Thune et al. *Journal of Mathematics in Industry* (2021) 11:12
https://doi.org/10.1186/s13362-021-00108-5

Journal of Mathematics in Industry
a SpringerOpen Journal

RESEARCH **Open Access**

On the impact of heterogeneity-aware mesh partitioning and non-contributing computation removal on parallel reservoir simulations

Andreas Thune[1,2*], Xing Cai[1,2] and Alf Birger Rustad[3]

*Correspondence: andreast@simula.no
[1]Simula Research Laboratory, Martin Linges vei 25, 1364 Fornebu, Norway
[2]University of Oslo, Oslo, Norway
Full list of author information is available at the end of the article

Abstract

Parallel computations have become standard practice for simulating the complicated multi-phase flow in a petroleum reservoir. Increasingly sophisticated numerical techniques have been developed in this context. During the chase of algorithmic superiority, however, there is a risk of forgetting the ultimate goal, namely, to efficiently simulate real-world reservoirs on realistic parallel hardware platforms. In this paper, we quantitatively analyse the negative performance impact caused by non-contributing computations that are associated with the "ghost computational cells" per subdomain, which is an insufficiently studied subject in parallel reservoir simulation. We also show how these non-contributing computations can be avoided by reordering the computational cells of each subdomain, such that the ghost cells are grouped together. Moreover, we propose a new graph-edge weighting scheme that can improve the mesh partitioning quality, aiming at a balance between handling the heterogeneity of geological properties and restricting the communication overhead. To put the study in a realistic setting, we enhance the open-source *Flow* simulator from the OPM framework, and provide comparisons with industrial-standard simulators for real-world reservoir models.

Keywords: Reservoir simulation; High performance computing; Mesh partitioning; Norne reservoir model

1 Introduction and motivation

Computer simulation is extensively used in the oil industry to predict and analyse the flow of fluids in petroleum reservoirs. The multi-phased flow in such porous media is mathematically described by a complicated system of partial differential equations (PDEs), only numerically solvable for realistic cases. At the same time, the quest for realism in reservoir simulation leads to using a large number of grid cells, thereby many degrees of freedom. Parallel computing is thus indispensable for achieving large scales and fast simulation time.

The fundamental step of parallelization is to divide the total number of degrees of freedom among multiple hardware processing units. Each processing unit is responsible for

© The Author(s) 2021. This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Springer

computing its assigned degrees of freedom, in collaboration with the other units. To use distributed-memory mainstream parallel computers for mesh-based computations, such as in a reservoir simulation, the division of the degrees of freedom is most naturally achieved by partitioning the global computational mesh. Each processing unit is therefore assigned with a sub-mesh consisting of two types of grid cells: *interior* and *ghost*. The distributed ownership of the interior cells gives a *disjoint* division of all the degrees of freedom among the processing units. The ghost cells per sub-mesh, which constitute one or several layers around the interior cells, are needed to maintain the PDE-induced coupling between the neighboring sub-meshes.

One major benefit of such a work division, based on mesh partitioning, is that each processing unit can *independently* discretize the PDEs restricted to its assigned sub-mesh. Any global linear or nonlinear system will thus only exist logically, collectively represented by a set of sub-systems of linear or nonlinear equations. The overall computing speed of a parallel simulator, however, hinges upon the quality of mesh partitioning. Apart from the usual objective of minimizing the inter-process communication volume, it is also desirable to avoid that strongly coupled grid cells are assigned to different processes. The latter is important for the effectiveness of parallel preconditioners that are essential for iteratively solving the linear systems arising from the discretized PDEs. The two objectives are not easy to achieve simultaneously.

It is therefore necessary to revisit the topic of mesh partitioning as the foundation of parallel reservoir simulations. In particular, the interplay between minimizing communication overhead and maximizing numerical effectiveness, especially in the presence of reservoir-characteristic features, deserves a thorough investigation. The novelty of this paper includes a study of how different edge-weighting schemes, which can be used in a graph-based method of mesh partitioning, will influence numerical effectiveness and communication overhead. We also quantify the negative performance impact caused by non-contributing computations that are associated with the ghost degrees of freedom. This subject is typically neglected by the practitioners of parallel reservoir simulations. Moreover, we present a simple strategy to avoid the non-contributing computations based on a reordering of the interior and ghost grid cells per subdomain.

The remainder of the paper is organized as follows. Section 2 gives a very brief introduction to the mathematical model and the numerical solution strategy for reservoir flow simulations. Then, Sect. 3 explains the parallelization with a focus on how to avoid non-contributing computations related to the unavoidable ghost grid cells. Thereafter, Sect. 4 devotes its attention to the details of mesh partitioning and the corresponding graph partitioning problem, with the presentation of a new edge-weighting scheme. The impacts of removing non-contributing computations and applying the new edge-weighting scheme are demonstrated by numerical experiments in Sect. 5, whereas Sects. 6 and 7, respectively, addresses the related work and provides concluding remarks.

2 Mathematical model and numerical strategy

In this section, we will give a very brief introduction to the most widely used mathematical model of petroleum reservoirs and a standard numerical solution strategy that is based on corner-point grids and cell-centered finite volume discretization.

2.1 The black-oil model

The standard mathematical model used in reservoir simulation is the black-oil model [1, 2]. It is a system of nonlinear PDEs governing three-phase fluid flow in porous media. The equations are derived from Darcy's law and conservation of mass. The model assumes that the different chemical species found in the reservoir can be separated into three categories of fluid phases $\alpha = \{w, o, g\}$: water (w), oil (o) and gas (g). There are consequently three main equations in the black-oil model, one for each phase:

$$\frac{\partial}{\partial t}\left[\frac{\phi S_o}{B_o}\right] = \nabla \cdot \left[\frac{k_{ro}K}{\mu_o B_o}\nabla\Phi_o\right] + q_o, \tag{1}$$

$$\frac{\partial}{\partial t}\left[\frac{\phi S_w}{B_w}\right] = \nabla \cdot \left[\frac{k_{rw}K}{\mu_w B_w}\nabla\Phi_w\right] + q_w, \tag{2}$$

$$\frac{\partial}{\partial t}\left[\phi\left(\frac{R_s S_o}{B_o} + \frac{S_g}{B_g}\right)\right] = \nabla \cdot \left[R_s\frac{k_{ro}K}{\mu_o B_o}\nabla\Phi_o + \frac{k_{rg}K}{\mu_g B_g}\nabla\Phi_g\right] + R_s q_o + q_{fg}. \tag{3}$$

Here, ϕ, K and R_s are porosity, permeability and gas solubility. They describe the geological properties of a reservoir. For each phase α, the terms S_α, μ_α, B_α and $k_{r\alpha}$ denote saturation, viscosity, formation volume factor and relative permeability. The phase potential Φ_α is defined by the phase pressure p_α and phase density ρ_α:

$$\Phi_\alpha = p_\alpha + \rho_\alpha \gamma z, \tag{4}$$

where γ and z are the gravitational constant and reservoir depth. The unknowns of the black-oil model are the saturation and pressure of each phase, so the following three relations are needed to complete Eqs. (1)-(3):

$$S_o + S_w + S_g = 1, \tag{5}$$

$$p_w = p_o - p_{cow}(S_w), \tag{6}$$

$$p_g = p_o + p_{cog}(S_g). \tag{7}$$

The dependencies of the capillary pressures p_{cow} and p_{cog} upon the saturations S_w and S_g, used in Eq. (6) and Eq. (7), are typically based on empirical models.

2.2 Well modelling

The right-hand sides of the black-oil model (Eqs. (1)-(3)) contain source/sink terms q_α, which represent either production or injection wells in a reservoir. The wells affect the fluid flow on a much finer scale than what is captured by the resolution of the computational mesh for the reservoir. Special well models, such as the Peaceman model [3], are incorporated to model important phenomena, such as stark pressure drops, in proximity to well in- and outflow. In the Peaceman well model, the pressure drop is modelled by introducing new variables and equations in cells that contain a well bottom-hole. The related well equations numerically couple all the grid cells perforated by each well.

2.3 Corner-point grid and discretization

It is common to use the 3D corner-point grid format [4] to represent a reservoir mesh. A corner-point grid is a set of hexahedral cells logically aligned in a Cartesian fashion.

The actual geometry of the grid is defined by a set of inclined vertical pillars, such that each grid cell in the mesh is initially formed by eight corner points on four of these pillars. Deformation and shifting of the sides of a cell are allowed independently of the horizontal neighboring cells. Moreover, a realistic reservoir may turn some of the cells to be inactive. The combined consequence is that the resulting computational mesh is unstructured. For example, a cell can have fewer than six sides, and there can be more than one neighboring cell on each side.

A standard cell-centred finite volume scheme, using two-point flux approximation with upwind mobility weighing [5], can be applied on a corner-point grid to discretize the PDEs of the black-oil model. The time integration is fully implicit to ensure numerical stability. Take for instance the water equation (Eq. (2)). Let $S_w^{\ell,i}$ denote S_w at time step ℓ inside cell C_i, which has V_i as its volume. Suppose the neighboring cells of C_i are denoted as $C_{j_0}, \ldots, C_{j_i}$, the discretization result of Eq. (2) restricted to cell C_i is thus

$$V_i \frac{\phi^i}{\Delta t}\left(\left(\frac{S_w}{B_w}\right)^{\ell+1,i} - \left(\frac{S_w}{B_w}\right)^{\ell,i}\right) - \sum_{j=j_0}^{j_i} \lambda_w^{\ell+1,ij} T_{ij}\left(\Phi_w^{\ell+1,i} - \Phi_w^{\ell+1,j}\right) = V_i q_w^{\ell+1,i}. \tag{8}$$

Here, $\lambda_w^{\ell+1,ij} = \frac{k_{rw}^{\ell+1,ij}}{\mu_w^{\ell+1,ij} B_w^{\ell+1,ij}}$ denotes the water mobility on the face intersection Γ_{ij} between a pair of neighboring cells C_i and C_j, whereas T_{ij} is the *static transmissibility* on Γ_{ij}:

$$T_{ij} = m_{ij}|\Gamma_{ij}|\left(\frac{\|\vec{c}_i\|^2}{\vec{n}_i K_i \vec{c}_i} + \frac{\|\vec{c}_j\|^2}{\vec{n}_j K_j \vec{c}_j}\right)^{-1}. \tag{9}$$

