Parallel Interior-Point Solver for Block-Structured Nonlinear Programs on SIMD/GPU Architectures

François Pacaud, Michel Schanen, Sungho Shin, Daniel Adrian Maldonado, Mihai Anitescu

Parallel Interior-Point Solver for Block-Structured Nonlinear Programs on SIMD/GPU Architectures

François Pacaud[a], Michel Schanen[b], Sungho Shin[b], Daniel Adrian Maldonado[b], Mihai Anitescu[b]

[a] Centre Automatique et Systèmes, Mines Paris - PSL, Paris, France; [b] Mathematics and Computer Science Department, Argonne National Laboratory, Lemont, USA

ARTICLE HISTORY
Compiled May 10, 2023

ABSTRACT
We investigate how to port the standard interior-point method to new exascale architectures for block-structured nonlinear programs with state equations. Computationally, we decompose the interior-point algorithm into two successive operations: the evaluation of the derivatives and the solution of the associated Karush-Kuhn-Tucker (KKT) linear system. Our method accelerates both operations using two levels of parallelism. First, we distribute the computations on multiple processes using coarse parallelism. Second, each process uses a SIMD/GPU accelerator locally to accelerate the operations using fine-grained parallelism. The KKT system is reduced by eliminating the inequalities and the state variables from the corresponding equations, to a dense matrix encoding the sensitivities of the problem's degrees of freedom, drastically minimizing the memory exchange. We demonstrate the method's capability on the supercomputer Polaris, a testbed for the future exascale Aurora system. Each node is equipped with four GPUs, a setup amenable to our two-level approach. Our experiments on the stochastic optimal power flow problem show that the method can achieve a 50x speed-up compared to the state-of-the-art method.

1. Introduction

Solving complex engineering problems often resorts to the solution of large-scale block-structured nonlinear programs. As such, there has been a long interest in designing efficient nonlinear optimization algorithms, particularly by using parallel computing. Parallelism can happen at two levels. At first, *coarse parallelism* splits the program into large computational chunks, usually dispatched to multiple processors using a message-passing interface in distributed memory. In this paradigm, the parallel algorithm is designed to minimize the communication between the different processes. In a complementary direction, *fine-grained parallelism* breaks down the program into small tasks, fast to compute in shared memory. This method requires a large number of processors to be efficient, and it is usually better on SIMD architectures with low communication overhead, as provided by Graphical Processing Units (GPUs). In the mathematical optimization community, coarse parallelism has traditionally been used to solve large-scale block-structured optimization problems, as encountered in dynamic or stochastic nonlinear programs. On the contrary, fine-grained parallelism has gained attraction only recently, with the renewed interests for machine learning

applications and stochastic gradient algorithms. In this work, we combine coarse and fine-grained parallelism to solve block-structured nonlinear problems on new exascale architectures, where the solution algorithm is streamlined on different GPUs using CUDA-aware MPI.

1.1. Literature review

In his pioneering work [39, 40], Robert Schnabel identified three practical approaches to run optimization algorithms in parallel: (i) parallelize the function evaluations; (ii) parallelize the linear algebra; and (iii) parallelize the optimization algorithm itself.

The first attempt to parallelize the evaluations has been to streamline the computation of the derivatives using finite-differences [29]. Soon, it has been noted that parallelizing the forward pass in automatic differentiation (AD) is also straightforward, provided that we can propagate the tangents (encoding the first-order sensitivity) in parallel [20]. Unfortunately, doing the same in the reverse pass is not trivial, as adjoining a mutable code leads to race conditions (e.g., every read becomes a write operation). This has led to extensive research on adapting automatic differentiation to parallel environments [4, 19, 27]. Now, most state-of-the-art differentiable tools employ a Domain Specific Language (DSL) constraining the user to specific differentiable operations. In particular, this approach has been adopted mainly in machine learning, leading to the development of fast AD libraries efficiently generating the derivatives efficiently on hardware accelerators such as GPUs or TPUs [3, 32].

The parallelization of linear algebra is usually more involved, as most large-scale optimization methods fall back on the solution of sparse indefinite Karush-Kuhn-Tucker (KKT) systems [30]. In the 1980s, preliminary results were obtained by running iterative methods in parallel, using block-Krylov [36] or block-truncated Newton methods [28]. However, block iterative algorithms are quickly limited by the lack of generic preconditioners for KKT systems. The 1990s witnessed the emergence of the interior-point methods (IPM), together with the development of large-scale sparse direct linear solvers [12, 38]. In IPM, a significant portion of the time is spent solving a sequence of (indefinite) KKT systems, hence the method directly benefits from efficient sparse linear solvers able to run in parallel [1, 13]. In the 2000s, it was shown that, for block-structured optimization problems as we consider here, the layout of the optimization problem can be exploited further in a Schur complement approach to solve the Newton step in parallel [2, 9, 17, 22, 33, 44, 45]. These developments led to the development of mature decomposition-based parallel nonlinear solvers for scenario-based problems in the 2010s [8, 16, 34, 41, 46].

Eventually, running an optimization algorithm fully in parallel generally requires a subtle combination of (i) and (ii), often devolving to a software engineering problem. The challenge is to evaluate the derivatives *and* solve the resulting KKT system each in parallel; all this while minimizing the communication between the different processes. This has led to the development of different prototypes for MPI-parallel modelers [10, 21, 34, 43], most of them extending a specific AD backend [5, 14, 15]. Such approaches have been successfully applied to solve large-scale block-structured nonlinear problems, as encountered in stochastic programming and dynamic optimization.

1.2. Contributions

In this article, we introduce a new parallel algorithm to solve block-structured nonlinear programs involving state equations on exascale supercomputers. Our algorithm uses the parallel interior-point solver MadNLP [41], using two layers of parallelism to streamline both the evaluation of the derivatives and the solution of the KKT system. This framework targets new exascale supercomputers, where each node is assigned to multiple GPUs connected with a unified memory (designed to have fast memory exchange between the different GPUs).

We demonstrate the capability of the algorithm on scenario-based power flow problems (block-OPF), here formulated as two-stage stochastic nonlinear programs. The scenarios can be stochastic or represent contingencies (which can be interpreted as stochastic outcomes with uniform distribution), as is the case of the very widely used security-constrained AC optimal power flow (SC-ACOPF) problem [7]. SC-ACOPF is one of the core analyses undertaken in the planning, operational planning, and real-time operation of transmission systems [7]. SC-ACOPF is run several times a day by many operators in the US and the world. For brevity, we will refer to such problems as stochastic.

The block structure of such problems is given by the different scenarios associated with the stochastic problem, leading to potential parallelism in both the evaluation of the derivatives and the solution of the resulting block-angular KKT system. The parallel solution of the block-OPF problem with a Schur complement approach has been studied extensively both with PIPS-NLP [8, 37] (multiprocessing) and with Beltistos [23, 25] (multiprocessing + factorization of the dense Schur complement on the GPU). Compared to the state-of-the-art solver Beltistos, our approach carries out almost all computation on the GPUs including a global CUDA-aware MPI reduction, from the evaluation of the derivatives to the assembling of the Schur complement. We test our implementation on the pre-exascale supercomputer Polaris, where each node is equipped with 4 A100 GPUs, and we solve block-OPF problems with up to 9,251 nodes.

2. Problem statement

In systems engineering, it is common to encounter optimization problems with relatively few degrees of freedom – "controls". Then, the goal is to appropriately fix the values for the degrees of freedom, e.g., by minimizing a given operational cost while satisfying the physical equations of the problem. In that context, the internal state of the system is described by a *state* variable $x \in \mathbb{R}^{n_x}$, whose values depend on the current *controls* $u \in \mathbb{R}^{n_u}$ associated with the problem's degrees of freedom. If the problem is well-posed, this translates to the *state equation* $g(x, u) = 0$, where the function g exhibits the physical structure of the problem (e.g., a differential equation encoding a dynamics, or a nonlinear network flow associated with static balance equations). When the system faces uncertainties, it is often appropriate to choose a control u feasible under a finite set of conditions (or scenarios). That is, the control u must satisfy N different state equations

$$g_i(x_i, u) = 0 \quad \text{for all} \quad i = 1, \cdots, N, \tag{1}$$

where the state x_i now depend on the current scenario i. The variables x_i can be assimilated into a recourse variable. The N functions $g_1, \cdots, g_N$ define the block structure of the problem.

2.1. Block-structured nonlinear programs

In addition to satisfying the N state equations (1), we aim at minimizing the average operating costs on the N different scenarios. The corresponding problem formulates as a two-stage nonlinear program, which, in our case, is a nonlinear program with *partially separable structure* [11]:

$$\min_{\substack{x_1,\cdots,x_N,\\ u}} \sum_{i=1}^{N} f_i(x_i,u) \quad \text{s.t.} \quad \begin{cases} x_i \geq 0\,, \quad u \geq 0 \\ g_i(x_i,u) = 0\,, \\ h_i(x_i,u) \leq 0\,, \end{cases} \quad \forall i = 1,\cdots,N\,, \tag{2}$$

with $f_i : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}$, $g_i : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x}$, $h_i : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{m}$ smooth functions encoding the objective, the state equations, and the operational constraints, respectively. We note that the number of variables $(N \times n_x + n_u)$ and constraints $(N \times (m + n_x))$ are linearly proportional to the number of blocks N.

