6-2019

Utilizing Parallelism in the Conjugate Gradient Algorithm

Adam James Craig
Portland State University

Utilizing Parallelism in the Conjugate Gradient Algorithm

Adam Craig

May 24th, 2019

0.1 Introduction

There are a myriad of applications that motivate the discussion of numerical solutions of linear systems, especially in the field of machine learning, where the matrices at hand may also be symmetric positive definite. For example, Fischer discriminate analysis, which is commonly used in computer vision algorithms, requires one to solve many such systems repeatedly. There is certainly no shortage of situations in which one might need to solve these systems numerically (a typical case for large-scale problems).

This paper will examine the conjugate gradient method specifically, which is an algorithm that approximates the inverse action of a symmetric positive definite matrix, and outline an approach that is designed to improve performance on modern machines. The original conjugate gradient method was published in 1952 by Eduard Stiefel and Magnus Hestenes, and is totally iterative [1]. Further, the modern philosophy behind the method is that the system is solved without explicitly storing the matrix in question in memory. We will outline a preconditioner of the conjugate gradient method that (a) converges in fewer iterations and (b) lends itself to parallel computation, which creates a real-time speed increase on modern machines with many cores.

0.2 Development of the conjugate gradient method

We examine the system $A\mathbf{x} = \mathbf{b}$, where $\mathbf{b} \in \mathbb{R}^n$ is given and $A \in \mathbb{R}^{n\times n}$ is assumed to be symmetric and positive-definite. Because A is symmetric positive-definite, A^{-1} exists and thus there is an exact solution $\mathbf{x}_*$ to the system. If the set $\{\mathbf{v_1}, \mathbf{v_2}, ..., \mathbf{v_n}\}$ is a collection of A-orthogonal vectors, then it is a basis for the domain of A. Therefore, it must be the case that there exist scalars c_i such that,

$$A\mathbf{x}_* = \sum_{i=1}^{n} c_i A\mathbf{v_i} \Rightarrow \mathbf{v_k}^T A\mathbf{x}_* = \sum_{i=1}^{n} c_i \mathbf{v}_k^T A\mathbf{v}_i \Rightarrow \mathbf{v}_k^T A\mathbf{x}_* = c_k \mathbf{v}_k^T A\mathbf{v}_k \Rightarrow c_k = \frac{\mathbf{v}_k^T \mathbf{b}}{\mathbf{v}_k^T A\mathbf{v}_k}$$

This holds primarily because $\mathbf{v}_i$ is A-orthogonal to $\mathbf{v}_j$ when $i \neq j$, meaning $\mathbf{v}_i^T A\mathbf{v}_j = 0$. However, this direct solution requires one to procure such a set of n orthogonal vectors and then perform this computation to attain each c_i. So we develop a procedure such that we may not need n vectors to get an acceptable approximation of $\mathbf{x}_*$. Clever choices of each $\mathbf{v}_i$, which are canonically referred to as *search directions*, will allow us to perform the algorithm without (theoretically) needing all n vectors.

This begins by exploring a seemingly unrelated problem: consider optimizing the quadratic function $\phi : \mathbb{R}^n \to \mathbb{R}$ defined by $\phi(x) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{x}^T\mathbf{b}$. It can be shown readily that ϕ is a convex function, and so it has a unique minimizer. In fact, setting $\nabla\phi(x) = A\mathbf{x} - \mathbf{b} = 0$, we see that this minimizer is $\mathbf{x}_*$. Therefore, minimizing ϕ and solving $A\mathbf{x} = \mathbf{b}$ are equivalent problems. So, how does this help us produce clever choices of $\mathbf{v}_i$? We initially take $\mathbf{v}_0 = -\nabla\phi(\mathbf{x}_0)$ because this is the direction that ϕ decreases the fastest. This idea describes the beginning of the "steepest descent" algorithm.