The m_{ij} term denotes a transmissibility multiplier, for incorporating the effect of faults. For example, when a fault acts as a barrier between cells C_i and C_j, we have $m_{ij} = 0$. Figure 1 illustrates the geometric terms $\vec{c}_i$, $\vec{n}_i$ and Γ_{ij} involved in the transmissibility calculation. A typical scenario of reservoir simulation is that s_w, s_g and p_o are chosen as the primary unknowns, and the cell-centered finite volume method is applied to the three main equations Eqs. (1)-(3) on all the grid cells. As result we get a system of nonlinear algebraic equations per time step. The total number of degrees of freedom is three times the number of active grid cells. Newton iterations are needed at each time level, such that a series

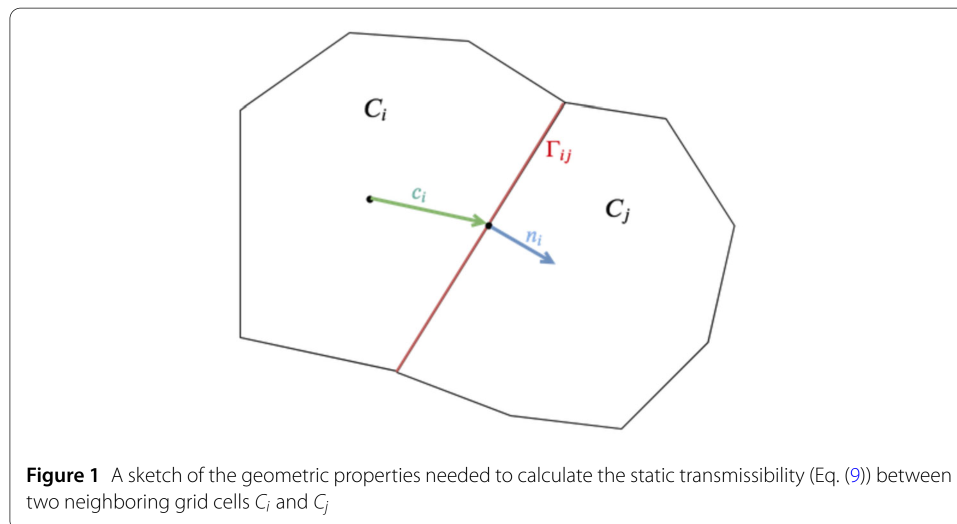

Figure 1 A sketch of the geometric properties needed to calculate the static transmissibility (Eq. (9)) between two neighboring grid cells C_i and C_j

of linear systems $Ax = b$ will be solved by an iterative method, such as BiCGStab or GMRES [6], which is accelerated by some preconditioner. The linear systems are sparse, often ill-conditioned, and non-symmetric due to the influence of well models. Although the nonzero values in the matrix A change with the time level and Newton iteration, the sparsity pattern remains unchanged (as long as the corner-point grid is fixed). This allows for a static partitioning of the computational mesh needed for parallelization. In this context, the static transmissibility T_{ij} defined in Eq. (9) is an important measure of the coupling strength between a pair of neighboring cells C_i and C_j.

3 Efficient parallelization of reservoir simulation

To parallelize the numerical strategy outlined in the preceding section, several steps are needed. The main focus of this section will be on two topics. First, we explain why ghost grid cells need to be added per sub-mesh after the global computational mesh is non-overlappingly partitioned. Second, we pinpoint the various types of non-contributing computations that arise due to the ghost cells, and show how these can be avoided for a better computational efficiency. We remark that the exact amount and spread of ghost cells among the sub-meshes are determined by the details of the mesh-partitioning scheme, which will be the subject of Sect. 4.

3.1 Parallelization based on division of cells

Numerical solution of the black-oil model consists of a time integration procedure, where during each time step several Newton iterations are invoked to linearize the nonlinear PDE system in Eqs. (1)-(3). The linearized equations are then discretized and solved numerically. The main computational work inside every Newton iteration is the construction and subsequent solution of a linear system of the form $Ax = b$. Typically, the 3D corner-point grid remains unchanged throughout the entire simulation. It is thus customary to start the parallelization by a static, non-overlapping division of the grid cells evenly among a prescribed number of processes. Suppose N denotes the total number of active cells in the global corner-point grid, and N_p is the number of cells assigned to process p, then we have

$$\sum_{p=1}^{P} N_p = N,$$

where P denotes the total number of processes. Process p is responsible for computing the $3N_p$ degrees of freedom that live on its N_p designated cells. The global linear system $Ax = b$ that needs to be calculated and solved inside each Newton iteration will only exist *logically*, i.e., collectively composed by the $3N_p$ rows of A and the $3N_p$ entries of x and b that are owned by every process $p = 1, 2, \ldots, P$.

3.2 The need for ghost cells

The non-overlapping cell division gives a"clean-cut" distribution of the computational responsibility among the P processes, specifically, through a divided ownership of the x entries. There are however two practical problems for parallel computing associated with such a non-overlapping division.

First, we recall that the numerical coupling between two neighboring cells C_i and C_j is expressed by the static transmissibility T_{ij} as defined in Eq. (9). In case C_i and C_j belong to

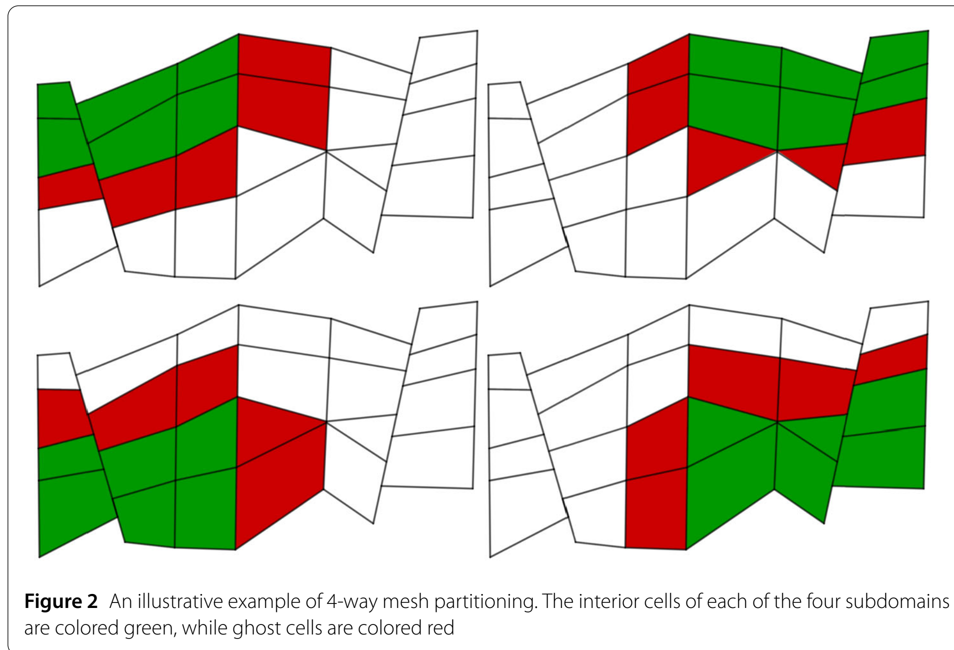

Figure 2 An illustrative example of 4-way mesh partitioning. The interior cells of each of the four subdomains are colored green, while ghost cells are colored red

two different processes, inter-process exchange of data is required in the parallel solution procedure. If each process has a local data structure storing only its designed $3N_p$ entries of x, the inter-process communication will be in the form of many individual 3-value exchanges, resulting in a drastic communication overhead. It is thus common practice to let each process extend its portion of the x vector by $3N_p^G$, where N_p^G denotes the number of *ghost* cells that are not owned by process p but border its internal boundary. For the finite volume method considered in this paper, only one layer of ghost cells is needed. Figure 2 demonstrates how ghost cells are added to the local grids of each process. The extended local data structure will allow aggregated inter-process communication, i.e., all the values needed by process p from process q are sent in one batch, at a much lower communication overhead compared with the individual-exchange counterpart. Specifically, whenever x has been distributedly updated, process p needs to receive in total $3N_p^G$ values of x from its neighbors. To distinguish between the two types of cells, we will from now on denote the originally designated N_p cells from the non-overlapping division as *interior* cells on process p.

Second, and perhaps more importantly, if a local discretization is carried out on process p by restricting to its designated N_p interior cells, the resulting local part of A, which is of dimension $3N_p \times 3N_p$, will be incomplete on the rows that correspond to the cells that have one or more of their neighboring cells owned by a process other than p. A similar problem also applies to the corresponding local entries in the vector b. Elaborate inter-process communication can be used to expand the sub-matrix on process p to be of dimension $3N_p \times 3(N_p + N_p^G)$, for fully accommodating the numerical coupling between process p and all its neighboring processes. However, a *communication-free* and thereby more efficient local discretization approach is to let each process also include its ghost cells. More specifically, the local discretization per process is independently done on a sub-mesh that comprises both the interior and ghost cells. This computation can reuse a sequential discretization code, without the need of writing a specialized subdomain discretization procedure. The resulting sub-matrix A_p will therefore be of dimen-

sion $3(N_p + N_p^G) \times 3(N_p + N_p^G)$ and the sub-vector b_p of length $3(N_p + N_p^G)$. We note that the $3N_p^G$ "extra" rows (or entries) in A_p (or b_p) that correspond to the N_p^G ghost cells will be incomplete/incorrect, but they do not actively participate in the parallel computation later. One particular benefit of having a square A_p is when a parallelized iterative solver for $Ax = b$ relies on a parallel preconditioner that adopts some form of incomplete factorization per process. The latter typically requires each local matrix A_p to be of a (logically) square shape.

In the following, we will discuss what types of computation and memory overhead can arise due to the ghost cells and how to alleviate them.

3.3 Non-contributing computation and memory overhead due to ghost cells

While promoting communication-free discretizations per sub-mesh and aggregated inter-process exchanges of data, the ghost cells (and the associated ghost degrees of freedom) on every sub-mesh do bring disadvantages. If not treated appropriately, these can lead to wasteful computations that are discarded later, as well as memory usage overhead. Such issues normally receive little attention in parallel reservoir simulators. To fully identify these performance obstacles, we will now dive into some of the numerical and programming details related to solving $Ax = b$ in parallel.

