discussion in Sect. 4.

A graphical model $H = (R, F)$, is a representation of the conditional independence between variables. Figure 1 shows the graphical model on the right for the Canagliflozin model. For brevity, it is conventional to summarize similar variables with the plate notation by placing them into boxes with the range of iterated indices specified at the bottom [12]. For hierarchical models, H is a Directed Acyclic Graph (DAG), where the set of nodes R represents the random variables in the hierarchical model and their priors, and the edges $F \subseteq R \times R$ denote how the posterior can be factorized, i.e. $p(\theta|\mathcal{D}) \propto p(\theta, \mathcal{D}) = p(\gamma) = \prod_i p(\gamma_i|\text{pa}(\gamma_i))$. An edge from γ_j to γ_i is placed in F if $\gamma_j \in \text{pa}(\gamma_i)$.

To convert a dataflow graph G into a graphical model H, the nodes R and edges F need to be defined in terms of V and E. All nodes with input parameters in G, i.e. θ_i and y_i at the top of Fig. 2, form R. The edges F are defined by the density evaluation nodes. By traversing the edges in E in the opposite direction starting at the node that provides the density input, the variables $\text{pa}(\gamma_i)$ can be found. Similarly, following the other input, γ_i can be found. This mapping introduces a function m from R to V where $m(r)$ is either the corresponding probability evaluation node if it exists, or the input node of that variable.

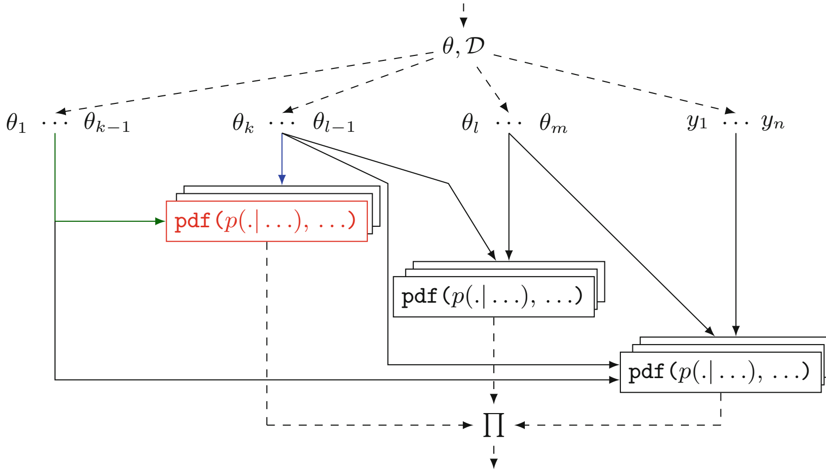

Fig. 2. Simplification of the structure of the dataflow graph of a function $f(\theta, \mathcal{D})$ built from a hierarchical model with four layers the last of which is the data layer. The structure generalizes to any generative specification of a hierarchical models with arbitrary interconnected layers. (Color figure online)

The set $\text{pa}(\gamma_i)$ and γ_i for the red node in Fig. 2 can be found by following the green edges and the blue edges respectively.

4 Extracting Parallelism from the Graphical Model

Since the dataflow representation naturally exposes parallelism in a model it is possible to execute the dataflow graph by starting execution of each node when all its inputs become available. It is well known that scheduling such a computational DAG is hard to solve optimally, but many heuristics exist [3].

Scheduling each node separately is prohibitively expensive in practice due to the amount of overhead introduced on a contemporary system; not only is overhead introduced by starting a function compiled from the expression in a node, but also by tracking and storing its inputs and outputs. To reduce overhead, sets of nodes can be grouped into larger tasks and treated as a single unit at the cost of potentially reducing parallelism. It is possible to find satisfactory assignments of tasks to processors by considering the dataflow graph and the characteristics of the underlying parallel system directly [20], but this search can be slow in practice, especially with graphs that have on the order of 10^4 nodes or more. Such graphs are not uncommon for typical pharmacometrics (PMX) models. Assuming that the order of tasks is not fixed, there are n^p possible assignments to consider with p processors. Parallelization based on the graphical model is more tractable, although less detailed.

