PARALLEL CONJUGATE GRADIENTS ON SPARSE STIFFNESS MATRICES

RAYMOND VAN VENETIË AND JAN WESTERDIEP

1. Introduction

> "The conjugate gradient method (CG) is an algorithm for the numerical solutions of particular systems of linear equations, namely those whose matrix is symmetric and positive definite. The conjugate gradient method is often implemented as an iterative algorithm, applicable to sparse systems that are too large to be handled by a direct implementation or other direct methods such as the Cholesky decomposition. Large sparse systems often arise when numerically solving partial differential equations or optimization problems." [**?**]
>
> "The finite element method (FEM) is a numerical technique for finding approximate solutions to boundary value problems for partial differential equations. It uses subdivision of a whole problem domain into simpler parts, called finite elements, and variational methods from the calculus of variations to solve the problem by minimizing an associated error function. Analogous to the idea that connecting many tiny straight lines can approximate a larger circle, FEM encompasses methods for connecting many simple element equations over many small subdomains, named finite elements, to approximate a more complex equation over a larger domain." [**?**]

We will be concerned with solving

$$Ax = b$$

with known A and b.

In this report, we will look at a parallel implementation of the conjuate gradient method. We will solve linear systems coming from the finite element method. Systems like this are sparse in the sense that a lot of the elements will be zero. This makes it possible to skip parts of the matrix-vector products one encounters.

To test our implementation, it is useful to have a supply of matrices ready. The University of Florida has a huge collection of sparse matrices. [**?**] We chose to read these matrices in Matrix Market file format, allowing us to use various tools in existence. [**?**, **?**]

2. Finite Element Method

Given the boundary value problem,

$$\begin{cases} -\Delta u = 1 & \Omega \\ u = 0 & \partial\Omega \end{cases} \tag{1}$$

{eqn:problem}

and some partition into simplices, FEM will find the continuous solution, piecewise polynomial (wrt. this partition) with smallest H^1-norm error. ofzo

If we apply this to our 2D case, we have a polygonal domain Ω and a set of elements $\{\triangle_k\}_{k=1}^K$, $\triangle_k \subset \Omega$ with $\cup_{k=1}^K \triangle_k = \Omega$ and $\triangle_k \cap \triangle_j$ for $k \neq j$ is either empty, a common vertex or a common edge (hoe heet dit ook alweer? regularity condition ofzo?).

We create one reference element $\hat{\triangle}$ spanned by vertices $(0,0)$, $(1,0)$, $(0,1)$ on which we have a polynomial basis of some degree – say $\bar{p}$. The amount of basis functions in this basis must be $p = (\bar{p}+2)(\bar{p}+1)/2$. So we have a basis $\hat{\Phi} = \{\hat{\phi}_i\}_{i=1}^{p}$.

We are now able to create an affine function $T_k : \triangle_k \to \hat{\triangle}$ from some element in the partition to this reference element. Hence we automatically have a polynomial basis on $\triangle_k$, namely $\Phi^k := \{\phi_i^k\}_{i=1}^{p}$ with $\phi_i^k := \hat{\phi}_i \circ T_k$.

We can in fact create a global basis $\mathbf{\Phi}$ by gluing together the correct functions (TODO dit is lastig). Each basis function of this basis is a piecewise polynoimal subject to the partition, and is necessarily zero on $\partial\Omega$.

2.1. **Weak formulation.** If u solves (1), then

$$-\Delta u = 1 \implies -v\Delta u = v \forall v \in H_0^1(\Omega) \implies \int_\Omega -v\Delta u = \int_\Delta v$$

and using Green's first identity

$$\int_\Omega -v\Delta u = \int_\Omega \nabla v \cdot \nabla u - \oint_{\partial\Omega} v(\nabla u \cdot n)$$

where the last term must equal zero as $u = 0$ on $\partial\Omega$. We therefore end up at the weak formulation:

$$u \text{ solves (1)} \implies \int_\Omega \nabla v \cdot \nabla u = \int_\Omega v \forall v \in H_0^1(\Omega).$$

The idea is that we will find functions u_{FE} that exhibit this property and "hope" (TODO vinden dat dit klopt) that u_{FE} "almost" solves (1).