For any Krylov-subspace iterative solver for $Ax = b$, such as BiCGStab and GMRES [6], the following four computational kernels must be parallelized:

- Vector addition: $w = u + v$. If all the involved vectors are distributed among the processes in the same way as for x and b, then no inter-process communication is needed for a parallel vector addition operation. Each process simply executes $w_p = u_p + v_p$ independently, involving the sub-vectors. However, unless the result vector w is used as the input vector to a subsequent matrix-vector multiplication (see below), the floating-point operations and memory traffic associated with the ghost-cell entries are wasted. It is indeed possible to test for each entry of w_p whether it is an interior-cell value or not, thus avoiding the non-contributing floating-point operations, but such an entry-wise `if-test` may dramatically slow down the overall execution of the parallel vector addition. Moreover, the memory traffic overhead due to the ghost-cell entries cannot be avoided on a cacheline based memory system, if the ghost-cell and interior-cell entries are "intermingled" in memory.
- Inner product: $u \cdot v$. Again, we assume that both sub-vectors u_p and v_p have $3(N_p + N_p^G)$ entries on process p. It is in fact numerically *incorrect* to let each process simply compute its local inner product $u_p \cdot v_p$, before summing up the local contributions from all the processes by a collective communication (such as the `MPI_Allreduce` function). The remedy is to let each process "skip" over the ghost-cell entries in u_p and v_p. In a typical scenario that the ghost-cell entries are mixed with interior-cell entries in u_p and v_p, some extra implementation effort is needed. For example, an assistant integer array named `mask` can be used, which is of length $N_p + N_p^G$, where `mask[i]==1` means cell i is interior and `mask[i]==0` means otherwise. Assume the three degrees of freedom per cell are stored contiguously in memory, the following code segment is a possible implementation of the parallel inner product:

```
double sub_dot_p = 0, global_dot_p;
for (int i=0; i<sub_num_cells; i++) {
```

```
    sub_dot_p += mask[i]*(sub_u[3*i]*sub_v[3*i]
                         +sub_u[3*i+1]*sub_v[3*i+1];
                         +sub_u[3*i+2]*sub_v[3*i+2]);
  }
  MPI_Allreduce (&sub_dot_p, &global_dot_p, 1, MPI_DOUBLE,
                 MPI_SUM, MPI_COMM_WORLD);
```

For example, the well-known DUNE software framework [7] adopts a similar implementation. It is clear that the floating-point operations associated with the ghost-cell entries in u_p and v_p, as well as all the multiplications associated with the array `mask`, are non-contributing work. Allocating the array `mask` also incurs memory usage and traffic overhead.

- Sparse matrix-vector multiplication: $u = Av$. Here, we recall that the global matrix A is logically represented by a sub-matrix A_p per process, arising from a communication-free discretization that is restricted to a sub-mesh comprising both interior and ghost cells. The dimension of A_p is $3(N_p + N_p^G) \times 3(N_p + N_p^G)$. Moreover, we assume that all the ghost-cell entries in the sub-vector v_p are consistent with their "master copies" that are owned by other processes as interior-cell entries. This can be ensured by an aggregated inter-process data exchange. Then, a parallel matrix-vector multiplication can be easily realized by letting each process independently execute $u_p = A_p v_p$. We note that the ghost-cell entries in u_p will not be correctly computed (an aggregated inter-process data exchange is need if, e.g., u_p is later used as the input to another matrix-vector multiplication). Therefore, the floating-point operations and memory traffic associated with the ghost-cell entries in u_p and the ghost-cell rows in A_p are non-contributing.
- Preconditioning operation: $w = M^{-1}u$. For faster and more robust convergence of a Krylov-subspace iterative solver, it is customary to apply a preconditioning operation to the result vector of a preceding matrix-vector multiplication. That is, a mathematically equivalent but numerically more effective linear system $M^{-1}Ax = M^{-1}b$ is solved in reality. The action of a parallelized preconditioner M^{-1} is typically applying $w_p = \tilde{A}_p^{-1} u_p$ per sub-mesh, where $\tilde{A}_p^{-1}$ denotes an inexpensive numerical approximation of the inverse of A_p. One commonly used strategy for constructing $\tilde{A}_p^{-1}$ is to carry out an incomplete LU (ILU) factorization [6] of A_p. Similar to the case of parallel matrix-vector multiplication, the floating-point operations and memory traffic associated with the ghost-cell entries in w_p and the ghost-cell rows in A_p are non-contributing.

3.4 Handling non-contributing computation and memory overhead

The negative impact on the overall parallel performance, caused by the various types of non-contributing computation and memory usage/traffic overhead, can be large. This is especially true when the non-overlapping mesh partitioning is of insufficient quality (details will be discussed in Sect. 4). It is thus desirable to eliminate, as much as possible, the non-contributing computation and memory overhead.

A closer look at the four kernels that are needed in the parallel solution of $Ax = b$ reveals the actual "evil". Namely, the interior-cell entries and ghost-cell entries are intermingled. This is a general situation if the interior and ghost cells of a sub-mesh are ordered to obey the original numbering sequence of the corresponding cells in the global 3D corner-point grid. (This is standard practice in parallel PDE solver software.) Hence, the key to avoiding

non-contributing computation and memory overhead is a separation of the interior-cell entries from the ghost-cell counterparts in memory. This can be achieved per sub-mesh by deliberately numbering all the ghost cells after all the interior cells, which only needs to be done once and for all. If such a local numbering constraint is enforced, the non-contributing computation and memory overhead can be almost completely eliminated.

Specifically, the parallel vector addition and inner-product can now simply stop at the last interior degree of freedom. The array `mask` is thus no longer needed in the parallel inner-product operation. For the parallel matrix-vector multiplication, the per-process computation can stop at the last interior-cell row of A_p. In effect, the local computation only touches the upper $3N_p \times 3(N_p + N_p^G)$ segment of A_p. The last $3N_p^G$ rows of A_p are not used. This also offers an opportunity to save the memory storage related to these "non-contributing" rows. More specifically, each of the last $3N_p^G$ rows can be zeroed out and replaced with a single value of 1 on the main diagonal. As a result, the sub-matrix A_p on process p is of the following new form:

$$A_p = \begin{bmatrix} A_p^{II} & A_p^{IG} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}, \tag{10}$$

where the A_p^{II} block is of dimension $3N_p \times 3N_p$ and stores the numerical coupling among the $3N_p$ interior degrees of freedom, whereas the A_p^{IG} block is of dimension $3N_p \times 3N_p^G$ and stores the numerical coupling between the $3N_p$ interior degrees of freedom and the $3N_p^G$ ghost degrees of freedom.

The "condensed" sub-mesh matrix A_p in Eq. (10) is still of a square shape. This is mainly motivated by the situations where an incomplete factorization (such as ILU) of A_p is used as M^{-1} restricted to sub-mesh p in a parallel preconditioner setting. Clearly, having only a nonzero diagonal for the last $3N_p^G$ rows of A_p means that there is effectively no computational work associated with these rows in an ILU, which is a part of the preparation work of a Krylov-subspace solver before starting the linear iterations. Moreover, the forward-backward substitutions, which are executed within each preconditioning operation, also have negligible work associated with the "ghost" rows in the condensed A_p. Compared with the non-condensed version of A_p, which arises directly from a local discretization on the sub-mesh comprising both interior and ghost cells, the condensed A_p is superior in the amount of computational work, the amount of memory usage and traffic, as well as the preconditioning effect. The latter is due to the fact that a "natural" non-flux boundary condition is implicitly enforced on the ghost rows of the non-condensed version of A_p. This is, e.g., incompatible with a parallel Block-Jacobi preconditioner that effectively requires M^{-1} to be on the form:

$$M^{-1} = \begin{pmatrix} (\widetilde{A}_{p=1}^{II})^{-1} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & (\widetilde{A}_{p=2}^{II})^{-1} & \cdots & \mathbf{0} \\ \vdots & \cdots & \ddots & \vdots \\ \mathbf{0} & \cdots & \cdots & (\widetilde{A}_{p=P}^{II})^{-1} \end{pmatrix}. \tag{11}$$

4 Mesh partitioning

As mentioned in the previous section, the first step of parallelizing a reservoir simulator is a disjoint division of all the cells in the global corner-point grid, i.e., a non-overlapping

mesh partitioning. We have shown how to eliminate the non-contributing computation and memory overhead, which are associated with the necessary inclusion of one layer of ghost cells per sub-mesh. The actual amount of ghost cells per sub-mesh depends on the non-overlapping division, which will be the subject of this section. One aim is to keep the number of resulting ghost cells low, for limiting the overhead of aggregated inter-process communication. At the same time, we want to ensure good convergence effectiveness of a parallel preconditioner such as the Block-Jacobi method that uses ILU as the approximate inverse of the sub-matrix A_p (Eq. (10)) per process.

For the general case of an unstructured global corner-point grid, the standard strategy for a disjoint division of the grid cells is through partitioning a corresponding graph. The graph is translated from the global grid by turning each grid cell into a graph vertex. If grid cells i and j share an interface, it is translated into a (weighted) edge between vertex i and vertex j in the graph. This standard graph-based partitioning approach is traditionally focused on load balance and low communication volume. The convergence effectiveness of a resulting parallel preconditioner is normally not considered. We will therefore propose a new edge-weighting scheme to be used in the graph partitioner, which targets specifically the reservoir simulation scenario. The objective is to provide a balance between the pure mesh-partitioning quality metrics and the convergence effectiveness.

4.1 Graph partitioning

A graph $G = (V,E)$ is composed of a set of vertices V and a set of edges $E \subset V \times V$ connecting pairs of vertices in V. If weights are assigned to each member of V and E through weighting functions $\sigma : V \to \mathbb{R}$ and $\omega : E \to \mathbb{R}$, then we get a weighted graph $G = (V,E,\sigma,\omega)$. The P-way graph partitioning problem is defined as follows: Partition the vertex set V into P subsets $V_1, V_2, \dots, V_P$ of approximately equal size, while minimizing the summed weight of all the "cut edges" $e = (v_i, v_j) \in E$ connecting vertices belonging to different vertex subsets. Suppose $\mathcal{C}$ denotes the cut set of a partitioned G, containing all the cut edges. The graph partitioning problem can be formulated more precisely as a constrained optimization problem, where the objective function J is the sum of the weights of all members of $\mathcal{C}$, also called *edge-cut*:

$$\min J(\mathcal{C}) = \sum_{e\in\mathcal{C}} \omega(e), \tag{12}$$

$$\text{Subject to } \frac{\max_p \sum_{v\in V_p} \sigma(v)}{\frac{1}{P}\sum_{v\in V} \sigma(v)} < \epsilon, \tag{13}$$

where $\epsilon \geq 1$ is the imbalance tolerance of the load balancing constraint.

