Domain Decomposition Preconditioners for Unstructured Network Problems in Parallel Vector Architectures

Daniel Adrian Maldonado
with Francois Pacaud, Michel Schanen, Mihai Anitescu

Mathematics and Computing Science
Argonne National Laboratory

August, 2021

Argonne NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY | Office of Science

Motivation

- We are concerned with optimizing the power grid.

$$\begin{aligned} \underset{u}{\text{minimize}} \quad & f(x,u) \\ \text{subject to} \quad & g(x,u) = 0, \\ & h(x,u) \leq 0\,. \end{aligned}$$

- What are good ways to solve this on a GPU?
- Here, $g(x,u)$ represents the Network equations.
- We have developed a **reduced method**: $d_u f = -f_x g_x^{-1} g_p$.

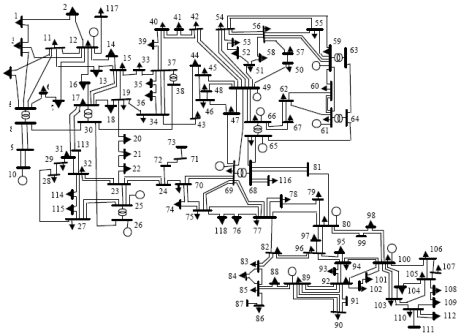


Figure: Node Network by AndersonS7 under CC-SA-BY 4.0. https://commons.wikimedia.org/wiki/File:Node_Network.gif

The power flow problem

- We put a price ($) to power (MW). We define *AC power* as $s_i = p_i + jq_i = v\bar{i}$.
- This generalizes to: $\boldsymbol{s} = \boldsymbol{p} + j\boldsymbol{q} = diag(\boldsymbol{v})\left(\boldsymbol{Yv}\right)^*$.
- Or $s_i = \boldsymbol{v}^* \boldsymbol{M_i} \boldsymbol{v}$ with $\boldsymbol{M_i} = \left(\boldsymbol{G} e_i + e_i^T \boldsymbol{G}\right) + j\left(-\boldsymbol{B} e_i + e_i^T \boldsymbol{B}\right)$. With $\boldsymbol{Y} = \boldsymbol{G} + j\boldsymbol{B}$.
- For OPF we often write it in polar form:

$$p_i = v_i \sum_j^n v_j (g_{ij} \cos(\theta_i - \theta_j) + b_{ij} \sin(\theta_i - \theta_j)),$$

$$q_i = v_i \sum_j^n v_j (g_{ij} \sin(\theta_i - \theta_j) - b_{ij} \cos(\theta_i - \theta_j)).$$

Solving the power flow

- Solving the power flow equations is one of the central themes in power engineering.
- Most commmon algorithm is Newton:

$$\begin{bmatrix} \boldsymbol{p} \\ \boldsymbol{q} \end{bmatrix} = \begin{bmatrix} f(x) \\ h(x) \end{bmatrix}, \quad x = \begin{bmatrix} \theta \\ v \end{bmatrix}.$$

Where $p \in \mathbb{R}^{npv+npq}$, $q \in \mathbb{R}^{npq}$, etc. Our Jacobian matrix:

$$\boldsymbol{J}(x) = \begin{bmatrix} \boldsymbol{F}_\theta & \boldsymbol{F}_v \\ \boldsymbol{H}_\theta & \boldsymbol{H}_v \end{bmatrix}; \quad \boldsymbol{b}(x) = \begin{bmatrix} \boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{p} \\ \boldsymbol{h}(\boldsymbol{x}) - \boldsymbol{q} \end{bmatrix}; \quad \Delta \boldsymbol{x} = -\boldsymbol{J}^{-1}\boldsymbol{b}.$$

- $\boldsymbol{J}(x)$ is non-symmetric, positive-indefinite.
- Because unstructured grid topology, has no "nice" structure (e.g. banded)
- Fortunately, it is very sparse.