In addition, if we introduce local control variables $u_1, \cdots, u_N$ with the additional coupling constraint $u_1 = \cdots = u_N = u$, we get a problem with a separable structure, solvable using the primal decomposition method; at the expense of increasing the search space [11, 35].

By introducing slack variables $s_1, \cdots, s_N$, we rewrite (2) in standard form:

$$\min_{\substack{x_1,\cdots,x_N,\\ s_1,\cdots,s_N,\\ u}} \sum_{i=1}^{N} f_i(x_i,u) \quad \text{s.t.} \quad \begin{cases} u \geq 0\,, \quad x_i \geq 0\,, \quad s_i \geq 0 \\ g_i(x_i,u) = 0\,, \\ h_i(x_i,u) + s_i = 0\,, \end{cases} \quad \forall i = 1,\cdots,N\,. \tag{3}$$

We define $y_i \in \mathbb{R}^{n_x}$ the multipliers (or *adjoints*) associated to the equality constraints $g_i(x_i,u) = 0$, $z_i \in \mathbb{R}^m$ the multipliers associated to the operational constraints $h_i(x_i,u) + s_i = 0$, as well as λ, κ_i, ν_i the three multipliers associated to the respective bound constraints $u \geq 0, x_i \geq 0, s_i \geq 0$. The Lagrangian associated to (3) is:

$$\begin{aligned} &L(x,u,s;y,z,\lambda,\mu,\nu) := \\ &\qquad \sum_{i=1}^{N} \Big[f_i(x_i,u) + y_i^\top g_i(x_i,u) + z_i^\top \big(h_i(x_i,u) + s_i\big) - \kappa_i x_i - \nu_i s_i \Big] - \lambda u\,, \end{aligned} \tag{4}$$

with $x := (x_1, \cdots, x_N)$, $s := (s_1, \cdots, s_N)$, $y := (y_1, \cdots, y_N)$, $z := (z_1, \cdots, z_N)$. To simplify the notations, we define the *extended* objective function and the *extended* constraints:

$$f(x,u) := \sum_{i=1} f_i(x_i,u)\,, \quad g(x,u) := \begin{bmatrix} g_1(x_1,u) \\ \vdots \\ g_N(x_N,u) \end{bmatrix}, \quad h(x,u) := \begin{bmatrix} h_1(x_1,u) \\ \vdots \\ h_N(x_N,u) \end{bmatrix}.$$

We assume the functions f, g, h are twice differentiable. We denote

$$
\begin{aligned}
&H = \partial_{(x,u)} h(x,u) \in \mathbb{R}^{Nm \times (Nn_x + n_u)} && \text{Jacobian of the inequality cons.} \\
&G = \partial_{(x,u)} g(x,u) \in \mathbb{R}^{Nn_x \times (Nn_x + n_u)} && \text{Jacobian of the equality cons.} \\
&W = \nabla^2_{(x,u)} L(x,u,s;\cdot) \in \mathbb{R}^{(Nn_x + n_u) \times (Nn_x + n_u)} && \text{Hessian of Lagrangian.}
\end{aligned}
$$

2.2. Interior-point method

The interior-point method (IPM) [30, Chapter 19] is a classical approach to solve (3).

2.2.1. KKT system

The Karush-Kuhn-Tucker (KKT) equations associated to (3) can be expressed as

$$
\begin{aligned}
&\nabla_x f_i + (G^i_x)^\top y_i + (H^i_x)^\top z_i - \kappa_i = 0, && \forall i = 1, \cdots, N && \text{(5a)} \\
&\sum_{i=1}^{N} \Big(\nabla_u f_i + (G^i_u)^\top y_i + (H^i_u)^\top z_i \Big) - \lambda = 0, && \text{(coupling)} && \text{(5b)} \\
&z_i - \nu_i = 0, && \forall i = 1, \cdots, N && \text{(5c)} \\
&g_i(x_i, u) = 0, && \forall i = 1, \cdots, N && \text{(5d)} \\
&h_i(x_i, u) + s_i = 0, && \forall i = 1, \cdots, N && \text{(5e)} \\
&X_i \kappa_i = 0,\ (x_i, \kappa_i) \geq 0, && \forall i = 1, \cdots, N && \text{(5f)} \\
&S_i \nu_i = 0,\ (s_i, \nu_i) \geq 0, && \forall i = 1, \cdots, N && \text{(5g)} \\
&U\lambda = 0,\ (u, \lambda) \geq 0, && && \text{(5h)}
\end{aligned}
$$

where $U = \mathrm{diag}(u)$, $X_i = \mathrm{diag}(x_i)$, $S_i = \mathrm{diag}(s_i)$.

The interior-point method uses a homotopy parameter $\mu > 0$ to replace the complementarity constraints (5f)-(5g)-(5h) by the smooth approximations: $X_i \kappa_i = \mu e_{n_x}, S_i \nu_i = \mu e_m, U\lambda = \mu e_{n_u}$ (e_n being the vector of all ones of dimension n). The resulting (smooth) system of nonlinear equations can be solved iteratively using Newton method, where at each iteration, the descent direction is updated by solving the following *augmented* linear system:

$$
\begin{bmatrix} W + \Sigma_p & 0 & G^\top & H^\top \\ 0 & \Sigma_s & 0 & I \\ G & 0 & 0 & 0 \\ H & I & 0 & 0 \end{bmatrix} \begin{bmatrix} p_d \\ p_s \\ p_y \\ p_z \end{bmatrix} = - \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \end{bmatrix} \tag{6}
$$

with $r_1 = \begin{bmatrix} \nabla_x f + G_x^\top y + H_x^\top z - \mu X^{-1} e_{n_x} \\ \nabla_u f + G_u^\top y + H_u^\top z - \mu U^{-1} e_{n_u} \end{bmatrix}$, $r_2 = z - \mu S^{-1} e_m$, $r_3 = g(x,u)$, $r_4 = h(x,u) + s$. The primal descent direction p_d decomposes as $p_d = (p_{x_1}, \cdots, p_{x_N}, p_u)$.

2.2.2. Block angular structure

The linear system (6) is sparse and symmetric indefinite, and can be factorized using the Bunch-Kaufman algorithm. However, it is often beneficial to exploit its block-angular structure. Indeed, both the Hessian of the Lagrangian and the Jacobians have

a block-angular structure, given as

$$W = \begin{bmatrix} W_{x_1x_1} & & & W_{x_1u} \\ & \ddots & & \vdots \\ & & W_{x_Nx_N} & W_{x_Nu} \\ W_{ux_1} & \dots & W_{ux_N} & W_{uu} \end{bmatrix}, \quad G = \begin{bmatrix} G^1_{x_1} & & & G^1_u \\ & \ddots & & \vdots \\ & & G^N_{x_N} & G^N_u \end{bmatrix}.$$

By reordering the linear system (6), we can expose the block-angular structure of the KKT system as:

$$\begin{bmatrix} A_1 & & & B_1^\top \\ & \ddots & & \vdots \\ & & A_N & B_N^\top \\ B_1 & \dots & B_N & A_0 \end{bmatrix} \tag{7}$$

with

$$A_0 = W_{uu}, \quad A_i = \begin{bmatrix} W_{x_ix_i} + \Sigma_{x_i} & 0 & G_{x_i}^\top & H_{x_i}^\top \\ 0 & \Sigma_{s_i} & 0 & I \\ G_{x_i} & 0 & 0 & 0 \\ H_{x_i} & I & 0 & 0 \end{bmatrix}, \quad B_i = \begin{bmatrix} W_{x_iu} \\ (G_u^i)^\top \\ (H_u^i)^\top \end{bmatrix}^\top.$$

The block-angular structure (7) can be exploited to solve the KKT linear system in parallel using a Schur complement approach. In that case, the submatrices A_i can be factorized independently to assemble the Schur complement in parallel [8].

2.3. Condensation and reduction

Instead of reordering the augmented KKT system (6) as a block angular matrix (7), we propose an alternative approach based on successive condensation and reduction of the KKT system, following the method introduced in [31]. If the structure is well-defined, we show that we can condense the KKT system (6) to a dense matrix with size $n_u \times n_u$ in two steps: first, by removing the inequality constraints in (6), then by exploiting the structure of the equality constraints to reduce the condensed system to a dense matrix. The condensation and reduction steps are illustrated in Figure 1.

2.3.1. Condensation step

The condensation step allows reducing the size of the KKT system drastically if the number of inequality constraints is large[1].