When the edge-weight function ω is uniform, i.e., each edge has a unit weight, the edge-cut is an approximation of the total volume of communication needed, e.g., before each parallel matrix-vector multiplication (see Sect. 3.3). When the edge-weight function ω is non-uniform, however, the objective function is no longer an approximation of communication overhead. As demonstrated in e.g. [8, 9], adopting non-uniform edge weights in the partitioning graph can be beneficial when partitioning linear systems with highly heterogeneous coefficients. Assigning edge weights based on the "between-cell coupling strength" can improve the quality of parallel preconditioners, such as Block-Jacobi (Eq. (11)).

Because the graph partitioning problem in Eqs. (12)-(13) is NP-complete, solving it exactly is practically impossible. Many existing algorithms find good approximate solutions, and implementations of these are available in several open software libraries such as Metis [10], Scotch [11] and Zoltan [12].

4.2 Edge-weighting strategies in graph partitioning for reservoir simulation

Although the black-oil equations are nonlinear and time dependent, and the values in the global linear system $Ax = b$ vary with each Newton iteration and time step, the global corner-point grid remains unchanged. The required mesh partitioning can thus be done once and for all, at the beginning of the simulation. When translating the corner-point grid to a corresponding graph, there are two *reservoir-specific* tasks. The first is that in case two neighboring cells i and j have a zero value for the static transmissibility T_{ij} (e.g., due to a barrier fault), the corresponding edge in the graph is removed, because such a pair of cells is not numerically coupled. The second is to include additional edges connecting vertex pairs corresponding to all the cells penetrated by a common well, because these grid cells are numerically coupled. We denote the set of well-related edges as E_w. The set containing the other regular edges is denoted by E_f. It is practical to avoid dividing a well (the penetrated cells) among multiple subdomains, and one way to achieve this is to ascribe a large edge weight to the edges in E_w. A corresponding uniform edge-weighting strategy for the edges in E_f, while ensuring that no well is partitioned between subdomains, is defined as follows:

$$\omega_u(e) = \begin{cases} \infty & e \in E_w, \\ 1 & e \in E_f. \end{cases} \tag{14}$$

In the above formula we ascribe weights of ∞ to the well edges. When implementing the edge-weighting scheme, the ∞-weights must be replaced by a large numerical value. We choose to use the largest possible value on a computer for the edge-weight data type.

To ensure good convergence effectiveness of a parallel preconditioner, we can modify the above uniform edge-weighting strategy by using the static transmissibility T_{ij} that lives on each cell interface (Eq. (9)). This is because T_{ij} can be used to estimate the between-cell flux, hence directly related to the magnitude of off-diagonal elements in A. Therefore, T_{ij} can be considered to describe the strength of the between-cell coupling. A commonly used edge-weighting strategy based on transmissibility is thus

$$\omega_t(e) = \begin{cases} \infty & e \in E_w, \\ T_{ij} & e = (v_i, v_j) \in E_f. \end{cases} \tag{15}$$

The transmissibility values can vary greatly in many realistic reservoir cases. One example of transmissibility heterogeneity can be seen with the Norne case, displayed in the histogram plot in Fig. 3b. Here, we observe a factor of more than 10^{12} between the smallest and largest transmissibility values. Using these transmissibilities directly as the edge weights, we can get partitioning results with a potentially large communication volume. Down-scaling the weights in Eq. (15) may help to decrease the communication overhead, while still producing partitions that yield better numerical performance than the uniform-weighted graph partitioning. We therefore propose an alternative edge-weighting strategy

by using the logarithm of T_{ij} as the weight of edge $e = (v_i, v_j)$:

$$\omega_l(e) = \begin{cases} \infty & e \in E_w, \\ \log(\frac{T_{ij}}{T_{\min}}) & e = (v_i, v_j) \in E_f. \end{cases} \tag{16}$$

5 Numerical experiments

In this section, we will investigate the effect of removing the non-contributing computations in solving $Ax = b$ (see Sect. 3), as well as using different edge-weighting strategies for graph partitioning (see Sect. 4), on parallel simulations of a realistic reservoir model. The main objective of the experiments is to quantify the impact on the overall simulation execution time. Moreover, for the different edge-weighting strategies, we will also examine the resulting numerical effectiveness and parallel efficiency. We have conducted our experiments on the publicly available Norne model, that we will describe in Sect. 5.1. In Sect. 5.4 we will compare the parallel performance of a thus improved open-source simulator with industry-standard commercial reservoir simulators.

We employ the open-source reservoir simulator *Flow* [13] to conduct our experiments and test our alternative implementations and methods. *Flow* is provided by the Open Porous Media (OPM) initiative [14], which is a software collaboration in the domain of porous media fluid flow. *Flow* offers fully-implicit discretizations of the black-oil model, and accepts the industry standard ECLIPSE input format. The linear algebra and grid implementations are based on DUNE [7]. In our experiments we restrict ourselves to the BiCGStab iterative solver, combined with a parallel Block-Jacobi preconditioner that uses ILU(0) as the approximate subdomain solver. Although *Flow* supports more sophisticated preconditioners, we stick to *Flow*'s recommended default option. As of yet, experiences with *Flow* show that ILU(0) achieves better overall performance than the alternatives on most relevant models.

Parallelization of *Flow* is mostly enabled by using the Message Passing Interface (MPI) library. For matrix assembly and I/O, shared memory parallelism with the OpenMP threading library is also available. Graph partitioning in *Flow* is performed by Zoltan. The well implementation in *Flow* does not allow for the division of a well over multiple subdomains. In addition to the well related edge-weights that discourage cutting wells, a post-processing procedure in *Flow* ensures that the cells perforated by a well always are contained on a single subdomain.

5.1 The Norne field model

To study reservoir simulation in a realistic setting we require real-world reservoir models. One openly available model that fits this criterion is the Norne benchmark case [15], which is a black-oil model case based on a real oil field in the Norwegian Sea. The model grid consists of 44,420 active cells, and has a heterogeneous and anisotropic permeability distribution. The model also includes 36 wells, which have changing controls during the simulation. Figure 3 depicts the Norne mesh colored by the x-directed permeability values, plus a histogram of the static transmissibility values.

In the histogram in Fig. 3b, we can see a huge span of the transmissibilities (Eq. (9)) of the Norne benchmark case. The majority of transmissibilities have a value between 10^{-16} and 10^{-9}. This large variation is a result of the heterogeneous distribution of permeability and cell size shown in Fig. 3a.

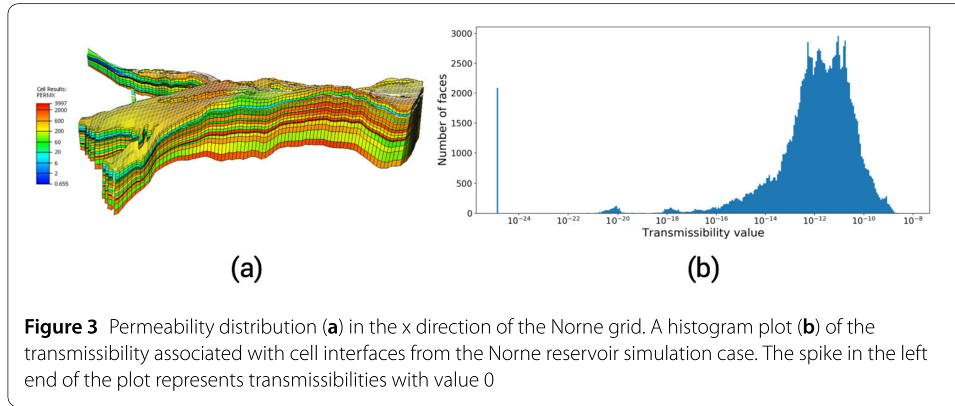

Figure 3 Permeability distribution (**a**) in the x direction of the Norne grid. A histogram plot (**b**) of the transmissibility associated with cell interfaces from the Norne reservoir simulation case. The spike in the left end of the plot represents transmissibilities with value 0

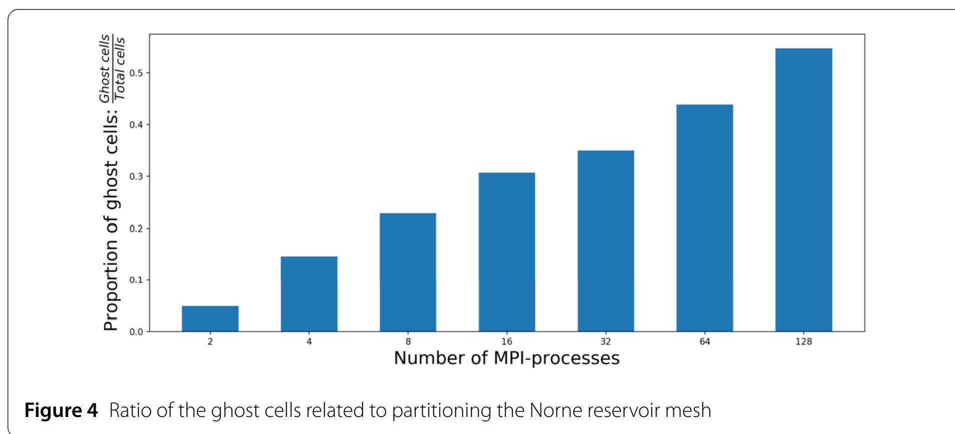

Figure 4 Ratio of the ghost cells related to partitioning the Norne reservoir mesh

Because of the modest grid size of the Norne model, we will also consider a refined version [16], where the cells are halved in each direction, resulting in a model with 8 times as many grid cells as the original model. The number of active cells in the refined Norne model is 355360.

We conducted our experiments of the Norne models on two computing clusters: Abel [17] and Saga [18]. The Abel cluster consists of nodes with dual Intel Xeon E5-2670 2.6 GHz 8-core CPUs interconnected with an FDR InfiniBand (56 Gbits/s) network, whereas Saga is a cluster of more modern dual socket Intel Xeon-Gold 6138 2.0 GHz 20-core CPUs interconnected with an EDR InfiniBand network (100 Gbits). The nodes on Abel and Saga have a total of 16 and 40 cores respectively. All experiments that use more MPI-processes than there are cores available on a single node are conducted on multiple nodes.