3. Linear polynomials on the triangle

If we take $\bar{p} = 1$, we are discussing linear polynomials on the triangle. One way to create a basis for this space is to use the *nodal basis*, where we create basis functions that are equal 1 on one vertex of the triangle and 0 on the others. The nodal basis on the reference element becomes

$$\hat{\phi}_1 = 1 - y - x, \quad \hat{\phi}_2 = y, \quad \hat{\phi}_3 = x.$$

If we look at this globally, we have a *nodal basis* for the whole partition, namely two-dimensional "hat" functions ϕ_i which are zero on every vertex but a single one – say v_i. The vertices on the boundary of the domain cannot have basis functions associated with them, as the resulting solution $u_{FE} = c^\top \mathbf{\Phi} = \sum c_i \phi_i$ must be zero on this boundary.

If we fill in u_{FE} in this weak formulation and set $v = \phi_j$, we get

$$\int_\Omega \nabla\phi_j \cdot \nabla\left(\sum c_i\phi_i\right) = \int_\Omega \phi_j \implies \sum c_i \int_\Omega \nabla\phi_j \cdot \nabla\phi_i = \int_\Omega \phi_j \forall j$$

or

$$Ac = b, \quad a_{ij} = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i, \quad b_i = \int_\Omega \phi_j.$$

In other words, finding this best solution $u_{FE} = c^\top \mathbf{\Phi}$ amounts to solving a linear system where the matrix A is real and symmetric (as inner products are commutative). This is where the Conjugate gradient method comes into play.

Hier moet nog aan toegevoegd worden:

- stiffness matrix is eigenlijk som van kleinere stiffness matrices
- deze grote stiffness matrix kan je maken/opslaan door op de goede plek shit in te voegen.

4. CG

In its most basic (sequential, non-preconditioned) form, CG can be written down as in Algorithm 1. [**?**, Alg. 4.8] We slightly adapted it from its original form to be able to use the different BLAS [**?**] routines.

Algorithm 1 Sequential CG

Require: $k_{max} \in \mathbb{N}$, $\epsilon \in \mathbb{R}$, $n \in \mathbb{N}$, A symmetric $n \times n$, $b \in \mathbb{R}^n$
Ensure: $x \in \mathbb{R}^n$ with $Ax \approx b$

```
int k := 0;
float r[n], u[n] = 0⃗, w[n];
float ρ, ρ_old, nbsq, α, β, γ;

r ← b;
ρ ← ⟨r, r⟩;
nbsq ← ρ;

while ρ > ε² · nbsq ∧ k < k_max do
    if k > 0 then
        β ← ρ/ρ_old;
        u ← βu;
    end if

    u ← r + u;
    w ← Au;
    γ ← ⟨u, w⟩;
    α ← ρ/γ;
    x ← x + αu;
    r ← r − αw;
    ρ_old ← ρ;
    ρ ← ⟨r, r⟩;

    k ← k + 1;
end while
```

4.1. **Sparse CG.** As FEM matrices are sparse in nature, we will adapt Algorithm 1 to a version that supports sparse matrices. For simplicity, we will assume the right-hand side to be dense. This allows us to effectively only change I/O and the function responsible for matrix-vector multiplication.

Our storage format uses the so-called coordinate scheme; we store tuples (i, j, a_{ij}) with i the row number, j the column number and a_{ij} the matrix value at this position. The Matrix Market file format adds some headers, e.g. to denounce symmetry so that one only has to store the lower triangular part. We denote by $nz(A)$ the amount of nonzero elements in this lower triangular part. If we store the list of tuples in three lists of length $nz(A)$, namely I, J and v, we can compute a sequential sparse matrix-vector multiplication using Algorithm 2 (from [**?**, Alg. 4.3]).