Proposition 2.1 (Condensed KKT system)**.** *The linear system* (6) *is equivalent to*

$$\begin{bmatrix} K + \Sigma_p & G^\top \\ G & 0 \end{bmatrix} \begin{bmatrix} p_d \\ p_y \end{bmatrix} = - \begin{bmatrix} r_1 + H^\top(\Sigma_s r_4 - r_2) \\ r_3 \end{bmatrix}, \tag{8}$$

where $K \in \mathbb{R}^{(Nn_x+n_u)\times(Nn_x+n_u)}$ *is the* condensed matrix $K := W + H^\top \Sigma_s H$*. The*

[1]It is equivalent to the normal equations in linear programming [30, Chapter 16, p.412]

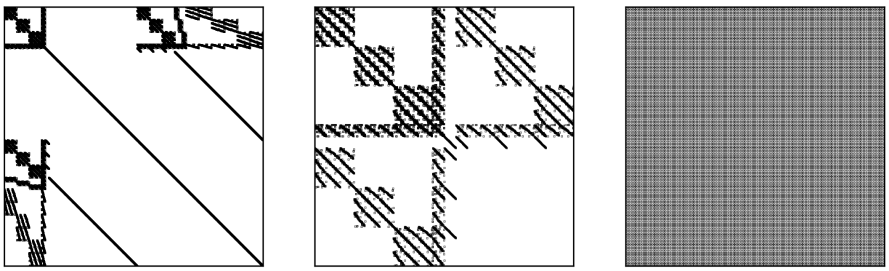

Figure 1.. Successive reductions for a block-structured nonlinear problem with $N = 3$: Augmented system (6), Condensed system (8), Reduced system (11).

descent directions p_s and p_z are recovered as

$$\begin{cases} p_z = \Sigma_s \big[H p_d + r_4 \big] - r_2 \ , \\ p_s = -\Sigma_s^{-1} \big[r_2 + p_z \big] \ . \end{cases} \tag{9}$$

Proof. See [31, Theorem 2.2]. □

The condensed matrix K inherits the block-angular structure of the Hessian of the Lagrangian W.

Proposition 2.2. *The condensed matrix $K = W + H^\top \Sigma_s H$ has a block-angular structure, given as*

$$K = \begin{bmatrix} K_{x_1 x_1} & & & K_{x_1 u} \\ & \ddots & & \vdots \\ & & K_{x_N x_N} & K_{x_N u} \\ K_{u x_1} & \dots & K_{u x_N} & K_{uu} \end{bmatrix} \tag{10}$$

where we have defined the condensed blocks $K_{x_i x_i} := W_{x_i x_i} + (H^i_{x_i})^\top \Sigma_{s_i} H^i_{x_i}$, $K_{u x_i} := W_{u x_i} + (H^i_u)^\top \Sigma_{s_i} H^i_{x_i}$ and $K_{uu} := W_{uu} + \sum_{i=1}^N (H^i_u)^\top \Sigma_{s_i} H^i_u$.

Proof. This is proved by induction. □

2.3.2. Reduction step

In addition, we can exploit the structure of the equality constraints $g_1, \cdots, g_N$ to further reduce the size of the linear system (8) down to a dense matrix with size $n_u \times n_u$. Equation (10) exhibits the structure w.r.t. the state x and the control u, we

rewrite as such the condensed KKT system (8) as

$$\begin{bmatrix} K_{x_1x_1} & & & K_{x_1u} & (G^1_{x_1})^\top & & \\ & \ddots & & \vdots & & \ddots & \\ & & K_{x_Nx_N} & K_{x_Nu} & & & (G^N_{x_N})^\top \\ K_{ux_1} & \dots & K_{ux_N} & K_{uu} & (G^1_u)^\top & \dots & (G^1_u)^\top \\ G^1_{x_1} & & & G^1_u & & & \\ & \ddots & & \vdots & & & \\ & & G^N_{x_N} & G^N_u & & & \end{bmatrix} \begin{bmatrix} p_{x_1} \\ \vdots \\ p_{x_N} \\ p_u \\ p^1_y \\ \vdots \\ p^N_y \end{bmatrix} = - \begin{bmatrix} \hat{r}^1_1 \\ \vdots \\ \hat{r}^N_1 \\ \hat{r}_2 \\ \hat{r}^1_3 \\ \vdots \\ \hat{r}^N_3 \end{bmatrix},$$

where we have renamed the right-hand-side in (8) as $\hat{r}$.

Proposition 2.3 (Reduction)**.** *Assume that for all $i = 1, \cdots, N$ the Jacobian matrices $G^i_x \in \mathbb{R}^{n_x \times n_x}$ are invertible. Then the linear system (8) is equivalent to*

$$\hat{K}_{uu}\, p_u = -\hat{r}_2 + \sum_{i=1}^{N} \left[(G^i_u)^\top (G^i_x)^{-\top} \hat{r}^i_1 + \left[K_{ux_i} - (G^i_u)^\top (G^i_x)^{-\top} K_{x_ix_i} \right] (G^i_x)^{-1} \hat{r}^i_3 \right] \quad (11)$$

with $\hat{K}_{uu} := Z^\top K Z$ and $Z \in \mathbb{R}^{(n_u + N n_x) \times n_u}$ is the reduction operator defined as

$$Z = \begin{bmatrix} -(G^1_x)^{-1} G^1_u \\ \vdots \\ -(G^N_x)^{-1} G^N_u \\ I \end{bmatrix}. \quad (12)$$

The descent directions p_x and p_y are recovered as

$$\begin{cases} p^i_x = -(G^i_x)^{-1} \left[\hat{r}^i_3 + G^i_u p_u \right] \\ p^i_y = -(G^i_x)^{-\top} \left[\hat{r}^i_1 + K_{x_ix_i} p^i_x + K_{x_iu} p_u \right]. \end{cases} \quad (13)$$

Proof. See [31, Theorem 2.1]. □

The reduction (11) is equivalent to a Schur complement approach applied to the condensed KKT system (8). In Proposition (2.1), we have shown that the condensed matrix K has a block-angular structure. The associated condensed KKT system (8) is also inheriting a block-angular structure in the form of (7), where the blocks are given by

$$A_0 = K_{uu}\,, \quad A_i = \begin{bmatrix} K_{x_ix_i} & (G^i_x)^\top \\ G^i_x & 0 \end{bmatrix}, \quad B_i = \begin{bmatrix} K_{x_iu} \\ G^i_u \end{bmatrix}^\top. \quad (14)$$

Proposition 2.4. *Assume that for each $i = 1, \cdots, N$ the Jacobian G^i_x is invertible. Let $S_{uu} = A_0 - \sum_{i=1}^N B_i A_i^{-1} B_i^\top$ be the Schur complement associated to the block-angular system (7) with the matrices (A_i, B_i) defined in (14). Then, the Schur complement S_{uu} is equal to the reduced matrix $\widehat{K}_{uu}$ defined in (11): $S_{uu} = Z^\top K Z$.*

Proof. First, note that if the Jacobian G_x^i is invertible, then the block matrix A_i defined in (14) is also invertible, with

$$A_i^{-1} = \begin{bmatrix} 0 & (G_x^i)^{-1} \\ (G_x^i)^{-\top} & -(G_x^i)^{-\top} K_{x_i x_i} (G_x^i)^{-1} \end{bmatrix} . \tag{15}$$

Using (14)-(15), we expand the expression of the terms in the sum constituting the Schur complement S_{uu}:

$$\begin{aligned} B_i A_i^{-1} B_i^\top &= \begin{bmatrix} K_{u x_i} & (G_u^i)^\top \end{bmatrix} \begin{bmatrix} 0 & (G_x^i)^{-1} \\ (G_x^i)^{-\top} & -(G_x^i)^{-\top} K_{x_i x_i} (G_x^i)^{-1} \end{bmatrix} \begin{bmatrix} K_{x_i u} \\ G_u^i \end{bmatrix} , \\ &= (G_u^i)^\top (G_x^i)^{-\top} K_{x_i u} + K_{u x_i} (G_x^i)^{-1} (G_u^i) - (G_u^i)^\top (G_x^i)^{-\top} K_{x_i x_i} (G_x^i)^{-1} G_u^i . \end{aligned}$$

Hence, the Schur complement $S_{uu} = A_0 - \sum_{i=1}^N B_i A_i^{-1} B_i^\top$ expands as

$$\begin{aligned} S_{uu} &= K_{uu} - \sum_{i=1}^N \left[(G_u^i)^\top (G_x^i)^{-\top} K_{x_i u} + K_{u x_i} (G_x^i)^{-1} (G_u^i) - (G_u^i)^\top (G_x^i)^{-\top} K_{x_i x_i} (G_x^i)^{-1} G_u^i \right] \\ &= Z^\top K Z . \end{aligned}$$

We recover the expression of the reduced matrix $\widehat{K}_{uu}$ in Proposition 2.3. □

2.4. Discussion

Hence, we can interpret the reduction step as a Schur complement approach. Forming the Schur complement has always been the bottleneck when solving distributed block angular problems in parallel [8, 26]. Its reduction operation involves large memory transfers between the processes, with the number of transfers being on the order of $\mathcal{O}(log(p))$, where p is the number of processes. Due to the quasi-shared memory architecture on GPUs, the reduction can be implemented efficiently [31]. In the next section, we propose to extend [31] to assemble the reduced matrix $\widehat{K}_{uu}$ using two levels of parallelism, using both MPI and CUDA, thus reducing the reliance on distributed memory.