In all experiments using *Flow* on the original and refined Norne models, we have turned off the OpenMP multithreading for system assembly inside *Flow*.

5.2 Impact of removing non-contributing computations

We study the impact of non-contributing computations on the performance of linear algebra kernels and the overall performance of *Flow*, by carrying out experiments of the original Norne model described above. For these experiments we have used the transmissibility edge weights in Eq. (15), the default choice of *Flow*, in the graph partitioning scheme.

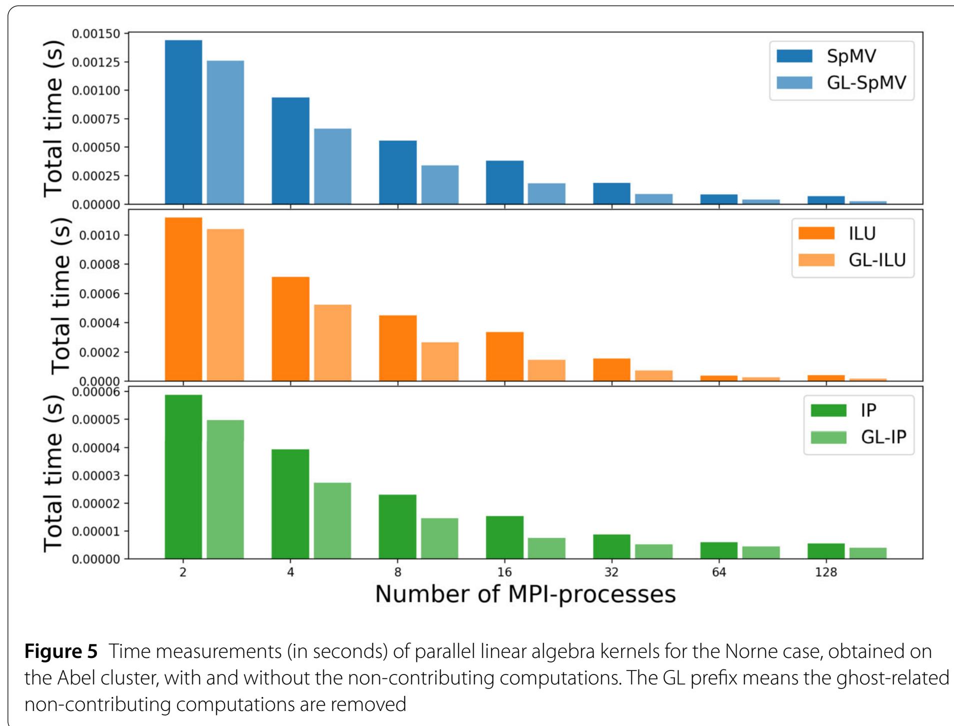

Figure 5 Time measurements (in seconds) of parallel linear algebra kernels for the Norne case, obtained on the Abel cluster, with and without the non-contributing computations. The GL prefix means the ghost-related non-contributing computations are removed

Figure 4 shows the ratio of ghost cells, i.e., $(\sum_{p=1}^{P} N_p^G)/N$, as a function of the number of subdomains used. We can see that the ghost cells can make up a significant proportion. For example, with $P = 128$, the ghost cell ratio is as high as 55%. As discussed in Sect. 3, non-contributing computations can arise due to the ghost cells. Because ghost cells increase as a proportion of total cells with an increasing number of MPI-processes, avoiding ghost cell related non-contributing computations is crucial for achieving good strong scaling.

To show the negative impact of non-contributing computations on the performance of solving $Ax = b$, we present the execution time of the linear algebra kernels used in the original Norne case in Fig. 5. It displays time measurements of sparse matrix-vector multiplication (SpMV), ILU's forward-backward substitution and inner product (IP), with and without the non-contributing computations. These time measurements are attained on the Abel cluster using selected matrices and vectors from the *Flow* Norne simulation. Using these matrices and vectors, SpMV, ILU forward-backward substitution and inner product operations are executed and timed.

Figure 5 shows a significant time improvement for the linear algebra kernels, due to removing the non-contributing computations as proposed in Sect. 3. Because the IP operation includes a collective reduction communication, it does not scale as well as the SpMV and ILU operations. We therefore observe that the relative improvement attained by removing the non-contributing IP computations start to decrease when the number of processes is higher than 16. A closer look at the performance of the linear algebra kernels is presented in Fig. 6 for the case of 16 MPI-processes. Here, the per-process execution times of SpMV, ILU and IP are displayed, with and without the non-contributing computations.

The top right plot of Fig. 6 displays the per-process numbers of interior and ghost cells. We notice a very uneven distribution of ghost cells among the processes. We also observe

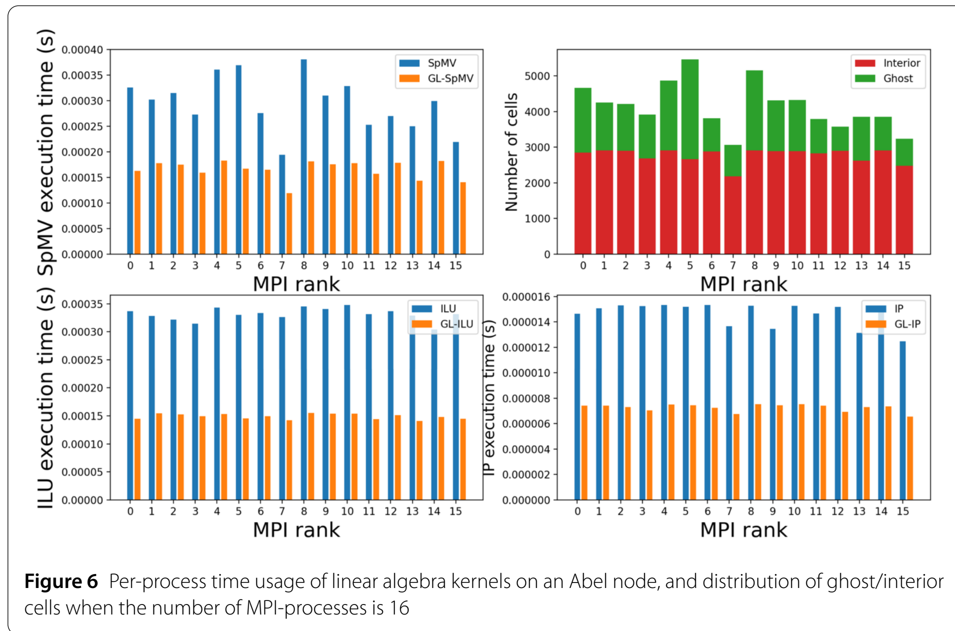

Figure 6 Per-process time usage of linear algebra kernels on an Abel node, and distribution of ghost/interior cells when the number of MPI-processes is 16

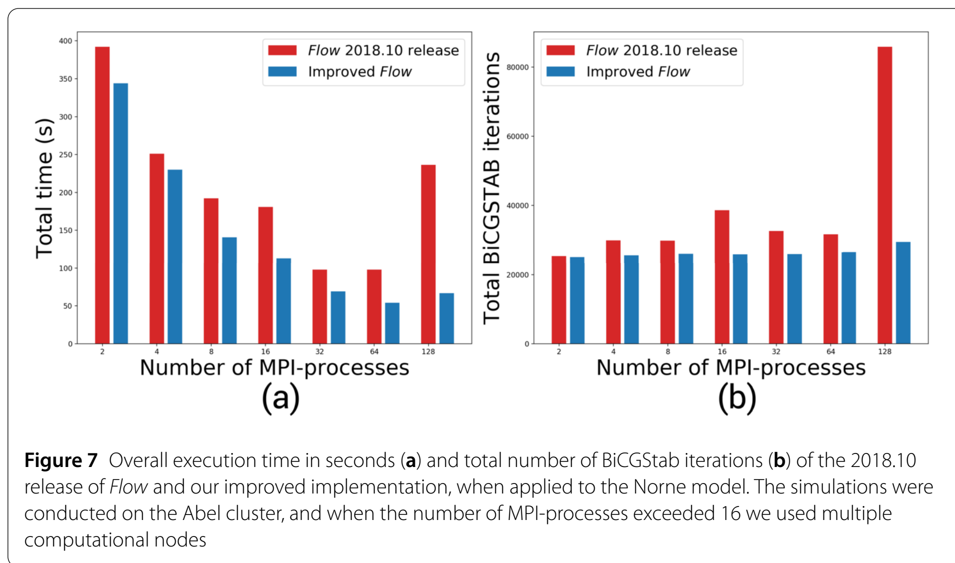

Figure 7 Overall execution time in seconds (**a**) and total number of BiCGStab iterations (**b**) of the 2018.10 release of *Flow* and our improved implementation, when applied to the Norne model. The simulations were conducted on the Abel cluster, and when the number of MPI-processes exceeded 16 we used multiple computational nodes

that the load imbalance induced by the ghost cells impacts the performance of the linear algebra operations, especially for SpMV.

The results displayed in Fig. 5 and Fig. 6 show the impact of ghost cells on the performance of key linear algebra operations, and the benefit of avoiding the non-contributing computations. To see the impact on the overall simulation time, we compare our improved implementation of *Flow* with the 2018.10 release of OPM's *Flow*, which is the latest release that does not contain any of the optimizations mentioned in this paper. The comparison results are displayed in Fig. 7, where the overall execution time and total number of BiCGStab iterations are presented.