The steepest descent algorithm is straightforward. Set $\mathbf{x} = \mathbf{0}$ or a guess approximating $\mathbf{x}_*$. Define the *residual* at $\mathbf{x}$ by $\mathbf{r} = \mathbf{b} - A\mathbf{x} = -\nabla\phi(x)$. In this way, we augment $\mathbf{x}$ by setting $\mathbf{x} := \mathbf{x} + \alpha\mathbf{r}$, for some real α, which causes us to 'step' in the direction of steepest descent at each iteration. To expand on this idea, we can do even better by minimizing $\phi(\mathbf{x} + \alpha\mathbf{r})$ with regards to α. Setting $\alpha = \mathbf{r}^T\mathbf{r}/\mathbf{r}^T A\mathbf{r}$ at each iteration minimizes $\phi(\mathbf{x} + \alpha\mathbf{r})$ and in fact $\phi(\mathbf{x} + \alpha\mathbf{r}) < \phi(\mathbf{x})$. Because $\mathbf{x}$ and $\mathbf{r}$ change with each iteration, it is helpful to introduce the following relation: $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{r}_k$ and $\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k$, where $\alpha_k = \mathbf{r}_k^T\mathbf{r}_k/\mathbf{r}_k^T A\mathbf{r}_k$. In this case, each search direction is $\alpha_k\mathbf{r}_k$. But the steepest descent has the potential to be "prohibitively slow" because situations can arise in which each search direction is 'too similar' to the last [2].

So we wish to improve upon our set of search directons $\{\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n\}$ such that they are not all equal to the residual at each iteration. In a similar manner, $\phi(\mathbf{x}_{k-1} + \alpha_k \mathbf{v}_k)$ is minimized by setting $\alpha_k = \mathbf{v}_k^T \mathbf{r}_{k-1} / \mathbf{v}_k^T A \mathbf{v}_k$, so long as $\mathbf{v}_k$ is not orthogonal to $\mathbf{r}_{k-1}$, and it can be shown that such a $\mathbf{v}_k \in \text{span}\{\mathbf{v}_{k-1}, \mathbf{r}_{k-1}\}$ that satisfies this requirement exists at each iteration $k \geq 2$ [2]. Therefore, this provides an easy way to compute the next search direction: $\mathbf{v}_k = \mathbf{r}_{k-1} + \beta_k \mathbf{v}_{k-1}$, where $\beta_k = -\mathbf{v}_{k-1}^T A \mathbf{r}_{k-1} / \mathbf{v}_{k-1}^T A \mathbf{v}_{k-1}$.

Of course, it is not always necessary to achieve the exact solution $\mathbf{x}_*$, and instead we may wish to be 'close enough.' For example, given some ϵ, say, $\epsilon = 10^{-6}$, we can consider the value $\epsilon \mathbf{b}^T \mathbf{b}$ and consider the the algorithm complete when $\mathbf{r}_k^T \mathbf{r}_k < \epsilon \mathbf{b}^T \mathbf{b}$. Furthermore, we may want to impose the limitation that the process must terminate within a certain amount of iterations. So, if *iterMax* is the maximum number of iterations and *iter* the current iteration, this gives us the following algorithm [2]:

```
Initialize $\mathbf{x} = 0$, $\mathbf{r} = \mathbf{b}$, and $iter = 0$.
Set $\mathbf{v} = \mathbf{r}$, $\delta_0 = \mathbf{b}^T \mathbf{b}$, and $\delta = \delta_0$.
while($\delta_0 \geq \epsilon^2 \mathbf{b}^T \mathbf{b}$ and $iter \leq iterMax$)
    (1) Set $\delta_{Old} = \delta$.
    (2 Compute $\mathbf{g} = A\mathbf{v}$.
    (3) Compute $\alpha = \delta / \mathbf{v}^T \mathbf{g}$.
    (4) Compute $\mathbf{x} := \mathbf{x} + \alpha \mathbf{v}$.
    (5) Compute $\mathbf{r} := \mathbf{r} - \alpha \mathbf{g}$.
    (6) Compute $\delta = \mathbf{r}^T \mathbf{r}$.
    (7) Compute $\mathbf{v} := \mathbf{r} + \beta \mathbf{v}$, where $\beta = \delta / \delta_{Old}$.
    (8) $iter := iter + 1$.
```

It should be noted that this algorithm requires $10n$ flops each iteration, and so the $O(n)$ convergence time can be cumbersome, when n is very large. Further, there exists iterative machine error, and even without error convergence is not even guaranteed [2]. The benefit, however, is that only one 'matrix-times-vector' is performed each iteration. Indeed, only four vectors are required to be stored, too, so this method's primary benefit is the low memory requirement. Combining this with a kernel function to produce a SPD A, we would not need to store A, either. But the main drawback is in the $O(n)$ time, which can surely be improved upon.