3. Parallel implementation

In the previous section, we have detailed the structure of block-angular nonlinear programs and presented the condensation and reduction steps for the KKT system. The loose coupling between the blocks is favorable for parallelizing the evaluation of the derivatives and the solution of the block-angular KKT system. *Globally*, we can distribute the computation on different processes using MPI (coarse parallelism). *Locally*, we can further streamline the computation using GPU accelerators (fine-grained parallelism). This paradigm, with its two levels of parallelism, is directly in line with what is currently offered by the new exascale architectures, where each node has 4 to 8 GPUs, all sharing a unified memory for fast communication. We present in §3.1 how we streamline the evaluation of the model using automatic differentiation, and in §3.2 how we parallelize the solution of the KKT system.

3.1. Parallel automatic-differentiation

First, we present how to evaluate the model in parallel using automatic differentiation [18]. We illustrate the procedure in Figure 2. The goal of the algorithm is to streamline the evaluation of the N scenarios on N/M GPUs, M being the number of scenarios evaluated locally on each GPU (we suppose here that N is a multiple of M).

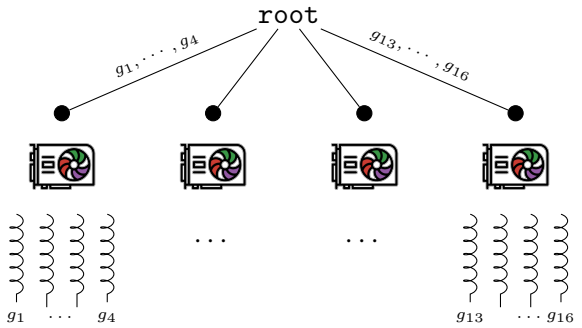

Figure 2.. Parallel evaluation of the derivatives for $g_1, \cdots g_N$ on 4 GPUs: we have a total of $N = 16$ scenarios, each GPU evaluating $M = 16/4 = 4$ scenarios locally.

3.1.1. Local parallelism

The first level of parallelism streamlines the evaluation of the model on SIMD/GPU devices. We have designed our implementation to run entirely on the GPU device, to avoid any data transfer between the host and the device.

3.1.1.1. Block evaluation. We suppose that the nonlinear functions (f_i, g_i, h_i) share the same structure, its expressions yielding the same Abstract Syntax Tree (AST) for all $i = 1, \cdots, M$. We illustrate the block evaluation on a simple abstract tree, but the reasoning extends to more complicated structures. We suppose that for all i, the functions f_i, g_i, h_i depend linearly on a nonlinear basis matrix $\psi : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_b}$: that is, there exists three *sparse* matrices L_f, L_g, L_h such that

$$f_i(x_i, u) = L_f \psi(x_i, u) \ , \quad g_i(x_i, u) = L_g \psi(x_i, u) \ , \quad h_i(x_i, u) = L_h \psi(x_i, u) \ . \tag{16}$$

Suppose we aim to evaluate the M functions $g_1, \cdots, g_M$ in batch for the states $x_1, \cdots, x_M$. The structure (16) is directly amenable for SIMD evaluation. We denote by $X_M = (x_1, \cdots, x_M) \in \mathbb{R}^{n_x \times M}$ the dense matrix obtained by concatenating the M states together. By using a proper GPU kernel or a parallel modeler, we can evaluate the basis in a SIMD fashion and build the matrix $\Psi(X_M, u) := \big(\psi(x_1, u), \cdots, \psi(x_M, u)\big) \in \mathbb{R}^{n_b \times M}$. Then, evaluating the functions $g_1, \cdots, g_M$ simultaneously translates to the evaluation of one `SpMM` product:

$$\big(g_1(x_1, u), \cdots, g_M(x_M, u)\big) = L_g \Psi(X_M, u) \in \mathbb{R}^{n_x \times M} \ . \tag{17}$$

The total memory required in the two successive operations is $\mathcal{O}((n_x + n_b) \times M)$, and depends linearly on the number of blocks M. We note the `SpMM` operations are generally implemented efficiently in the vendor library (`cusparse` for CUDA, `rocSPARSE` for AMDGPU).

3.1.1.2. First-order derivatives. Suppose that for a given i we have a differentiable implementation $\texttt{gb}_\texttt{i} : \mathbb{R}^{\texttt{n}_\texttt{d}} \to \mathbb{R}^{\texttt{n}_\texttt{x}}$ associated to the function g_i. We aim to evaluate the Jacobian-matrix products $(\nabla g_i)D$ for p tangents encoded in a matrix $D \in \mathbb{R}^{n_d \times p}$ using forward-mode AD and operator overloading. This operation translates to propagating *forward* a vector of dual numbers. Denoting by $\underline{d} \in \mathbb{D}_p^{n_d}$ the dual number encoding the p tangents stored in D, evaluating $(\nabla g_i)D$ simply amounts to call $\texttt{gb}_\texttt{i}(\underline{\texttt{d}})$ and extract the results in the dual numbers returned as a result. As G^i is sparse, we can apply the technique of Jacobian coloring [18] to compress the independent columns of the sparse matrix G^i and reduces the number of required seeding tangents p needed to evaluate the full Jacobian.

Suppose now we want to evaluate the sparse Jacobians $G^1, \cdots, G^M$ in batch. As the functions g_i are based on the same AST, their respective Jacobians $G^1, \cdots, G^M$ are sharing the same sparsity pattern. By seeding a matrix of dual numbers $\underline{D}_M = (\underline{d}_1, \cdots, \underline{d}_M) \in \mathbb{D}_p^{n_d \times M}$, we can use the same operation as (17) to streamline the evaluation of the M Jacobian-vector products using the SIMD kernel $\Psi(\cdot)$ and `SpMM` operations:

$$\big(\texttt{gb}_1(\underline{d}_1), \cdots, \texttt{gb}_M(\underline{d}_M)\big) := L_g \Psi(\underline{D}_M) \in \mathbb{D}_p^{n_x \times M} \ . \tag{18}$$

Once the results are evaluated, it remains to uncompress the dual outputs to build the M sparse Jacobians $G^1, \cdots, G^M$. Hence, we can streamline the evaluation of the Jacobian along with the number of tangents p and the number of blocks M. This comes at the expense of increasing memory usage to $\mathcal{O}((n_x + n_b + n_d) \times M \times p)$ (to store the dual matrices associated to the input, the intermediate basis Ψ and the output).

3.1.1.3. Second-order derivatives. The evaluation of the second-order derivatives follows the same procedure, using *forward-over-reverse* AD. For each i, we suppose available an adjoint function $\texttt{adj\_gb}_i : \mathbb{R}^{n_d} \times \mathbb{R}^{n_x} \to \mathbb{R}^{n_d}$ which for any primal $x \in \mathbb{R}^{n_d}$ and adjoint $y \in \mathbb{R}^{n_x}$ evaluates the Jacobian-transpose vector product $(G^i(x))^\top y$ (reverse-mode). Using forward-mode AD on top of $\texttt{adj\_gb}_i$, we can compute the second-order derivatives $y^\top \nabla^2 g_i(x) V$ for p directions V by calling $\texttt{adj\_gb}_i(\underline{x}, \underline{y})$. Using Hessian coloring, we can compress the independent columns of the sparse matrix $y^\top \nabla^2 g(x)$ and reduce the number of seeding tangents p required to evaluate the full Hessian. We note that in general obtaining an adjoint $\texttt{adj\_gb}_i$ running in parallel is nontrivial due to potential race conditions incurred by the control flow reversal of the original code.

Computing the Hessian $y^\top \nabla^2 g_i(x)$ in parallel for $i = 1, \cdots M$ amounts to defining two matrices of dual numbers $\underline{X}_M = (\underline{X}_1, \cdots, \underline{X}_M) \in \mathbb{D}_p^{n_d \times M}$, $\underline{Y}_M = (\underline{y}_1, \cdots, \underline{y}_M) \in \mathbb{D}_p^{n_x \times M}$ and evaluate $\nabla \Psi(\underline{X}_M)^\top L_g^\top \underline{Y}_M$. The dual outputs are uncompressed to build the M sparse Hessians (as the sparsity pattern of the Hessians is different than those of the Jacobians, the matrix $\underline{X}_M$ employed here is different than the one used in (18)). The total memory required to store the duals is $\mathcal{O}((2n_x + n_d + n_b) \times M \times p)$. For more details, we refer to the vector forward mode as described in [18].

3.1.2. Global parallelism

Now, if we have several GPUs at our disposal, we can push the parallelism further by distributing the evaluations using multiprocessing and a Message Passing Interface (MPI) library. Coming back at our original problem (2), we illustrate in Figure 2 how to dispatch the evaluation of the N nonlinear constraints $g_1, \cdots, g_N$ (the same reasoning

applies to the objectives $f_1, \cdots, f_N$ and the inequality constraints $h_1, \cdots, h_N$). We use the streamlined implementation described in the previous subsection to evaluate the constraints in a batch of size M: the first GPU evaluates the constraints $g_1, \cdots, g_M$, the second GPU evaluates $g_{M+1}, \cdots, g_{2M}$, and so on. In total, the evaluation of the N constraints requires N/M GPUs (if $M = 1$, each GPU evaluate one constraint; if $M = N$, we use only one GPU evaluating all the constraints).