Since the posterior can be seen as a product of conditional densities as discussed in Sect. 3, the most basic approach is to create one task for each conditional density from the dataflow graph. This is accomplished by traversing the dataflow graph backwards from each node c that contains the expression $\texttt{pdf}(p(.|\ldots), \gamma_i)$ and selecting all reachable nodes, denoted by the set $\text{pred}(c)$. The procedure $\text{pred}(c)$ extends naturally to sets of random variables as well.

While this leads to an embarrassingly parallel solution since each task can be computed independently, the downside is that many nodes will be recomputed due to the similarities. More formally, for two density evaluation nodes c_1 and c_2, $\text{pred}(c_1) \cap \text{pred}(c_2) \neq \emptyset$. For example, suppose that one of the input parameters γ_i is first transformed to $g(\gamma_i)$ and there are multiple γ_j such that $\gamma_i \in \text{pa}(\gamma_j)$. Then, to compute each conditional density $p(\gamma_j|g(\gamma_i), \ldots)$, $g(\gamma_i)$ will need to be recomputed for every conditional density, a situation that is undesirable if a significant portion of computation effort is spent in g.

Therefore, this paper proposes to use the conditional independence to find similarities between conditional densities. Computationally, conditionally independence of two random variables γ_i and γ_j given γ_k means not only that $p(\gamma_i, \gamma_j|\gamma_k) = p(\gamma_i|\gamma_k) \cdot p(\gamma_j|\gamma_k)$ holds, but also that some of the tasks related to γ_i and γ_j are to be executed after some of the tasks related to γ_k. In addition, there might be some similarity between the computation related to γ_i and γ_j, but there will also be some differences. If this was not the case, then $p(\gamma_i|\gamma_k) = p(\gamma_j|\gamma_k)$.

Computational similarities can be captured by introducing deterministic variables β so that $p(\gamma_i|\beta, \gamma_k) = p(\gamma_i|\beta)$ and $p(\gamma_j|\beta, \gamma_k) = p(\gamma_j|\beta)$. Probabilistically, after marginalizing the deterministic variables β, Eq. (1) holds.

$$p(\gamma_i, \gamma_j|\gamma_k) = p(\gamma_i|\beta) \cdot p(\gamma_j|\beta) \cdot p(\beta|\gamma_k) \tag{1}$$

Once $p(\beta|\gamma_k)$ has been computed, both $p(\gamma_i|\beta)$ and $p(\gamma_j|\beta)$ can be computed sharing as little information as possible. If no information is shared, they can be computed in parallel. Algorithm 1 shows how to accomplish this by processing random variables in the graphical model.

The end goal is to assign random variables to layers and to construct tasks from the variables in these layers. The assumption is that tasks constructed from a layer are independent. In the extreme, when a deterministic variable is introduced for each node in the dataflow graph, all tasks will be independent given their predecessors. Note however that Algorithm 1 introduces only a limited number of deterministic variables. Therefore, the predecessor relationship imposed by E will still need to be respected since there might still be some dependencies. The rationale behind this is that variables in the same layer tend to share computation through their connection with previous layers, on a layer by layer basis, while little or no computation is shared within a layer.

First, following the depth definition from MultiBUGS [12], the level $d(r)$ is computed for $r \in R$. If $\text{pa}(r) = \emptyset$, then $d(r) = 1$. Otherwise, $d(r) = 1 + \max_{p \in \text{pa}(r)} d(p)$. In Fig. 1 the depth is 1 for all μ and σ variables, 2 for all ϕ

Algorithm 1. Extracting layers to construct parallel tasks.

procedure ExtractLayers(G, H, m) ▷ G, H and m defined in Section 3
  Compute $d(r)$ for $r \in R$ as in MultiBUGS [12]
  for $i = 1, \ldots, D$ **do**
    $L_{i,i} = \{\{m(r)\} | r \in R_i\}$
    $P_i = \cup_{r \in R_i} \text{pa}(r)$ ▷ All direct parents of level i
    for $j = 1, \ldots, i-1$ **do**
      $P_{i,j} = P_i \cap R_j$ ▷ Direct parents in level $j < i$
      $L_{i,j} = \{\text{lcpred}(\text{ch}(p) \cap R_i) | p \in P_{i,j}\}$ ▷ Find computational similarities
    end for
  end for
  return $L_{1,1} \ldots, L_{D,D}$
end procedure

variables, and 3 for the data variables $y_{i,j}$. The levels of the random variables partition R into sets $R_1, \ldots, R_D$. Here, R_i contains all the random variables at level i.