In Fig. 7 we observe that our improved implementation of *Flow* achieves a significant improvement in both execution time and total iteration count. The latter is due to using the condensed local sub-matrix A_p of form Eq. (10), instead of the non-condensed

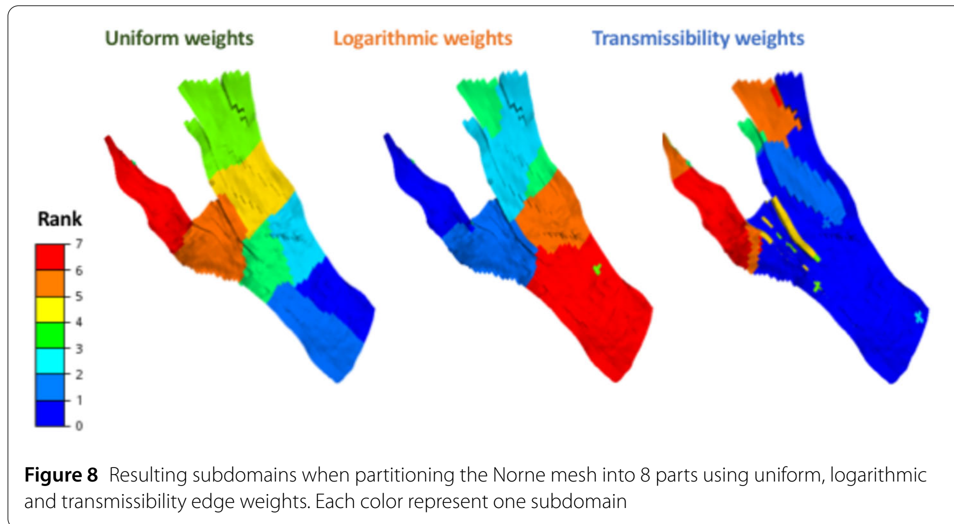

Figure 8 Resulting subdomains when partitioning the Norne mesh into 8 parts using uniform, logarithmic and transmissibility edge weights. Each color represent one subdomain

version of A_p, in the ILU subdomain solver. This leads to better convergence of the Block-Jacobi preconditioner. Note that removing ghost related non-contributing computations and improving convergence only impact the performance of the simulators linear solver. The main computational work of the reservoir simulator also consists of a system assembly part. Therefore, for example, we only achieve a speedup of around 3.5 in the $P = 128$ case, despite 2.9 times smaller iteration count, and about 2 to 3 times faster SpMV and ILU operations.

5.3 Impact of different edge-weighting strategies

In this subsection we will study and compare the different edge-weighting strategies presented in Sect. 4. We are ultimately interested in how the uniform weights (Eq. (14)), transmissibility weights (Eq. (15)) or logarithmic transmissibility weights (Eq. (16)) impact the overall performance of the reservoir simulation, but we will also consider their impact on partitioning quality and numerical effectiveness. The strategies are enforced before passing the graph to the Zoltan graph partitioner inside *Flow*, and in our experiments we have used a 5% imbalance tolerance ($\epsilon = 1.05$). All simulation results in this subsection are attained with the improved *Flow* where we use the ghost-last procedures described in Sect. 3. Figure 8 displays a visualization of a $P = 8$ partitioning of the Norne mesh for the different edge-weighting strategies. We observe that subdomains generated by the logarithmic and transmissibility edge-weighted partitioning schemes are less clean cut and more disconnected, than the subdomains resulting from the uniform edge-weighted partitioning scheme.

5.3.1 Mesh partitioning quality

We start our tests by focusing on how the different edge-weighting strategies affect partitioning quality, when used to partition the original and refined Norne meshes. In Fig. 9 we report the total communication volume for the three partitioning schemes. The total communication volume is equal to the sum of the DoFs associated with ghost cells over all processes, $3\sum_p N_p^G$, multiplied with the data size of double precision floats, which is 8 bytes. The partitioning results for the original Norne mesh are displayed in Fig. 9a, whereas Fig. 9b is for the refined Norne mesh.

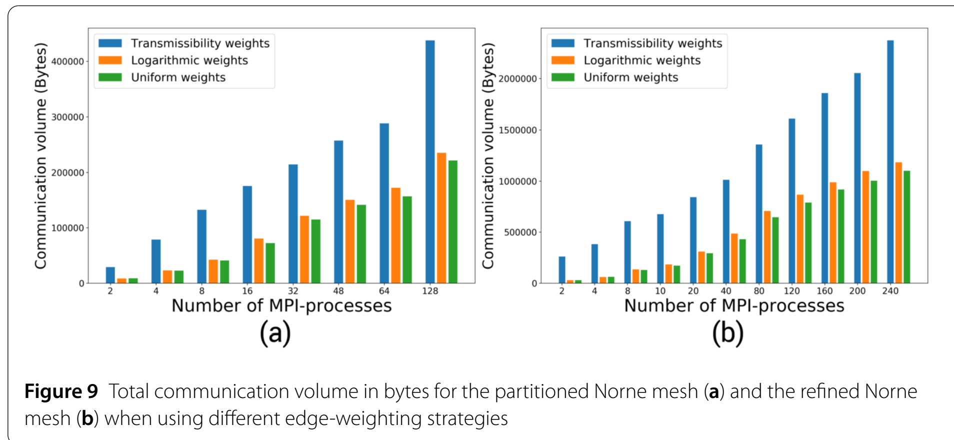

Figure 9 Total communication volume in bytes for the partitioned Norne mesh (**a**) and the refined Norne mesh (**b**) when using different edge-weighting strategies

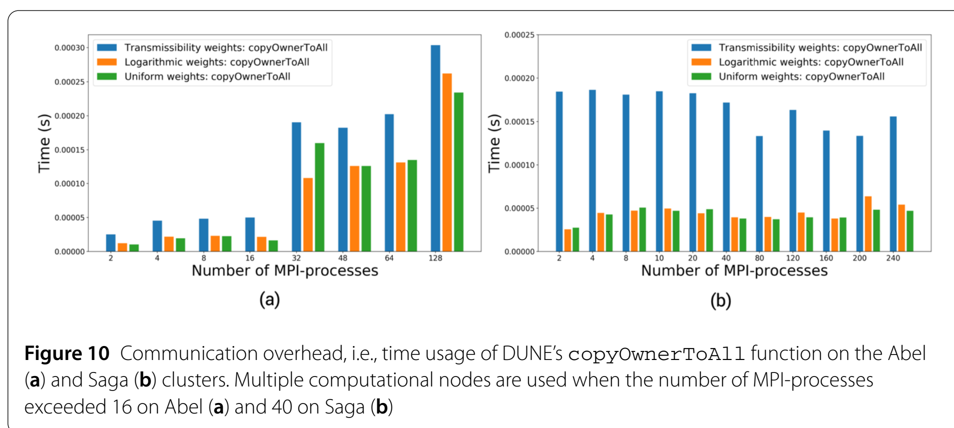

Figure 10 Communication overhead, i.e., time usage of DUNE's `copyOwnerToAll` function on the Abel (**a**) and Saga (**b**) clusters. Multiple computational nodes are used when the number of MPI-processes exceeded 16 on Abel (**a**) and 40 on Saga (**b**)

In the plots of Fig. 9 we observe that the partitions obtained using the transmissibility edge-weights yield significantly higher communication volume than the two other alternatives. This holds for both the original and the refined Norne models, and for all counts of MPI-processes P. We also notice that although the uniform edge-weighting strategy outperforms the logarithmic scheme, the differences in the resulting communication volume are relatively small.

To precisely measure how the difference in communication volume affects the actual communication overhead, we consider the execution time of the MPI data transfer operations required to perform before a parallel SpMV. The data transfer operations are implemented in the DUNE function `copyOwnerToAll`. In Fig. 10 we present the execution time of `copyOwnerToAll` related to the original and refined Norne meshes on the Abel and Saga clusters, corresponding to the communication volumes shown in Fig. 9.

The plots in Fig. 10 demonstrate that using transmissibility edge-weights yields larger communication overhead than the uniform and logarithmic alternatives on both the Abel and Saga clusters. The relatively high execution time of `copyOwnerToAll`, resulting from the transmissibility edge-weighted partitioning scheme, can partially be explained by the communication volume displayed in Fig. 9. However, the `copyOwnerToAll` execution time is also affected by the hardware. For example, the jump in execution time between $P = 16$ and $P = 32$, observed in Fig. 10a, occurs when we start using two instead of one computational node, and there is thus inter-node communication over the network.

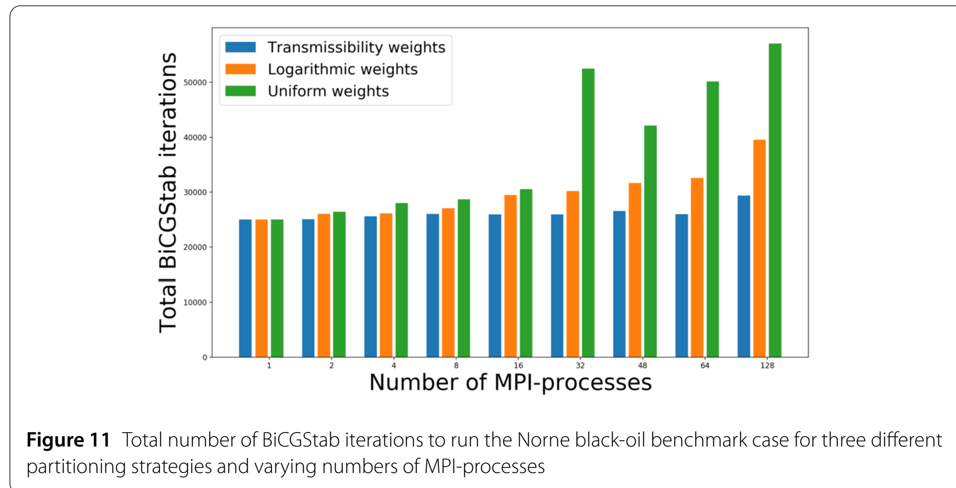

Figure 11 Total number of BiCGStab iterations to run the Norne black-oil benchmark case for three different partitioning strategies and varying numbers of MPI-processes

A similar jump in execution time is not observed in Fig. 10b on the Saga cluster between $P = 40$ and $P = 80$, because the Saga interconnect is better than the Abel interconnect (100 Gbits vs. 56 Gbits).

5.3.2 Numerical and overall performance

We can find the impact of the edge-weighting schemes on *Flow*'s numerical performance by looking at the total number of Block-Jacobi/ILU0 preconditioned BiCGStab iterations needed to complete each simulation of the original Norne benchmark case. This iteration count is displayed in Fig. 11 for the three edge-weighting strategies.

One interesting finding from Fig. 11 is that the transmissibility weighted partitioning strategy can keep the number of BiCGStab iterations almost completely independent of number of subdomains. It means that the transmissibility edge weights are indeed good for the convergence of the parallel Block-Jacobi preconditioner. On the opposite side, the uniform edge-weighting scheme leads to a large increase in the BiCGStab iterations, when the number of subdomains is large. This is contrary to its ability of keeping the communication overhead low. The logarithmic transmissibility edge-weighting scheme seems a good compromise, which is confirmed by Fig. 12 showing the total simulation time.