Algorithm 2 Sequential sparse matrix vector multiplication: find $y \leftarrow \alpha Ax + \beta y$ for symmetric A

Require: $\alpha \in \mathbb{R}$, $\beta \in \mathbb{R}$, $n \in \mathbb{N}$, amount of nonzeroes $nz(A)$, $I \in \mathbb{N}^{nz(A)}$, $J \in \mathbb{N}^{nz(Z)}$, $v \in \mathbb{R}^{nz(A)}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^n$
Ensure: $y \leftarrow \alpha Ax + \beta y$
1: **for** $i = 0$ until n **do**
2: $y[i] = \beta y[i]$;
3: **end for**
4: **for** $j = 0$ until $nz(A)$ **do**
5: $y[I[j]] = \alpha v[j] x[J[j]] + y[I[j]]$;
6: **if** $I[j] \neq J[j]$ **then** ▷ A is symmetric
7: $y[J[j]] = \alpha v[j] x[I[j]] + y[J[j]]$;
8: **end if**
9: **end for**

4.2. **Preconditioned CG.** The iterates x_k obtained from the CG algorithm satisfy the following inequality [**?**, Slide 23]:

$$\frac{\|x - x_k\|_A}{\|x - x_0\|_A} \leq 2 \left(\frac{\sqrt{\kappa_2(A)} - 1}{\sqrt{\kappa_2(A)} + 1} \right)^k \leq 2 \exp\left(-\frac{2k}{\sqrt{\kappa_2(A)}} \right)$$

where $\kappa_2(A)$ is the 2-condition number of A, which for symmetric positive definite matrices equates

$$\kappa_2(A) = \frac{\lambda_{max}}{\lambda_{min}}.$$

It is therefore of interest to create a condition number that is as low as possible.

A preconditioner P of a matrix A is a matrix such that $P^{-1}A$ has a smaller condition number than A. As the theoretical convergence rate is highly dependent on the condition number, we can improve this using such a preconditioner. Instead of solving $Ax = b$, we will solve $P^{-1}Ax = P^{-1}b$. Blablabla

5. Parallellizing CG

With our sequential algorithm in hand, we are now ready to parallellize the Conjugate gradient method. Given the symmetric positive definite matrix $A \in \mathbb{R}^{n \times n}$ and vector $b \in \mathbb{R}^n$, this requires us to find distributions for A, b and all other vectors present in the algorithm. If we want to minimize communication cost, we desire that the distributions of n, x, r, u, w are the same.[1] This allows us to perform all vector updates locally and makes for easy implementation of the inner product algorithm.

Every iteration of Algorithm 1 has 5 vector updates, two inner products and one matrix-vector multiplication. As $nz(A) \gg n$, the matrix-vector multiplication will likely [TODO profiling result] be the most time consuming. Thus, we want to optimize the distribution of A. To find the distributions of A and b (and with this, the distributions of all other vectors) we can use Mondriaan.[2]

[1] This was also pointed out in [**?**, p. 174].

[2] Mondriaan is a sequential program written in C that can be used to partition a rectangular sparse matrix, an input vector, and an output vector for parallel sparse matrix-vector multiplication. The program is based on a recursive bipartitioning algorithm that cuts the matrix horizontally and vertically, in a manner resembling some of the famous Mondriaan paintings. The algorithm is multilevel, hypergraph-based, and two-dimensional. It reduces the amount of communication and it spreads both computation and communication evenly over the processors. The program

With its option `-SquareMatrix_DistributeVectorsEqual=yes` we can force input and output vector to have the same distribution.