It might seem that D layers can now be constructed, one for each set of variables R_i. However, this does not expose computational similarities present between layers. Instead, multiple layers will be introduced for each level i, represented by $L_{i,j}$. The elements of layer $L_{i,j}$ are sets of dataflow graph nodes.

For a level i, the deepest layer $L_{i,i}$ contains the dataflow graph node associated with the random variables $r \in R_i$ as singletons. Next, the directly reachable parents of the variables in R_i are collected in P_i. For each $j < i$, $|L_{i,j}| = |P_{i,j}|$ where $P_{i,j} \subseteq P_i$ are the direct parents on level j. Each element of $L_{i,j}$ is given by the last common predecessors of the children of $p \in P_{i,j}$ in R_j, denoted by $\text{lcpred}(\text{ch}(p) \cap R_j)$. For a set of nodes $S \subseteq R$, $\text{lcpred}(S)$ is computed by taking the nodes in $\cap_{c \in S} \text{pred}(m(c))$ for which edges lead to nodes in $\cup_{c \in S} \text{pred}(m(c)) \setminus \cap_{c \in S} \text{pred}(m(c))$. The expressions from the dataflow graph in each set in $L_{i,j}$ with $j < i$ constitute the computational similarities of random variables with depth i with respect to parents at depth j. These similarities correspond to deterministic variables like β.

For the model from Fig. 1, $L_{1,1}$ contains singletons for the random variables at depth 1 like $\{m(\mu_1)\}$ and $\{m(\sigma_2)\}$. Analogously, $L_{2,2}$ and $L_{3,3}$ contains singletons for the random variables ϕ and the data entries y respectively. The direct parents of the variables with depth 3 are $\mu_1, \mu_2, \mu_5, \phi_{i,1}$ and $\phi_{i,2}$. Since $d(\phi_{i,1}) = d(\phi_{i,2}) = 2$ and $d(\mu_1) = d(\mu_2) = d(\mu_5) = 1$, two additional layers $L_{3,2}$ and $L_{3,1}$ will be introduced. Here, among others $\text{lcpred}(\text{ch}(\phi_{i,1}) \cap R_2)$ contains calls to `int_ode`, and $\text{lcpred}(\text{ch}(\phi_{i,1}) \cap R_2)$ contains calls to `h2`.

Finally, to turn the constructed layers $L_{i,j}$ into a partitioning of V, they are processed from shallowest to deepest while assigning all nodes in V. Each set $S' \in L_{i,j}$ is replaced by nodes in $\cup_{s \in S'} \text{pred}(s)$ except for those that have already been assigned. The resulting sets form the final tasks.

While it might seem that annotating the for-loops in the model description like the one given in Fig. 1 to specify that these should be parallelized is straightforward, the parallelization described here will not only automatically detect this, but it will also work for more arbitrarily interrelated models in which loops need not necessarily match the levels of the hierarchy.

The tasks within each layer can be scheduled to run in parallel. To maximize parallel efficiency [13], idle times need to be kept to a minimum. The only heuristic considered during performance evaluation is LPT [9] although other heuristics could be used as well. The focus is not so much on scheduling, but on presenting a mapping between two representations of a model to identify parallel parts.

Since subsequent posterior evaluations occur at similar positions in the parameter space, i.e. $\theta^t \approx \theta^{t+1}$, it turns out that the execution time for each task changes only gradually. For this reason, after running one iteration with tasks scheduled using a round-robin (RR) strategy, subsequent rounds can be scheduled with LPT using the execution time measured during evaluation of the previous candidate parameter θ.

5 Performance Evaluation

PMX models are key computational components leveraged for decision making during drug development. Here, only a limited amount of data is available [4]. The data includes the compound concentration in the blood of subjects, a costly measurement to make. In contrast to more classical models where all data is "independent and identically distributed", the data also specifies from which patient each measurement is taken creating a hierarchy as discussed above.