In Fig. 12 we observe that simulations using the logarithmic edge-weight strategy outperforms simulations using the transmissibility and uniform edge-weights for all numbers of processes. Although significant, the improvements achieved by logarithmic weights in comparison to transmissibility weights are modest in absolute terms. However, the relative improvement in execution time increase with the number of processes involved. For 2 processes we observe a 4.7% reduction in simulation execution time. For 48 processes the reduction is 24.5%. The BiCGStab iterations count required to complete the simulation of the Norne case is significantly higher when using logarithmic and uniform edge-weights instead of transmissibility edge-weights, especially when using more than 16 processes. Despite higher iteration counts, logarithmic and uniform edge-weights yield equally good or better performance than transmissibility edge-weights. There are two reasons for this. First, as demonstrated in Fig. 10, using logarithmic and uniform edge-weights results in lower communication overhead, which results in lower execution time per BiCGStab iteration. Second, the uniform and logarithmic edge-weights result in a lower number of per-process ghost cells. This has a positive impact on system assembly performance.

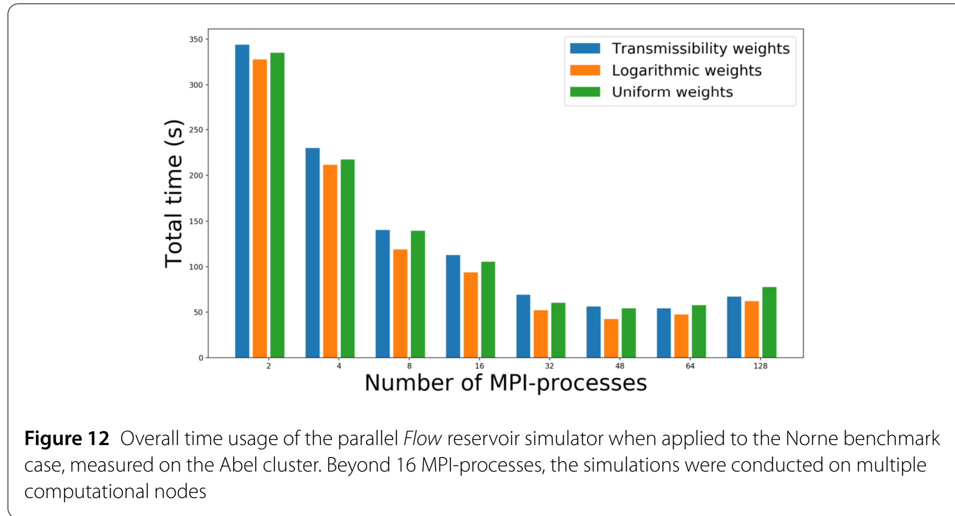

Figure 12 Overall time usage of the parallel *Flow* reservoir simulator when applied to the Norne benchmark case, measured on the Abel cluster. Beyond 16 MPI-processes, the simulations were conducted on multiple computational nodes

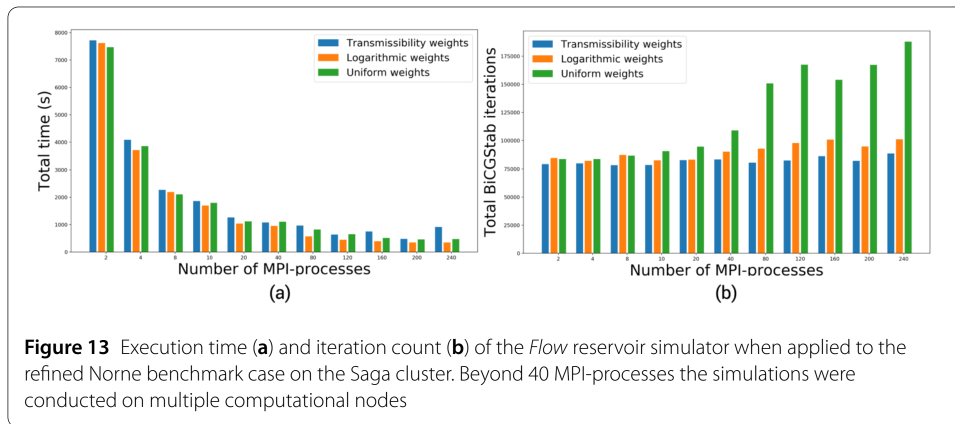

Figure 13 Execution time (**a**) and iteration count (**b**) of the *Flow* reservoir simulator when applied to the refined Norne benchmark case on the Saga cluster. Beyond 40 MPI-processes the simulations were conducted on multiple computational nodes

We also notice little or no reduction in execution time beyond 48 processes for all edge-weighting strategies. From $P = 64$ to $P = 128$ we even see an increase in simulation execution time. This is not unexpected, because at this point the mesh partitioning produces an average of only $44{,}420/128 \approx 347$ cells per process, which correspond to around 1041 DoFs. When DoFs per process reaches this point we are beyond the strong scaling limit, so adding more hardware resources yields no benefit.

Performance results for the refined Norne model, measured on the Saga cluster, are displayed in Fig. 13. Here, we have used $P = 2, 4, 8, 10, 20, 40$ MPI-processes on a single compute node, as well as $P = 80, 120, 160, 200, 240$ MPI-processes on two to six nodes. Execution times are presented in Fig. 13a and BiCGStab iterations in Fig. 13b.

The results for the refined Norne model attained on the Saga cluster, displayed in Fig. 13, are similar to the Norne results on the Abel cluster. Transmissibility edge-weights yield better linear solver convergence than the logarithmic and uniform alternatives, but the overall performance improves when using logarithmic and uniform edge-weights. We observe that simulations ran with logarithmic edge-weights had the lowest execution time for all number of MPI-processes except $P = 2$ and $P = 8$. The improvement over pure transmissibility edge-weights again appears quite modest. However, the benefit increases with the number of processes. For $P = 2, 4$ and 8 logarithmic transmissibility edge-weights yield

a 1.3%, 9.1% and 3.6%, reduction in execution time, while for $P = 80$, 120 and 160 the improvement is respectively 40.1%, 29.5% and 47.6%.

In Fig. 13a we observe an expected diminishing parallel efficiency for an increasing number of MPI-processes. Simulation execution time increases for all edge-weight schemes between 200 and 240 processes. At $P = 240$ there is around 1500 cells and 4500 DoFs per process, and we have reached the strong scaling limit.

5.4 Comparing *Flow* with industry-standard simulators

The Norne model exhibits several features rarely found in academically available data sets. It is therefore interesting to compare the performance of the improved *Flow* simulator, when applied to Norne, with commercial alternatives. We consider two industry-standard simulators, *Eclipse* 100 version 2018.2 and *Intersect* version 2019.1. For our experiments *Eclipse* and *Intersect* were used in their default configurations. All simulations presented in this subsection were done on a dual socket workstation with two Intel E5-2687W processors with a total of 16 cores available. The system has 128 GB of memory, which was sufficient for all simulations. The CPUs have a base frequency of 3.1 GHz and a turbo frequency rating of 3.8 GHz. The operating system was Red Hat 6.

We should note that the three simulators have different numerics internally. Unfortunately, only *Flow* has open source code, so we cannot investigate the implementation details of the other two. We refer the reader to the publicly available information on the numerics of the two proprietary simulators. While mpirun is handled internally by *Eclipse* and *Intersect*, we use mpirun directly for parallel simulations with *Flow*. The only other command-line option used by mpirun is `-map-by numa`. This option is particularly important for runs with two processes, since it ensures that the two are distributed on different sockets, taking advantage of cache and memory bandwidth on both. Without the option, the runtime may put both MPI processes on the same NUMA-node. Results from the previous subsection have shown that the parallel *Flow* simulator works best with logarithmic transmissibility edge-weights for the Norne case. The *Flow* simulation results presented in this subsection therefore use the logarithmic transmissibility edge-weighting scheme.

The MPI implementation in *Eclipse* is simplistic. According to the documentation it simply does domain decomposition by dividing the mesh cells evenly along one axis dimension. Nevertheless, for some models (the Norne model is actually one of them) this works well.

The comparison in parallel performance between *Flow*, *Eclipse* and *Intersect* on the Norne benchmark case is presented in Fig. 14. In this plot we also include results from simulations where multithreading is activated with two OpenMP threads per MPI process for *Flow* and *Intersect*. Multithreading is not activated when 16 MPI-processes is used, because of a lack of remaining hardware resources.

The results presented in Fig. 14 show that the parallel *Flow* simulator outperforms *Eclipse* and *Intersect*, for $P \geq 4$. Although *Eclipse* is faster than *Flow* for serial runs, *Flow* scales better than *Eclipse* on the Norne model. *Flow* achieves a speedup of 7.6 for $P = 16$ compared to *Eclipse*'s speedup of 4.4. *Intersect* performs rather poorly on Norne, but it scales better than *Eclipse*. Activating multithreading yields improved performance for both *Flow* and *Intersect*.

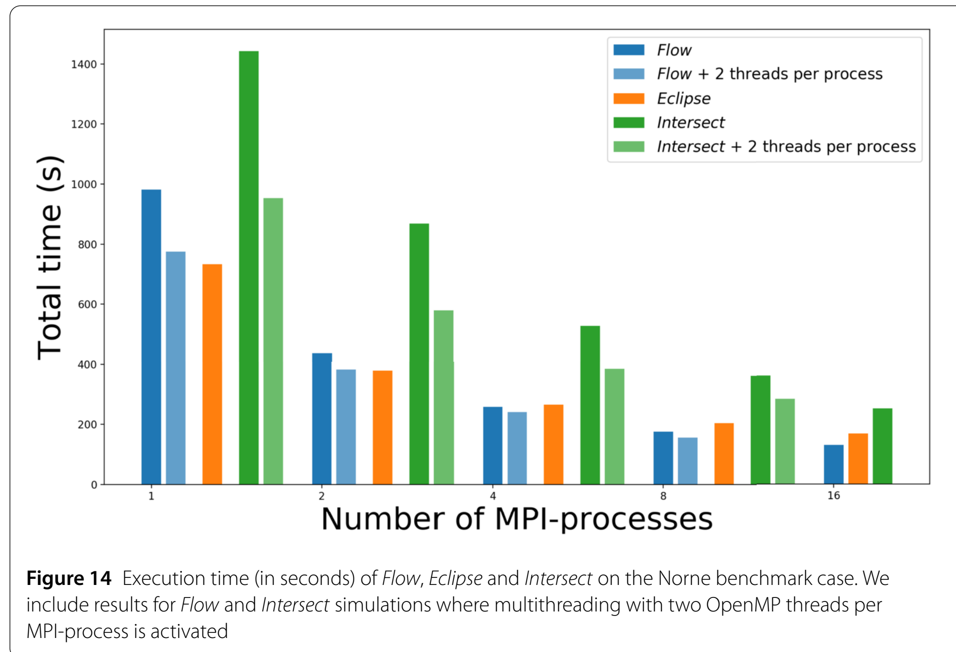

Figure 14 Execution time (in seconds) of *Flow*, *Eclipse* and *Intersect* on the Norne benchmark case. We include results for *Flow* and *Intersect* simulations where multithreading with two OpenMP threads per MPI-process is activated

6 Related work

Because of the demand for large-scale reservoir simulations in the petroleum industry, there exist several commercial and in-house simulators that are able to take advantage of parallel computing platforms. Examples include the Saudi Aramco POWERS [19–21] and GigaPOWERS [22] simulators, which are able to run simulations on reservoir grids with billions of cells.