Solving $J\Delta x = b$

- Initially direct solvers: LU and (later) KLU.
- Krylov solvers [Semlyen, 1996], GMRES. [Flueck, 1998] preconditioned with ILU, Jacobi, Fast-Decoupled. BiCGSTAB vs GMRES [deLeon, 2002].
- Parallel block Jacobi and Additive Schwarz Method (ASM) [Abhyankar, 2013].
- GPU is well suited for block preconditioners (batched factorization/solve). Excellent if blocks are small (fine-grained parallelism)
- But the smaller the block the worse the convergence. Difficult for unstructured grids.
- In our work we implement RASM to precondition GMRES and BiCGSTAB.

Additive Schwarz Method (ASM)

- Partition the vector x of unknowns into k subsets $x_1, x_2, \ldots, x_k$.
- To each subset we can define a *restriction matrix*, R_i, such that $x_i = R_i x$.
- ASM iteration:

$$x^{n+1} = x^n + \sum_{j=1}^{k} R_j^\top A_j^{-1} R_j (b - Ax)$$

- As a pre-conditioner:

$$M_{AS}^{-1} := \sum_{j=1}^{k} R_j^\top A_j^{-1} R_j$$

- To avoid communication, RASM:

$$M_{RAS}^{-1} := \sum_{j=1}^{k} \hat{R}_j^\top A_j^{-1} R_j$$

- where $\sum_{j=1}^{k} \hat{R}_j \hat{R}_j^\top = I$.

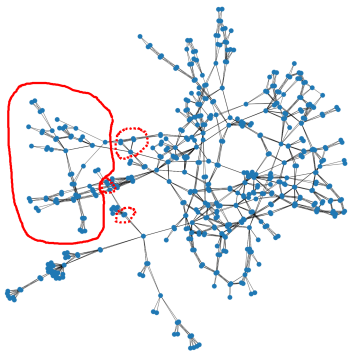
Partition and Overlap

We use METIS and then compute overlap.

```
function overlap(Graph, subset; level=1)

   subset2 = [neighbors(Graph, v) for
                 v in subset]
   subset2 = reduce(vcat, subset2)
   subset2 = unique(vcat(subset, subset2))

   level -= 1
   if level == 0
     return subset2
   else
     return overlap(Graph,
        subset2, level=level)
   end
 end
```



Implementation: *ExaPF.jl*

- Implementation in *Julia*.
- Preconditioner and solver lives in the GPU.
- We use *KernelAbstractions.jl* for fast prototyping.

```
@oneapi residual_kernel(F,...)
@cuda residual_kernel(F,...)
@roc residual_kernel(F,...)
```

```
@kernel function residual_kernel!(F, ...)
 i = @index(Global, Linear)
 fr = (i <= npv) ? pv[i] : pq[i-npv]
 F[i] -= pinj[fr]
 if i>npv
  F[i + npq] -= qinj[fr]
 end
 for c in colptr[fr]:colptr[fr+1]-1
  to  = ybus_re_rowval[c]
  aij = v_a[fr]-v_a[to]
  coef_cos = v_m[fr]*v_m[to]*ybus_re_nzval[c]
  coef_sin = v_m[fr]*v_m[to]*ybus_im_nzval[c]
  cos_val  = cos(aij)
  sin_val  = sin(aij)
  F[i] += coef_cos*cos_val+coef_sin*sin_val
  if i > npv
   F[npq + i] += coef_cos*sin_val
                 - coef_sin*cos_val
  end
 end
end
```

Implementation: *ExaPF.jl*

```
@inline function (*)(
  P::ASMPreconditioner,
  b::CuVector{Float64}
)
    n = size(b, 1)
    P.y .= 0.0
    mb_kernel! = mult_blks_gpu!(
            CUDADevice())
    ev = mb_kernel!(C.cuyaux, b,
            P.part_size, P.rest_size,
            P.part, P.blocks,
            ndrange=P.nblocks)
    wait(ev)
    return P.y
end
```

```
@kernel function mult_blks_gpu!(
  y, b, p_len, r_len, part, blocks
)
    i = @index(Global, Linear)
    plen = p_len[i]
    rlen = r_len[i]
    for j=1:rlen
        idxA = part[j,i]
        for k=1:len
            idxB = part[k,i]
            y[idxA] += blocks[j,k,i]*b[idxB]
        end
    end
end
```

Memory transfer is expensive.