In this section, the performance of the proposed method is evaluated using two models from PMX. The first model, called the Nimotuzumab model, describes a humanized monoclonal antibody mAb, in patients with advanced breast cancer [21]. The second model is the Canagliflozin model used as the example in Sect. 3.

The structure is similar in both models; it consists of a population layer in which a set of patients that have taken part in the clinical trial are each modeled separately. However, it is important to note that the parallelization outlined in this paper can be applied to models with more layers assuming that there are enough computationally intensive tasks in each layer.

The data for the Nimotuzumab model contains measurements of 12 patients resulting in limited amount of parallelization. On the other hand, the data for the Canagliflozin model consists of measurements of 1144 patients. For this model, it is important to note that some patients in the placebo group are not given the compound, while others are given the compound for either a shorter or longer period. Therefore, the time required to simulate PK and PD for each patient varies drastically [14]. For example, execution time of numeric integration varies up to 100× across patients for Canagliflozin.

If all expressions are compiled separately, respectively 6643 and 46261 tasks are created for the two models. The overhead of running these tasks separately,

estimated by a run on a single system, slows down execution time by a few orders of magnitude. By applying the steps outlined in Sect. 4, the number of tasks drops to 375 and 9080 reducing task management overhead.

Table 1. The number of tasks per layer and the percentage of time in each layer for the two test models. Most of the time is spent in the fifth layer, where tasks that perform the numeric integration are concentrated. The final layer, with the most tasks, contains likelihood evaluations.

Model	Metric	$L_{1,1}$	$L_{2,1}$	$L_{2,2}$	$L_{3,1}$	$L_{3,2}$	$L_{3,3}$
Nimotuzumab	Tasks (#)	1	3	36	1	12	321
	Coverage (%)	0.00%	0.09%	1.43%	0.03%	90.35%	8.10%
Canagliflozin	Tasks (#)	1	2	2694	1	1144	5237
	Coverage (%)	0.00%	0.01%	0.03%	0.00%	99.90%	0.06%

The distribution of tasks across layers is shown in Table 1. Most of the computation time, 90% and 99% respectively, is spent in the numeric integration of the PK and PD equations. The tasks that perform this integration are captured in a single layer. Both models compile to 5 layers with the most tasks in the last layer containing likelihood evaluations. Since likelihood evaluations in these models are lightweight, they also serve to demonstrate that the presented parallelization can be too aggressive as all layers are parallelized while manual parallelization would only assign more resources in the layer that captures numeric integration tasks.

The number of messages exchanged between processors depends on how tasks are scheduled, and varies at runtime for each evaluation when the scheduling step reassigns tasks. It is important to note that the LPT heuristic has a local view. Tasks in each layer are scheduled without considering the assignment of tasks in other layers.

Figure 3 compares performance when tasks in a phase are scheduled using a RR strategy or by using the LPT heuristic on a single Haswell system with 2 Xeon E5-2699 v3 @ 2.30 GHz CPUs, each with 18 physical cores for a total of 36 cores. The parallelization was implemented in the Julia programming language [2]. For the sake of stability of the results, frequency scaling was disabled. While other custom message passing implementations were also tested, the results are reported for an implementation relying on Intel MPI Version 2018 as it is widely available. Preparing and copying messages adds overhead, but note that since the results are for a single system, this could be avoided by using threads instead. Nevertheless, the mapping between the two representations with this overhead still shows promising performance scalability. It is also applicable to larger systems with a higher latency interconnect as long as the tasks are sufficiently compute intensive.

Since the outlined approach uses the message passing model for parallel execution, the more general term "processor" is used here [13]. The comparison is

made in terms of the speedup achieved by running on p processors, denoted by S_p and given by the ratio between the execution time with one processor and p processors, i.e. T_1/T_p. As both T_1 and T_p are stochastic due to noise in the system [15,18], execution time is measured 200 times for each choice of p to obtain stable results. Samples for T_1 are paired with T_p to generate samples for S_p. The 5th and 95th quantiles are shown to quantify the spread of S_p.