The resulting parallel algorithm is in appearance almost exactly as Algorithm 1, so we will not rewrite this. As the vector updates can be done locally without communication, BLAS routines were used (just as in the sequential algorithm). The big changes are made in computation of the inner product and the matrix-vector product. The `bspmv` algorithm found in [**?**, Alg. 4.5] and implemented in BSPedupack [**?**] was used without alterations, but the `bspip` algorithm [**?**, Alg. 1.1] was found unusable as this assumed a cyclical distribution. See Algorithm 3 for a parallel inner product algorithm that only assumes that both vectors have the same distribution. The beauty of this algorithm is that no processor has to know the distribution, as long as it has its own vector components stored as a smaller vector.

Algorithm 3 Parallel inner product $\langle v, y\rangle$ assuming v and y have equal distributions

Require: p total number of processors, $0 \leq s < p$ current processor number, nv_s the amount of vector elements locally stored, $v_s, y_s \in \mathbb{R}^{nv_s}$ local vectors
Ensure: $\alpha = \langle v, y\rangle$
  float $\alpha_s = 0$;
  float $\alpha = 0$;

  for $i = 0$ to nv_s **do** ▷ Compute local inner product
    $\alpha \leftarrow \alpha + v_s[i] \cdot y_s[i]$;
  end for

  for $q = 0$ to p **do** ▷ Put local inner product
    Put α_s to $P(q)$;
  end for

  for $q = 0$ to p **do** ▷ Find global inner product
    $\alpha \leftarrow \alpha + \alpha_q$;
  end for

5.1. **BSP cost.** We first calculate the BSP cost of the parallel algorithm per iteration, using Algorithm 1 as reference. Let nv_s be the amount of vector elements locally stored on processor s. Line 12 costs nv_s operations and lines 15, 19 and 20 cost $2nv_s$ each. This makes vector updates contribute $7nv_s$ to the total amount of operations.

Looking at Algorithm 3, each inner product yields $2nv_s - 1$ operations for Superstep 1, pg operations for Superstep 2 and $p-1$ operations for Superstep 3, with 2 synchronizations in between for a total of $2(2nv_s - 1 + p - 1 + pg + 2l)$ inner product operations per iteration.

TODO: uitzoeken of die load imbalance ook voor de vector distributies geldt

TODO MV

6. Testing our parallel CG

To test our implementation, we want to have access to a lot of similar matrices. One way to do this is described in the book: create a random sparse matrix B with values in $[-1, 1]$, then take

can partition hypergraphs with integer vertex weights and uniform hyperedge costs, but it is primarily intended as a matrix partitioner. [**?**]

$A \leftarrow B + B^\top + \mu I$ with μ such that A is strictly diagonally dominant. As the idea of this is merely to generate a matrix A that is symmetric positive definite, we opted for a slightly easier approach.

We need to compute a *symmetric* matrix, so we only have to look at the lower triangular part of this matrix. First we create a random sparse strict lower triangular[3] matrix B. We do this by specifying some density δ and placing a random number in $[-1, 1]$ on position (i, j) which chance δ. The matrix must be positive definite, which is achieved if

$$|A_{ii}| > \sum_{j=1, j\neq i}^{n} |A_{ij}| = \sum_{j=1}^{i-1} |B_{ij}| + |B_{ji}|.$$

While placing the non-zeros in the lower triangular part of B, we therefore iteratively store the sum in the right hand side of this equation for the corresponding diagonal element. Let μ be the maximum of these sums, we now know that $B + B^t + uI$ is a symmetric positive definite matrix. Finally, to add some more randomness to the diagonal, we replace the diagonal element u by a random number in $[\mu, \mu + 2]$ on with chance δ.

This parameter δ gives is an indication for how the 'sparness' of the matrix. With this method we can generate a variety of matrices for different combinations of n and δ, which gives insight in the performance of our parallel implementation.

6.1. **Profiling.**

6.2. **Scaling tests.**

With these matrices in hand, we can perform scaling tests by varying p, n and δ.

7. Creating FEM matrices

8. hoe heette het ook alweer als je na een kleine verandering in je matrix opnieuw ging optimizen

9. Future work

9.1. **Adaptive FEM.**

[3]Only elements under the main diagonal are set