Graph partitioning is often used to enable parallel reservoir simulation [23–26]. However, no reservoir simulation specific considerations were made in these cases. In [27] the authors suggest two guiding principles for achieving good load balancing when performing mesh partitioning for thermal reservoir simulation. First, grid cells and simulation wells should be evenly distributed between the processors. Second, if faults are present in the reservoir, they should serve as subdomain boundaries between the processes.

In the PhD thesis [28] a mesh partitioning scheme based on edge-weighted graph partitioning is described. The edge weights are formed based on the transmissibility on the interface of the cell blocks in the reservoir mesh. Additionally, the presence of wells in the reservoir is accounted for by modifying the partitioning graph. In [29] the authors derive similar strategies for partitioning non-uniform meshes in the context of computational fluid dynamics. A graph partitioning approach with edge-weights corresponding to the cell face area is implemented. The aim of this approach is to improve solver convergence, by accounting for the coefficient matrix heterogeneity introduced by the non-uniform mesh. The authors of [29] do not only focus on how this edge-weighted approach can affect the numerical performance, but also consider measures of partitioning quality, such as edges cut and number of processor neighbors. Despite poorer partitioning quality, the edge-weighted graph partitioning scheme gave the best overall performance.

Several attempts at incorporating coefficient matrix information into the partitioning of the linear systems have been made [8, 9, 30–32]. In [8] the authors add coefficient-based edge weights to the partitioning graph, and demonstrate improved numerical per-

formance in comparison with standard non-weighted schemes. The previously mentioned paper [9] presents a spectral graph partitioning scheme that outperforms standard graph partitioners, even with weighted edges, for symmetric positive definite systems with heterogeneous coefficients.

7 Conclusion

In this paper, we have given a detailed description of the domain decomposition strategy used to parallelize the simulation of fluid flow in petroleum reservoirs. We proposed an improved parallel implementation of the linear solver based on a local "ghost last" reordering of the grid cells. We also investigated the use of edge-weighted graph partitioning for dividing the reservoir mesh. A new edge-weighting scheme was devised with the purpose to maintain a balance between the numerical effectiveness of the Block-Jacobi preconditioner and the communication overhead.

Through experiments based on the Norne black-oil benchmark case, we showed that the ghost cells make up an increasing proportion of the cells in the decomposed submeshes when the number of processes increases. Further we found that removing non-contributing calculations related to these ghost cells can give a significant improvement in the parallel simulator performance.

For the Norne black-oil benchmark case, which has extremely heterogeneous petrophysical properties, using edge-weights directly derived from these properties can have negative consequences for the overall performance. Although this approach yields satisfactory numerical effectiveness, it does not make up for the increase in the communication overhead. The large communication overhead associated with the default transmissibility-weighting scheme is due to poor partitioning quality, in particular with respect to the communication volume and number of messages. The scaled logarithmic transmissibility approach, which also uses non-uniform edge-weights, yields a much better partitioning quality. Even though the numerical effectiveness due to the logarithmic scheme may be lower than the transmissibility weighted scheme, it can still result in a better overall performance.

Acknowledgements
This research was conducted using the supercomputers Abel and Saga in Norway.

Funding
This research was funded by the SIRIUS Centre for Scalable Data Access.

Abbreviations
PDE, partial differential equation; ILU, incomplete LU; OPM, open porous media; MPI, Message Passing Interface; SpMV, sparse matrix-vector multiplication; IP, inner product.

Availability of data and materials
Data and source codes are available upon reasonable request.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
All authors contributed to the writing of the manuscript. AT implemented the improved version of *Flow* and performed most of the numerical experiments. XC and ABR provided research advice and guidance. ABR performed experiments with industry-standard simulators. All authors read and approved the final manuscript.

Author details
[1] Simula Research Laboratory, Martin Linges vei 25, 1364 Fornebu, Norway. [2] University of Oslo, Oslo, Norway. [3] Equinor Research Centre, Ranheim, Norway.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 31 August 2020 Accepted: 18 June 2021 Published online: 30 June 2021

References

1. Chen Z, Huan G, Ma Y. Computational methods for multiphase flows in porous media. vol. 2. Philadelphia: SIAM; 2006.
2. Aziz K, Settari A. Petroleum reservoir simulation. London: Applied Science Publishers; 1979.
3. Peaceman DW et al. Interpretation of well-block pressures in numerical reservoir simulation (includes associated paper 6988). Soc Pet Eng J. 1978;18(03):183–94.
4. Ponting DK. Corner point geometry in reservoir simulation. In: ECMOR I - 1st European conference on the mathematics of oil recovery. 1989.
5. Lie K-A. An introduction to reservoir simulation using Matlab/GNU octave. Cambridge: Cambridge University Press; 2019.
6. Saad Y. Iterative methods for sparse linear systems. Philadelphia: SIAM; 2003.
7. Bastian P, Blatt M, Dedner A, Engwer C, Klöfkorn R, Kornhuber R, Ohlberger M, Sander O. A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE. Computing. 2008;82(2–3):121–38.
8. Cullum JK, Johnson K, Tuma M. Effects of problem decomposition (partitioning) on the rate of convergence of parallel numerical algorithms. Numer Linear Algebra Appl. 2003;10(5–6):445–65.
9. Vecharynski E, Saad Y, Sosonkina M. Graph partitioning using matrix values for preconditioning symmetric positive definite systems. SIAM J Sci Comput. 2014;36(1):63–87.
10. Karypis G, Kumar V. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN **38** (1998)
11. Pellegrini F, Roman J. Scotch: a software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In: International conference on high-performance computing and networking. Berlin: Springer; 1996. p. 493–8.
12. Boman E, Devine K, Fisk LA, Heaphy R, Hendrickson B, Vaughan C, Catalyurek U, Bozdag D, Mitchell W, Teresco J. Zoltan 3.0: parallel partitioning, load-balancing, and data management services; user's guide. Sandia National Laboratories, Albuquerque, NM. 2007;**2**(3).
13. Rasmussen AF, Sandve TH, Bao K, Lauser A, Hove J, Skaflestad B, Klöfkorn R, Blatt M, Rustad AB, Sævareid O et al. The open porous media flow reservoir simulator. Comput Math Appl. 2021;81:159–85.
14. Open Porous Media Initiative. http://opm-project.org (2017). Accessed 2017-07-26.
15. Norne reservoir simulation benchmark. https://github.com/OPM/opm-data (2019)
16. Refined Norne reservoir simulation deck. https://github.com/andrthu/opm-data/tree/refined-222-norne/refined-norne (2020)
17. The Abel computer cluster. https://www.uio.no/english/services/it/research/hpc/abel/ (2019)
18. The Saga computer cluster. https://documentation.sigma2.no/quick/saga.html (2019)
19. Dogru A, Li K, Sunaidi H, Habiballah W, Fung L, Al-Zamil N, Shin D, McDonald A, Srivastava N. A massively parallel reservoir simulator for large scale reservoir simulation. In: SPE symposium on reservoir simulation. 1999. p. 73–92.
20. Dogru AH, Sunaidi H, Fung L, Habiballah WA, Al-Zamel N, Li K et al. A parallel reservoir simulator for large-scale reservoir simulation. SPE Reserv Eval Eng. 2002;5(1):11–23.
21. Al-Shaalan TM, Fung LS, Dogru AH et al. A scalable massively parallel dual-porosity dual-permeability simulator for fractured reservoirs with super-k permeability. In: SPE annual technical conference and exhibition. Society of Petroleum Engineers; 2003.
22. Dogru AH, Fung LSK, Middya U, Al-Shaalan T, Pita JA et al. A next-generation parallel reservoir simulator for giant reservoirs. In: SPE reservoir simulation symposium. Society of Petroleum Engineers; 2009.
23. Elmroth E. On grid partitioning for a high-performance groundwater simulation software. In: Simulation and visualization on the grid. Berlin: Springer; 2000. p. 221–34.
24. Wu Y-S, Zhang K, Ding C, Pruess K, Elmroth E, Bodvarsson G. An efficient parallel-computing method for modeling nonisothermal multiphase flow and multicomponent transport in porous and fractured media. Adv Water Resour. 2002;25(3):243–61.
25. Guo X, Wang Y, Killough J. The application of static load balancers in parallel compositional reservoir simulation on distributed memory system. J Nat Gas Sci Eng. 2016;28:447–60.
26. Maliassov S, Shuttleworth R et al. Partitioners for parallelizing reservoir simulations. In: SPE reservoir simulation symposium. Society of Petroleum Engineers; 2009.
27. Ma Y, Chen Z. Parallel computation for reservoir thermal simulation of multicomponent and multiphase fluid flow. J Comput Phys. 2004;201(1):224–37.
28. Zhong H. Development of a new parallel thermal reservoir simulator. PhD thesis. University of Calgary; 2016.
29. Wang M, Xu X, Ren X, Li C, Chen J, Yang X. Mesh partitioning using matrix value approximations for parallel computational fluid dynamics simulations. Adv Mech Eng. 2017;9(11):1687814017734109.
30. Saad Y, Sosonkina M. Non-standard parallel solution strategies for distributed sparse linear systems. In: International conference of the Austrian center for parallel computation. Berlin: Springer; 1999. p. 13–27.
31. Duff IS, Kaya K. Preconditioners based on strong subgraphs. Electron Trans Numer Anal. 2013;40:225–48.
32. Janna C, Castelletto N, Ferronato M. The effect of graph partitioning techniques on parallel block FSAI preconditioning: a computational study. Numer Algorithms. 2015;68(4):813–36.