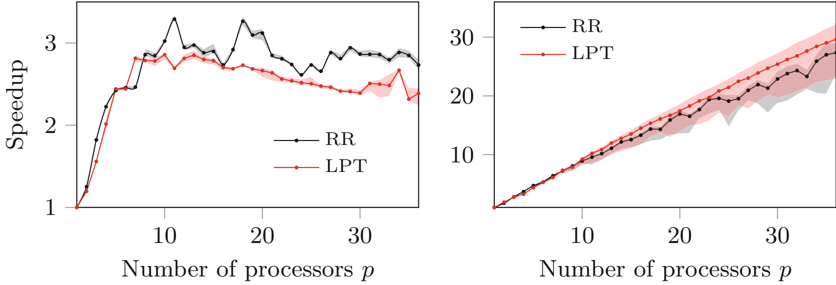

Fig. 3. Mean scalability of the Nimotuzumab model on the left and the Canagliflozin model on the right with the shaded regions showing the uncertainty range for the 5th and 95th quantile of the speedup. The efficacy of the parallelization approach is model dependent, but performance improves for both models.

The limited number of patients in the Nimotuzumab model causes execution time not to scale past approximately 10 processors. Note also that performance does not reach 10× with respect to baseline. Through profiling, it became apparent that this is not only due to the varying computational requirements between tasks associated with different patients, but also due to communication overhead. With the relatively small amount of available parallelism, this cannot be neglected, and it causes performance to degrade past 10 processors.

Note that the LPT heuristic results in slightly slower performance when compared to RR. This is due to the increase in the time spent communicating between some cores in some layers, an aspect not taken into account by the heuristic while in RR communication cost is spread more evenly.

Note also that initially, there is little to no difference between the two strategies. This is due to the two strategies behaving similarly when a few processors are used. As the number of processors increases, the performance of the two strategies diverges.

The Canagliflozin model scales better since there is a much larger opportunity for parallelization. Due to the amount of imbalance between patients, the LPT scheduling heuristic further improves performance by about 8%. Around 10% is lost due to overhead introduced by communication between processors and task management. This is verified by comparing to theoretically computed execution time where this overhead is ignored. Note that efficiency, computed by comparing actual scalability with linear scalability, stays above 90%. From this, it can be concluded that most of the available parallelism is exploited.

Since multiple processors are employed in each layer of the hierarchy, it only improves performance in layers with tasks that take a sufficient amount of computation to dwarf communication overhead. For layers with small tasks, the benefits of parallel execution will be outweighed by the overhead introduced by communication. In this case, overall performance will improve only when other compute-intensive layers make the overhead for layers with many small tasks negligible.

6 Future Work and Conclusion

This paper introduces a novel way to parallelize evaluation of hierarchical models by observing that conditional independence in a graphical model representation can be mapped to the dataflow graph. The presented method has been shown to work for two characteristic models from PMX. Note that it is not limited to this domain. The efficacy of the model depends on the amount of parallelism inherent in the input models and the computational size of its tasks. The results show that by using a simple well-known scheduling heuristic within each layer, performance can further improve in case execution time varies between tasks.

One drawback of the presented method is that *all* layers are parallelized. As long as there are enough layers with many compute-intensive tasks, the presented approach results in high utilization of parallel resources. However, the communication introduced in layers with small, but numerous, tasks can degrade performance. Therefore, future work will explore how to disable parallelization selectively if communication overhead is high relative compared to the amount of computation.

The scheduling heuristic relies on the measured execution time of tasks during previous model evaluations. As long as the assumption holds that the execution time of tasks changes gradually while the encompassing sampling algorithm or optimization routine takes small steps in the parameter space, such an approach will suffice. There is additional overhead introduced by measuring and collecting the execution time of each task. Therefore, future work will study the trade-off of occasionally disabling these measurements while the scheduling heuristic uses less up-to-date measurements.

The current results were limited to a single system with communication between processors accomplished through memory. Another aspect that will be explored next is how to mitigate the latency of contemporary interconnects.

Finally, while the partitioning of nodes is used in this paper to construct tasks, using the resulting assignments for initializing more complex heuristics as those used in other work [20] to speed up convergence will be studied next.

Acknowledgments. Part of the work presented in this paper was funded by Johnson & Johnson.