The implementation has been designed to minimize the communication between the different processes: each batch $g_1, \cdots, g_M$ stores the data it needs *locally*, the only data exchange with the other processes being the vector of input and the vector of output. In addition, we will see in the next section we do not have to transfer the first- and second-order information if a parallel linear solver is being used.

3.2. Parallel KKT solver

By exploiting the block-angular structure of the KKT system, we can solve the Newton step in parallel using a Schur complement approach. The challenge lies in the computation of the Schur complement matrix $S = A_0 - \sum_{i=1}^{N} B_i A_i^{-1} B_i^\top$. Each product $B_i A_i^{-1} B_i^\top$ requires the factorization of the matrix A_i and the solution of a linear system with multiple (sparse) right-hand-side $A_i^{-1} B_i$. State-of-the-art methods are evaluating the Schur complement using an *incomplete augmented factorization* applied on the auxiliary matrix $\begin{bmatrix} A_i & B_i^\top \\ B_i & 0 \end{bmatrix}$, as currently implemented in the Pardiso linear solver [33]. Here, we use an alternative approach building on the reduced KKT system §2.3.2 (equivalent to the Schur complement approach). As the reduction can be streamlined on GPU accelerators [31], this approach can assemble the Schur complement in parallel using CUDA-aware MPI. We illustrate the parallel computation of the Schur complement in Figure 3.

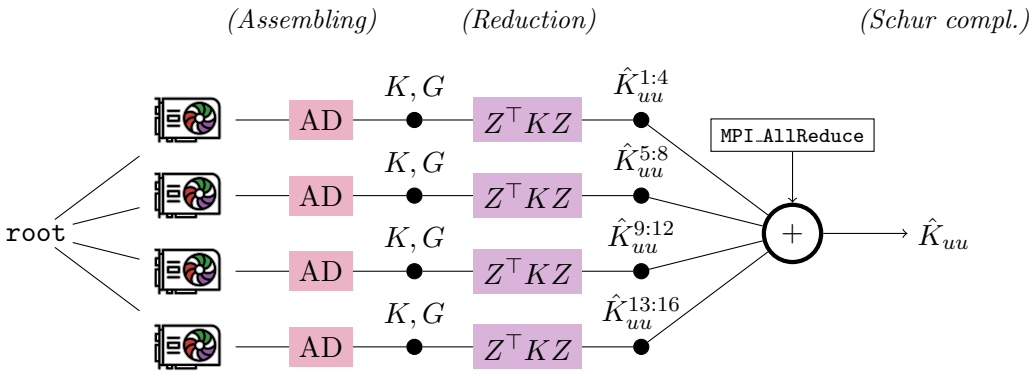

Figure 3.. Parallel computation of the Schur complement.

Assembling the sparse matrices. Using the procedure introduced in §3.1.1, we evaluate *locally* the Jacobians $G^1, \cdots, G^M$ the Jacobians $H^1, \cdots, H^M$ and the Hessians $W^1, \cdots, W^M$. Using dedicated kernels, we uncompress the results in the block-

angular sparse Jacobians

$$G_x^{1:M} = \begin{bmatrix} G_{x_1}^1 & & \\ & \ddots & \\ & & G_{x_M}^M \end{bmatrix},\ G_u^{1:M} = \begin{bmatrix} G_u^1 \\ \vdots \\ G_u^M \end{bmatrix},\ H^{1:M} = \begin{bmatrix} H_{x_1}^1 & & & H_u^1 \\ & \ddots & & \vdots \\ & & H_{x_M}^M & H_u^M \end{bmatrix},$$

and sparse Hessian

$$W^{1:M} = \begin{bmatrix} W_{x_1x_1} & & & W_{x_1u} \\ & \ddots & & \vdots \\ & & W_{x_Mx_M} & W_{x_Mu} \\ W_{ux_1} & \dots & W_{ux_M} & W_{uu} \end{bmatrix}.$$

Once the sparse matrices are obtained, we recover the condensed matrix $K^{1:M} = W^{1:M} + (H^{1:M})^\top \Sigma (H^{1:M})$ (Proposition 2.1) using one `SpGEMM` operation and we factorize the matrix $G_x^{1:M}$ using a sparse LU factorization (potentially running in batch as the matrices $G_{x_1}^1, \cdots, G_{x_M}^M$ are sharing the same sparsity pattern). Once the matrix $G_x^{1:M}$ is factorized as $PG_x^{1:M}Q = LU$ (P, Q being two permutation matrices), computing $(G_x^{1:M})^{-1}b$ translates to two backsolves (`SpSV`) and two matrix-vector multiplications (`SpMV`), as $(G_x^{1:M})^{-1}b = QU^{-1}L^{-1}Pb$.

Local reduction. Once the sparse matrices are built, we evaluate locally the reduced matrix $\widehat{K}_{uu}^{1:M}$ on the GPU, using $\mathrm{div}(n_u, n_{batch}) + 1$ matrix-matrix product $\widehat{K}_{uu}^{1:M}V$ (with $V \in \mathbb{R}^{n_u \times n_{batch}}$ a dense matrix encoding n_{batch} vectors of the Cartesian basis of $\mathbb{R}^{n_u}$). The evaluation of one batched matrix-matrix product $\widehat{K}_{uu}^{1:M}V = (Z^\top K^{1:M} Z)V$ proceeds in three steps

(1) Solve $T_x = -(G_x^{1:M})^{-1}(G_u^{1:M}V)$.
(2) Evaluate $\begin{bmatrix} L_x \\ L_u \end{bmatrix} := \begin{bmatrix} K_{xx}^{1:M} & K_{xu}^{1:M} \\ K_{ux}^{1:M} & K_{uu}^{1:M} \end{bmatrix} \begin{bmatrix} T_x \\ V \end{bmatrix}$.
(3) Set $\widehat{K}_{uu}^{1:M}V = L_u - G_u^{1:M}(G_x^{1:M})^{-\top}L_x$.

In total, we need 2 `SpSM` and 3 `SpMM` operations in the first step, 1 `SpMM` in the second step, and 2 `SpSM` and 3 `SpMM` operations in the third step, giving a total of 4 `SpSM` and 7 `SpMM` operations. More than the computation, the reduction is limited by the memory, as we have to store the three buffers L_x, T_x, T_u with a total size of $(2M \times n_x + n_u) \times n_{batch}$. If n_x is too large, it is in our interest to reduce M (by using more GPUs) or to reduce n_{batch} (at the expense of computing more matrix-matrix product $\widehat{K}_{uu}^{1:M}V$).

Global reduction. Once we obtain the locally reduced matrices $\widehat{K}_{uu}^{nM+1:(n+1)M}$ for $n = 0, \cdots, N/M - 1$, we can assemble the global reduced matrix $\widehat{K}_{uu} = \sum_{n=0}^{N/M-1} \widehat{K}_{uu}^{nM+1:(n+1)M}$ using one *all reduce* (`MPI_Allreduce`) operation. The size of the reduced matrix $\widehat{K}_{uu}$ is $n_u \times n_u$, hence limiting the memory transfer required in the algorithm.

3.3. Discussion

We have presented a practical way to assemble the Schur complement on multi-GPU architectures. The parallelism occurs both at the *local* level (SIMD evaluations on the GPUs) and at the *global* level (distributed computation with MPI). The algorithm has the advantage of assembling the sparse Jacobians and Hessians only locally, as the reduction occurs before proceeding to the memory transfer with `MPI_Allreduce`. The reduced matrix has a dimension $n_u \times n_u$, which compresses the memory transfer significantly if the number of degrees of freedom n_u is small. However, this comes at the expense of storing a vector of dual numbers (whose memory is linearly proportional to the number of blocks M evaluated locally and the number of tangents p being employed to evaluate the sparse derivatives) and additional buffers in the reduction algorithm. In the next section, we will test an implementation of the algorithm on CUDA GPUs, and show that the algorithm is practical.

4. Numerical results

We demonstrate the capabilities of the algorithm we introduced in Section §3 on the supercomputer Polaris, using CUDA-aware MPI to dispatch the solution on multiple GPUs. We present in §4.1 the stochastic optimal power flow problem, and give in §4.2 detailed assessments of the algorithms we have introduced earlier in §3. Eventually, we present in §4.3 a benchmark comparing our parallel solution algorithm with a state-of-the-art solution method running on the CPU.