Preliminary results

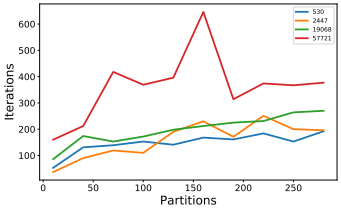
System of 30k variables. ASM with no overlap.

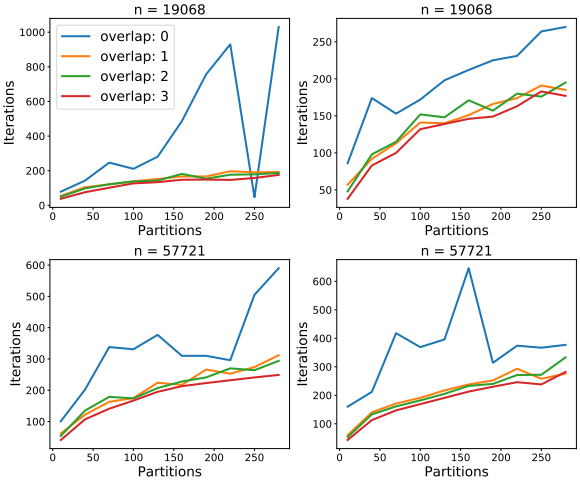
Table: GMRES(n) vs BiCGSTAB iterations

Partitions	BiCGSTAB	GMRES(3)	GMRES(100)
10	86	666	110
40	174	-	474
70	153	-	674
100	172	2046	764
130	198	-	1240
160	212	-	900
190	225	-	1581

Preliminary results

We tested several systems with and w/o overlap.



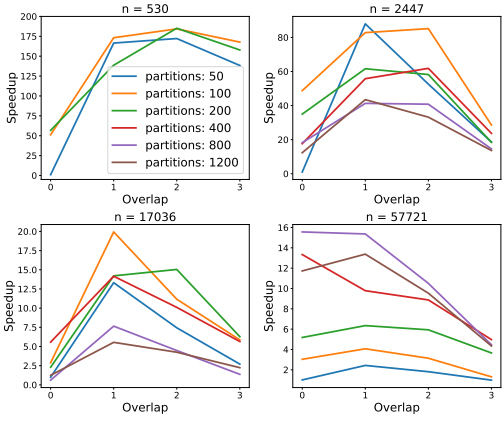


Preliminary results

- Seeking fine-grained parallelism:
- - blocks, - iters, + mvmul, + fact.
- + overlap, - iters, + mvmul, + fact.
- Many factors. Need to experiment.

# Overlap	mul. blocks	nrm2	factorize
0	50 %	10%	5%
1	50 %	5 %	20%
2	33 %	2 %	55%
3	15 %	1%	81%

Table: Percentage of total running time spent in the most important kernel, for case $n = 17036$, 400 partitions and increasing overlap.



Takeaways and future pathways

- To exploit GPUs, we would like to exploit fine-grained parallelism.
- Some iterative solvers seem to behave better (e.g. BiCGSTAB)
- Impact of ordering/partitions (METIS alternatives?)
- Other pre-conditioners? Example: Sparse Approximate Inverse (SPAI).
- Performance sometimes not intuitive.
- Optimizing solver and preconditioner: kernel launch and memory locality.

References

A. Semlyen (1996)

Fundamental concepts of a Krylov subspace power flow methodology

IEEE Transactions on Power Systems.

A.J. Flueck and Hsiao-Dong Chiang (1998)

Solving the nonlinear power flow equations with an inexact Newton method using GMRES

IEEE Transactions on Power Systems.

F. de Leon and A. Semlyen (2002)

Iterative solvers in the Newton power flow problem: preconditioners, inexact solutions, and partial Jacobian updates.

IEE Proceedings - Generation, Transmission and Distribution.

S. Abhyankar (2013)

Evaluation of overlapping restricted additive Schwarz preconditioning for parallel solution of very large power flow problems.

ACM.

Thank you

https://github.com/exanauts/ExaPF.jl