References

1. Beck, M., Pingali, K.: From control flow to dataflow. Cornell University, Technical report (1989)
2. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: a fresh approach to numerical computing. SIAM Rev. **59**(1), 65–98 (2017). https://doi.org/10.1137/141000671
3. Błażewicz, J., Ecker, K.H., Pesch, E., Schmidt, G., Weglarz, J.: Handbook on Scheduling: From Theory to Applications. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-32220-7
4. Carey, V., Wang, Y.G.: Mixed-Effects Models in S and S-PLUS. Springer, New York (2001). https://doi.org/10.1007/b98882
5. Carpenter, B., et al.: Stan: a probabilistic programming language. J. Stat. Softw. **76**(1), (2017)
6. Casella, G., George, E.I.: Explaining the Gibbs sampler. Am. Stat. **46**(3), 167–174 (1992)
7. Chakravarthi, V.S.: SOC Physical Design. A Practical Approach to VLSI System on Chip (SoC) Design, pp. 173–199. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-23049-4_9
8. Culler, D.E.: Dataflow architectures. Annu. Rev. Comput. Sci. **1**(1), 225–253 (1986)
9. Della Croce, F., Scatamacchia, R.: The longest processing time rule for identical parallel machines revisited. J. Sched. **23**(2), 163–176 (2018). https://doi.org/10.1007/s10951-018-0597-6
10. Eijkhout, V.: Introduction to High Performance Scientific Computing. Lulu press, Morrisville (2012)
11. Ge, H., Xu, K., Ghahramani, Z.: Turing: a language for flexible probabilistic inference. In: International Conference on Artificial Intelligence and Statistics, pp. 1682–1690 (2018)
12. Goudie, R.J., Turner, R.M., De Angelis, D., Thomas, A.: Multibugs: a parallel implementation of the bugs modelling framework for faster Bayesian inference. arXiv preprint arXiv:1704.03216 (2017)
13. Grama, A., Kumar, V., Gupta, A., Karypis, G.: Introduction to Parallel Computing. Pearson Education, London (2003)
14. Haber, T., van Reeth, F.: Improving the runtime performance of non-linear mixed-effects model estimation. In: Schwardmann, U., et al. (eds.) Euro-Par 2019: Parallel Processing Workshops, Euro-Par 2019. Lecture Notes in Computer Science, vol. 11997. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48340-1_43
15. Hoefler, T., Belli, R.: Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12 (2015)
16. Kessler, C., Keller, J.: Models for parallel computing: review and perspectives. Mitteilungen-Gesellschaft für Informatik eV, Parallel-Algorithmen und Rechnerstrukturen **24**, 13–29 (2007)
17. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv. (CSUR) **31**(4), 406–471 (1999)
18. Lameter, C.: Shoot first and stop the OS noise. In: Linux Symposium, p. 159. Citeseer (2009)
19. Lunn, D.J., Thomas, A., Best, N., Spiegelhalter, D.: Winbugs-a Bayesian modelling framework: concepts, structure, and extensibility. Stat. Comput. **10**(4), 325–337 (2000). https://doi.org/10.1023/A:1008929526011

20. Nemeth, B., Haber, T., Liesenborgs, J., Lamotte, W.: Automatic parallelization of probabilistic models with varying load imbalance. In: International Symposium on Cluster, Cloud and Grid Computing (CCGRID) Workshop on High Performance Machine Learning Workshop (2020)
21. Rodríguez-Vera, L., et al.: Semimechanistic model to characterize nonlinear pharmacokinetics of nimotuzumab in patients with advanced breast cancer. J. Clin. Pharmacol. **55**(8), 888–898 (2015)
22. Saintes, F.: I-56 sebastian weber supporting drug development as a Bayesian in due time?!. In: Euro-Par, vol. 2020 (2019)
23. Sivia, D., Skilling, J.: Data Analysis: A Bayesian Tutorial. OUP Oxford, Oxford (2006)
24. de Winter, W., et al.: Dynamic population pharmacokinetic-pharmacodynamic modelling and simulation supports similar efficacy in glycosylated haemoglobin response with once or twice-daily dosing of canagliflozin. Br. J. Clin. Pharmacol. **83**(5), 1072–1081 (2017)
25. Yildirim, I.: Bayesian Inference: Gibbs Sampling. MIT Press, New York (2012)