4.1. Settings

4.1.1. Case study: the block-structured optimal power flow

The stochastic optimal power flow problem aims at finding an optimal dispatch for the generators u. The solution u should minimize the operational costs while satisfying the physical constraints (power flow equations $g(x, u) = 0$, here playing the role of the state equations) and operational constraints (line flow constraints $h(x, u) \leq 0$) on a given set of scenarios. Each scenario is assigned given load parameters (energy demands) and potential contingencies (line tripping). The values of the state x depend on the local scenario we are in, the state x being the *recourse* variable in our case. As such, the problem has a partially separable structure as introduced in Problem (2), the control u being shared across all scenarios. We refer to [7] for the original presentation of the stochastic optimal power flow problem and to [8, 23, 24, 26] for practical algorithms solving the stochastic optimal power flow problem (some also focus on the multistage setting, which is not covered in this article). For our benchmark, we look at reference instances provided by MATPOWER [47], whose characteristics are detailed in Table 1. We recall that in our case, the size of the Schur complement matrix $\hat{K}_{uu}$ is given by the number of controls n_u.

4.1.2. Implementation

The algorithm has been implemented entirely in Julia 1.8. The Schur complement approach has been developed as an extension of the nonlinear optimization solver MadNLP [41], using CUDA-aware MPI as provided in [6]. We have used the package

Name	#bus	#lines	#gen	n_x	n_u
case118	118	186	54	181	107
case1354pegase	1,354	1,991	260	2,447	519
case2869pegase	2,869	4,582	510	5,227	1,019
case9241pegase	9,241	16,049	1,445	17,036	2,889

Table 1.. MATPOWER instances used in the benchmark.

ExaPF as a nonlinear modeler for the optimal power flow problem. All the results presented here have been generated on the supercomputer Polaris equipped with a total of 560 nodes, each node having with 1 CPU and 4 A100 GPUs.

4.2. Assessment of the parallel implementation

4.2.1. Assessing the performance of the parallel automatic differentation

We first assess the performance of the parallel automatic differentiation we introduced in §3.1 in a multi-GPU setting. We compare the performance we obtain with a CPU implementation. We use `case1354pegase` as a representative instance, and display the time spent in the automatic differentiation as we increase the total number of scenarios N. The results are displayed in Figure 4.

We observe that the computation time depends linearly on the number of scenarios, as expected. For $N = 8$, it is not worthwhile dispatching the evaluation on multiple GPUs as the problem is small enough to be evaluated on a single GPU. For $N = 512$, the evaluation time is 12.3s on the CPU, compared to 0.50, 0.41, 0.31, and 0.28s using 1, 2, 4 and 8 GPUs, respectively. Hence, we get a 40x speed-up when evaluating the derivatives in a multi-GPU setting, and it is not worthwhile to use more than 4 GPUs (one node).

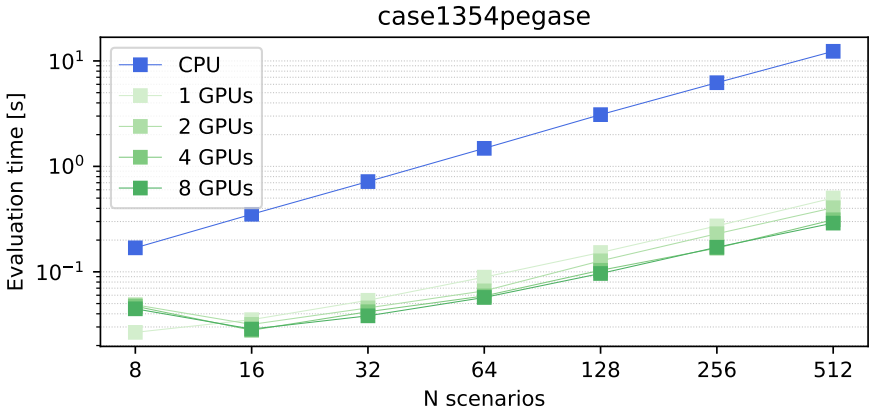

Figure 4.. Time spent to evaluate the model and its derivatives with automatic differentiation.

4.2.2. Assessing the performance of the parallel KKT solver

We proceed to the same performance analysis to assess the performance of the parallel KKT solver detailed in §3.2. We compare the time required to evaluate the full solution of the KKT system afresh (including reduction time, factorization time and backsolve time) on `case1354pegase` as we increase the number of scenarios N. As a reference, we give the time taken by the sparse linear solvers HSL MA27 (single-threaded) and HSL MA57 (multi-threaded). The results are displayed in Figure 5.

On the left, we display the evolution of the time spent in the linear solver as we increase the number of scenarios. For $N = 512$, we observe that we get a linear speed-up as we increase the number of GPUs: using 8 GPUs, the parallel KKT solver is 40x faster than using HSL MA27 on the CPU. Interestingly, we observe that HSL MA57 is not faster than HSL MA27, despite being multithreaded. This is consistent with the observation made in [42], and illustrates the difficulty of parallelizing effectively the sparse LDL factorization (Bunch-Kaufman). On the right, we display a performance profile detailing the time spent in MA27 and the parallel KKT solver on `case1354pegase` with $N = 512$ scenarios. We observe that most of the time in HSL MA27 is spent on factorizing the sparse augmented KKT system (6). On the other side, the factorization of the dense reduced matrix $\hat{K}_{uu}$ is trivial using LAPACK on the GPU; the bottleneck in the parallel KKT solver is the reduction algorithm itself. Fortunately, the reduction algorithm can run in parallel: we get a linear speed-up as we increase the number of GPUs used in the reduction algorithm.

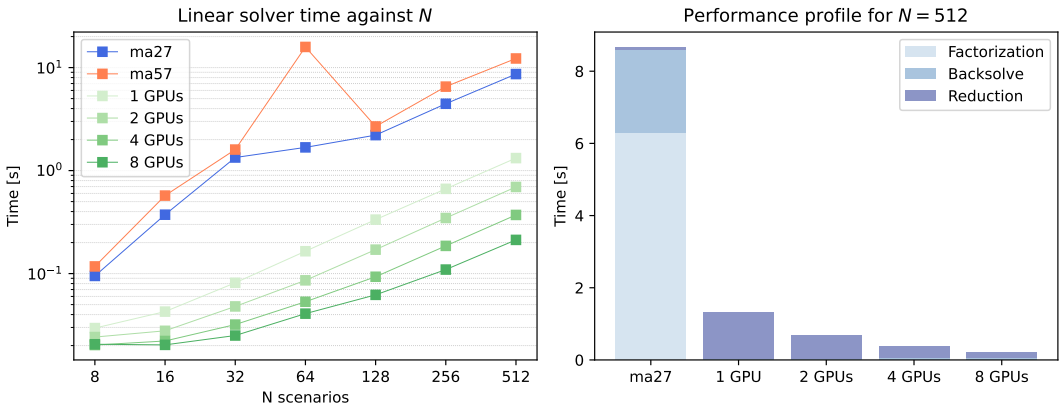

Figure 5.. Time spent to solve the KKT system for `case1354pegase`.

4.2.3. Assessing the memory consumption

We have observed in §3.1 that the total memory required to store the duals is $\mathcal{O}((2n_x + n_d + n_b) \times M \times p)$, with M being the number of scenarios stored locally ($M = N$ on 1 GPU, $M = N/2$ on 2 GPUs) and p the number of tangents. We display in Table 2 the memory taken by the automatic differentiation backend and by the parallel KKT solver for `case1354pegase` as we increase the number of scenarios N. We note that storing the duals is expensive in terms of memory, with up to 10.9GB for $N = 512$ on one GPU (as a reference, each NVIDIA A100 GPU on Polaris has 40GB of memory available). By evaluating the model on different processes with MPI, we can split the memory consumption on the different GPUs we are using, leading to better use of the resource at our disposal.

	1 GPU		2 GPUs	
N	AD	KKT solver	AD	KKT solver
8	171.1	92.3	85.5	48.1
16	342.2	181.5	171.1	93.1
32	684.3	360.0	342.2	183.2
64	1,368.7	716.8	684.3	363.2
128	2,737.3	1,430.5	1,368.7	723.4
256	5,474.7	2,858.0	2,737.3	1,443.6
512	10,949.3	5,712.8	5,474.7	2,884.1

Table 2.. Memory consumption in MB

4.3. Parallel solution of the block-structured OPF problem

We analyze the parallel performance of our implementation on block-structured OPF problems.

4.3.1. Assessing the parallel performance w.r.t. the number of scenarios

First, we are interested in the scaling of the parallel algorithm in relation to the total number of scenarios N. We consider the `case118` instance, and increase the number of scenarios N from 8 up to 2,048. For each N, we solve the block-structured OPF problem with MadNLP using our parallel KKT solver, and we compare with the performance we obtained with HSL MA27. The results are displayed in Figure 6. We observe that the solver HSL MA27 is initially faster than our parallel KKT solver, as the problem is too small to benefit from parallelism. However, as soon as $N \geq 16$ the parallel KKT solver becomes competitive with HSL MA27. The relative performance is improving as we increase the number of scenarios N: for $N = 512$, we get a 68x speed-up when using 8 GPUs, compared to the reference HSL MA27 (10.4s versus 712s). Interestingly, using 2 nodes (=8 GPUs) does not lead to any speed-up compared to a single node (=4 GPUs) if $N \leq 256$; this setting is attractive only when the size of the problem becomes sufficiently large ($N \geq 1024$) to compensate for the additional memory exchange.

4.3.2. Assessing the parallel performance w.r.t. the size of the problem

Second, we increase the size of the problems. We set a fixed number of scenarios $N = 8$, and look at the time to solution for `case1354pegase`, `case2869pegase` and `case9241pegase`. We detail the respective dimension of each problem in Table 3. We display the results in Figure 7, and give the detailed benchmark in Table 4. On the left (a), we display the total time required to find the solution of the three instances as a function of the number of GPUs; on the right (b), we show the performance profile associated to `case9241pegase`. In (a), we observe that overall the parallel algorithm is faster than the CPU implementation. The parallel algorithm scales well as we increase the number of GPUs we are using, the parallel algorithm being 35x faster than the reference when using 8 GPUs to solve `case9241pegase`. In (b), we detail the time

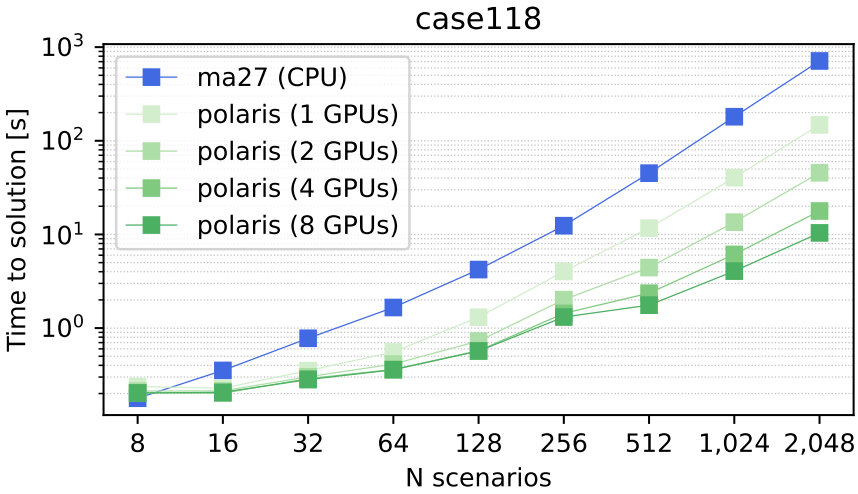

Figure 6.. Time to solve the block-structured OPF problem `case118` as a function of the number of scenarios N.

spent in the different operations for `case9241pegase`: the time spent to factorize the Schur complement with Lapack (using `cusolve`) is constant as the size of the Schur complement remains the same as we increase the number of GPUs. We observe that the time spent in the AD decreases linearly with the number of GPUs exploited, but the relative time spent in AD is negligible (less than 5% of the total time). Most of the time is spent in the parallel reduction, as discussed earlier in §4.2.2.

	N	`nvar`	`ncon`	$\hat{K}_{uu}$ (mb)
1354pegase	8	20,095	53,520	2.1
2869pegase	8	42,835	119,216	7.9
9241pegase	8	139,177	404,640	63.7
1354pegase	512	1,253,383	4,425,280	2.1

Table 3.. Dimension of the instances we have used in our benchmark.

4.3.3. Assessing the parallel performance on a very large-scale instance

We finish our numerical experiments by solving a very large-scale instance: `case1354pegase` with $N = 512$ scenarios. The dimension of the resulting optimization problem is displayed in Table 3: the problem has more than 1 million variables, and 4 millions constraints. We solve this instance on resp. 1 node, 2, 4 and 8 nodes (resp. 4, 8, 16 and 32 GPUs). The results are displayed in Figure 8. We observe that the scaling is almost perfect when we use 2 nodes (8 GPUs) instead of a single node (4 GPUs) but we do not observe the same behavior when we increase the number of nodes to 4 and 8. On that instance, the gain we get when using 8 nodes (32 GPUs) is marginal

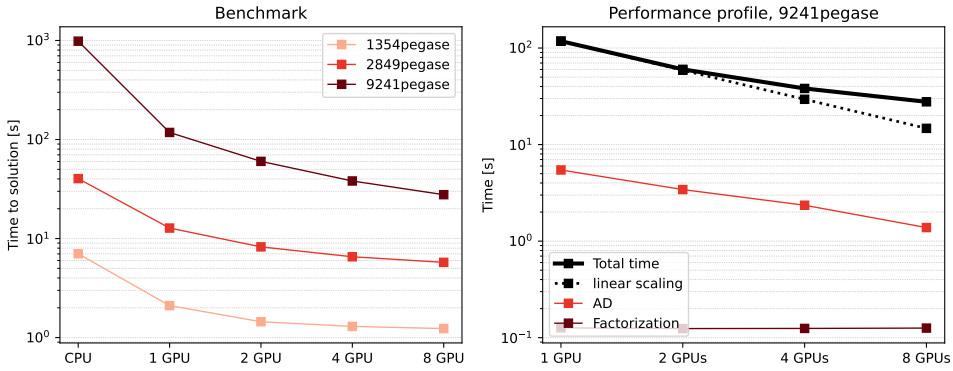

Figure 7.. For a fixed number of scenarios $N = 8$, (a) total time spent solving the block-OPF `case1354pegase`, `case2869pegase` and `case9241pegase` with MadNLP (b) performance profile for `case9241pegase` with varying number of GPUs.

	`1354pegase`				`2869pegase`				`9241pegase`			
	#it	AD	KKT	**Tot.**	#it	AD	KKT	**Tot.**	#it	AD	KKT	**Tot.**
CPU	44	2.6	4.2	**7.0**	77	11.9	27.4	**40.3**	136	205.6	771.8	**984.1**
1 GPU	44	0.3	1.8	**2.1**	93	1.1	11.7	**12.8**	98	5.5	112.3	**117.8**
2 GPUs	44	0.3	1.1	**1.4**	93	0.8	7.4	**8.2**	98	3.4	56.8	**60.2**
4 GPUs	44	0.3	1.0	**1.3**	93	0.8	5.7	**6.5**	98	2.3	35.8	**38.1**
8 GPUs	44	0.2	1.0	**1.2**	93	0.6	5.1	**5.7**	98	1.4	26.4	**27.7**

Table 4.. Detailed results

compared to when using 4 nodes (16 GPUs): the solving time only decreases from 67s to 58s. This corroborate our observations: it is better to pack all the computation on a single node to use four A100 GPUs connected together via unified memory (NVLINK has a transfer rate of 600GB/s). When we have to use more than 2 nodes, the memory transfers are more involved as they have to pass through the network of the supercomputer.

5. Conclusion

We show promising results for leveraging massively parallel SIMD architectures like GPUs for block-structured nonlinear programs. The parallelism is applied to both the derivative evaluation and the solution of the KKT linear system. The main operation in the KKT algorithm is the assembling of the Schur complement, the factorization of the dense Schur complement being fast to carry on the GPU.

At all levels, the method benefits significantly from the massive parallelism, achieving a speedup of around 40 for the derivatives compared to a sequential CPU implementation. The speedup is very application dependent, not least on the Hessian coloring and the problem's structure. The assembling of the Schur complement is bottlenecked by a distributed reduction operation bound by the interconnect's latency and throughput between GPUs. Current, so-called *super nodes* with multiple GPUs

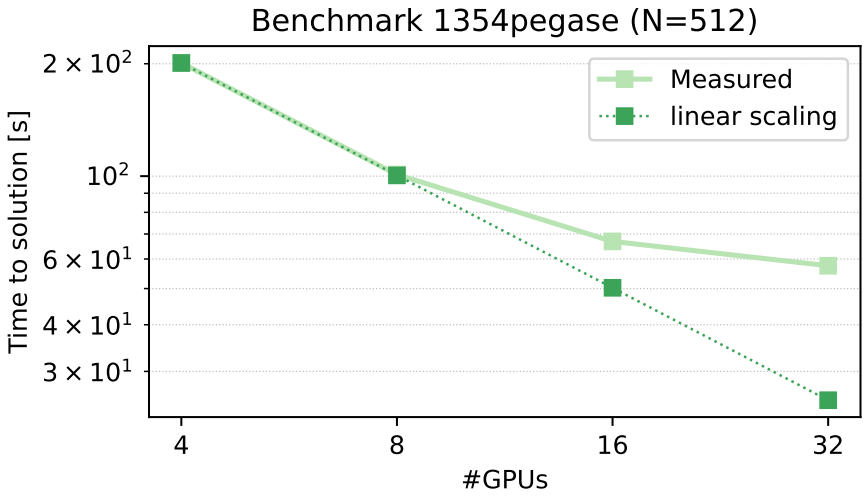

Figure 8.. Solving `case1354pegase` with $N = 512$

connected via fast networks like NVLINK greatly accelerate this operation. Lastly, our method is limited by the memory capacity of the GPU accelerators as it grows linearly with the number of problem blocks. In the context of ACOPF we are confident that upcoming GPUs will provide enough memory to solve a large number of scenarios in parallel, even for the largest grid instances (e.g., Eastern Interconnection with 70,000 nodes).

With the upcoming release of the Aurora supercomputer, these SIMD architectures will allow new science in regimes that were impossible with previous CPU architectures.

Acknowledgment

This material was based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR) under Contract DE-AC02-06CH11347 and by NSF through award CNS-1545046. The authors gratefully acknowledge the funding support from the Applied Mathematics Program within the U.S. Department of Energy's (DOE) Office of Advanced Scientific Computing Research (ASCR) as part of the project ExaSGD. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[1] Amestoy, P. R., Duff, I. S., and L'excellent, J.-Y. (2000). Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer methods in applied mechanics and engineering*, 184(2-4):501–520.

[2] Birge, J. R. and Qi, L. (1988). Computing block-angular Karmarkar projections with applications to stochastic programming. *Management science*, 34(12):1472–1479.

[3] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G.,

Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs.

[4] Bücker, H. M., Lang, B., an Mey, D., and Bischof, C. H. (2001). Bringing together automatic differentiation and OpenMP. In *Proceedings of the 15th international conference on Supercomputing*, pages 246–251.

[5] Bussieck, M. R. and Meeraus, A. (2004). General algebraic modeling system (GAMS). In *Modeling languages in mathematical optimization*, pages 137–157. Springer.

[6] Byrne, S., Wilcox, L. C., and Churavy, V. (2021). MPI. jl: Julia bindings for the Message Passing Interface. In *Proceedings of the JuliaCon Conferences*, volume 1, page 68.

[7] Capitanescu, F., Ramos, J. M., Panciatici, P., Kirschen, D., Marcolini, A. M., Platbrood, L., and Wehenkel, L. (2011). State-of-the-art, challenges, and future trends in security constrained optimal power flow. *Electric power systems research*, 81(8):1731–1741.

[8] Chiang, N., Petra, C. G., and Zavala, V. M. (2014). Structured nonconvex optimization of large-scale energy systems using PIPS-NLP. In *2014 Power Systems Computation Conference*, pages 1–7. IEEE.

[9] Choi, I. C. and Goldfarb, D. (1993). Exploiting special structure in a primal—dual path-following algorithm. *Mathematical Programming*, 58(1):33–52.

[10] Colombo, M., Grothey, A., Hogg, J., Woodsend, K., and Gondzio, J. (2009). A structure-conveying modelling language for mathematical and stochastic programming. *Mathematical Programming Computation*, 1(4):223–247.

[11] DeMiguel, V. and Nogales, F. J. (2008). On decomposition methods for a class of partially separable nonlinear programs. *Mathematics of Operations Research*, 33(1):119–139.

[12] Duff, I. S. (2004). MA57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM Transactions on Mathematical Software (TOMS)*, 30(2):118–144.

[13] Duff, I. S. and Van Der Vorst, H. A. (1999). Developments and trends in the parallel solution of linear systems. *Parallel Computing*, 25(13-14):1931–1970.

[14] Dunning, I., Huchette, J., and Lubin, M. (2017). JuMP: A modeling language for mathematical optimization. *SIAM review*, 59(2):295–320.

[15] Fourer, R., Gay, D. M., and Kernighan, B. W. (1990). A modeling language for mathematical programming. *Management Science*, 36(5):519–554.

[16] Gondzio, J. and Grothey, A. (2009). Exploiting structure in parallel implementation of interior point methods for optimization. *Computational Management Science*, 6(2):135–160.

[17] Gondzio, J. and Sarkissian, R. (2003). Parallel interior-point solver for structured linear programs. *Mathematical Programming*, 96(3):561–584.

[18] Griewank, A. and Walther, A. (2008). *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM.

[19] Hovland, P. and Bischof, C. (1998). Automatic differentiation for message-passing parallel programs. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 98–104. IEEE.

[20] Hovland, P. D. (1997). *Automatic differentiation of parallel programs*. University of Illinois at Urbana-Champaign.

[21] Huchette, J., Lubin, M., and Petra, C. (2014). Parallel algebraic modeling for stochastic optimization. In *2014 First Workshop for High Performance Technical Computing in Dynamic Languages*, pages 29–35. IEEE.

[22] Jessup, E. R., Yang, D., and Zenios, S. A. (1994). Parallel factorization of structured matrices arising in stochastic programming. *SIAM journal on Optimization*, 4(4):833–846.

[23] Kardoš, J., Kourounis, D., and Schenk, O. (2019). Two-level parallel augmented Schur complement interior-point algorithms for the solution of security constrained optimal power flow problems. *IEEE Transactions on power systems*, 35(2):1340–1350.

[24] Kardoš, J., Kourounis, D., and Schenk, O. (2020). Structure-exploiting interior point methods. In *Parallel Algorithms in Computational Science and Engineering*, pages 63–93. Springer.

[25] Kardoš, J., Kourounis, D., Schenk, O., and Zimmerman, R. (2022). Beltistos: A robust interior point method for large-scale optimal power flow problems. *Electric Power Systems*

Research, 212:108613.

[26] Lubin, M., Petra, C. G., Anitescu, M., and Zavala, V. (2011). Scalable stochastic optimization of complex energy systems. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE.

[27] Moses, W. S., Churavy, V., Paehler, L., Hückelheim, J., Narayanan, S. H. K., Schanen, M., and Doerfert, J. (2021). Reverse-mode automatic differentiation and optimization of GPU kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16.

[28] Nash, S. G. and Sofer, A. (1989). Block truncated-Newton methods for parallel optimization. *Mathematical Programming*, 45(1):529–546.

[29] Nash, S. G. and Sofer, A. (1991). A general-purpose parallel algorithm for unconstrained optimization. *SIAM Journal on Optimization*, 1(4):530–547.

[30] Nocedal, J. and Wright, S. J. (2006). *Numerical optimization*. Springer series in operations research. Springer, New York, 2nd edition.

[31] Pacaud, F., Shin, S., Schanen, M., Maldonado, D. A., and Anitescu, M. (2022). Accelerating condensed interior-point methods on SIMD/GPU architectures. *arXiv preprint arXiv:2203.11875*.

[32] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

[33] Petra, C. G., Schenk, O., Lubin, M., and Gärtner, K. (2014). An augmented incomplete factorization approach for computing the Schur complement in stochastic optimization. *SIAM Journal on Scientific Computing*, 36(2):C139–C162.

[34] Rodriguez, J. S., Parker, R., Laird, C. D., Nicholson, B., Siirola, J. D., and Bynum, M. (2021). Scalable parallel nonlinear optimization with PyNumero and Parapint. *Optimization Online*.

[35] Ruszczynski, A. (1993). Interior point methods in stochastic programming. *Working paper*.

[36] Saad, Y. (1980). On the rates of convergence of the Lanczos and the block-Lanczos methods. *SIAM Journal on Numerical Analysis*, 17(5):687–706.

[37] Schanen, M., Gilbert, F., Petra, C. G., and Anitescu, M. (2018). Toward multiperiod ac-based contingency constrained optimal power flow at large scale. In *2018 Power Systems Computation Conference (PSCC)*, pages 1–7. IEEE.

[38] Schenk, O. and Gärtner, K. (2004). Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*, 20(3):475–487.

[39] Schnabel, R. B. (1985). Parallel computing in optimization. In *Computational Mathematical Programming*, pages 357–381. Springer.

[40] Schnabel, R. B. (1995). A view of the limitations, opportunities, and challenges in parallel nonlinear optimization. *Parallel computing*, 21(6):875–905.

[41] Shin, S., Coffrin, C., Sundar, K., and Zavala, V. M. (2021). Graph-based modeling and decomposition of energy infrastructures. *IFAC-PapersOnLine*, 54(3):693–698.

[42] Tasseff, B., Coffrin, C., Wächter, A., and Laird, C. (2019). Exploring benefits of linear solver parallelism on modern nonlinear optimization applications. *arXiv preprint arXiv:1909.08104*.

[43] Watson, J.-P., Woodruff, D. L., and Hart, W. E. (2012). PySP: modeling and solving stochastic programs in python. *Mathematical Programming Computation*, 4(2):109–149.

[44] Word, D. P., Kang, J., Akesson, J., and Laird, C. D. (2014). Efficient parallel solution of large-scale nonlinear dynamic optimization problems. *Computational Optimization and Applications*, 59(3):667–688.

[45] Zavala, V. M., Laird, C. D., and Biegler, L. T. (2008). Interior-point decomposition approaches for parallel solution of large-scale nonlinear parameter estimation problems. *Chemical Engineering Science*, 63(19):4834–4845.

[46] Zhu, Y., Word, D., Siirola, J., and Laird, C. D. (2009). Exploiting modern computing architectures for efficient large-scale nonlinear programming. In *Computer Aided Chemical*

Engineering, volume 27, pages 783–788. Elsevier.

[47] Zimmerman, R. D., Murillo-Sánchez, C. E., and Thomas, R. J. (2010). MATPOWER: Steady-state operations, planning, and analysis tools for power systems research and education. *IEEE Transactions on Power Systems*, 26(1):12–19.