It is appropriate to mention now a few conditions that would be indicative of a 'speed up' in this scenario. For one, if A's condition number is sufficiently close to 1, the algorithm is faster. Also, if A has many repeated eigenvalues, then we would see improved speed [2].

0.3 Preconditioning

The idea of a preconditioned conjugate gradient method is relatively straightforward. We begin by working with the new system $\hat{A}\hat{\mathbf{x}} = \hat{\mathbf{b}}$. Here, $\hat{A} = B^{-1}AB^{-1}$, $\hat{\mathbf{x}} = B\mathbf{x}$, and $\hat{\mathbf{b}} = B^{-1}\mathbf{b}$, where B is symmetric positive definite. From here, we just apply the conjugate gradient method to this new system [2]. Clearly, in light of the fact that we can improve the speed of this algorithm by ensuring $\hat{A}$ has repeated eigenvalues or a sufficiently low condition number, we have an idea of what types of B we are looking for.

It follows naturally that $\hat{\mathbf{r}}_{k+1} = \hat{\mathbf{b}} - \hat{A}\hat{\mathbf{x}}_k$, $\hat{\mathbf{x}}_k = \hat{\mathbf{r}}_k + \alpha \hat{\mathbf{v}}_k$, and $\hat{\mathbf{v}}_k = \hat{\mathbf{r}}_k + \beta \hat{\mathbf{v}}_{k-1}$. But notice that, once this system has been solved, we must procure B^{-1} to compute $\mathbf{x} = B^{-1}\hat{\mathbf{x}}$. If $B \in \mathbb{R}^{n\times n}$, then this is exactly the type of direct computation that we originally wished to avoid. To circumvent this, we define $\hat{\mathbf{v}}_k = B\mathbf{v}_k$, $\hat{\mathbf{x}}_k = B\mathbf{x}_k$, and $\hat{\mathbf{r}}_k = B^{-1}\mathbf{r}_k$. Defining terms in this way, our modified algorithm uses the benefits of preconditioning without needing to find B^{-1} and instead uses B^2. We call $M = B^2$ the preconditioner, and we have the following preconditioned algorithm [2]:

```
Initialize $\mathbf{x} = 0$, $\mathbf{r} = \mathbf{b}$, and $iter = 0$.
    (0) Solve $M\mathbf{z} = \mathbf{r}$ and let $\delta = \delta_0 = \mathbf{z}^T\mathbf{r}$.
while ($\delta \geq \epsilon^2\delta_0$ and $iter \leq iterMax$)
    (1) Set $\delta_{Old} = \delta$.
    (2) Compute $\alpha = \frac{\delta}{\mathbf{v}^T A\mathbf{v}}$ and then $\mathbf{x} := \mathbf{x} + \alpha\mathbf{v}$.
    (3) $\mathbf{r} := \mathbf{b} - A\mathbf{x}$.
    (4) Solve $M\mathbf{z} = \mathbf{r}$ and let $\delta = \mathbf{z}^T\mathbf{r}$.
    (5) Compute $\beta = \frac{\delta}{\delta_{Old}}$ and then $\mathbf{v} := \mathbf{z} + \beta\mathbf{v}$
    (6) $iter := iter + 1$.
```

It is worth noting that because we have slightly altered the definitions of some of these terms, $\alpha_{k-1} = \mathbf{r}_{k-1}^T\mathbf{z}_{k-1}/\mathbf{v}_k^T A\mathbf{v}_k$, and $\beta_k = \mathbf{r}_{k-1}^T\mathbf{z}_{k-1}/\mathbf{r}_{k-2}^T\mathbf{z}_{k-2}$. In this version of the method, our choice of M is essential because it will ultimately determine the speed of the method. One must take care to make sure that the system $M\mathbf{z} = \mathbf{r}$ is able to be solved rapidly because this system will be solved at each iteration. Clever choices of M can significantly increase the speed of the algorithm [2].

0.4 An exponential kernel and additive Schwarz preconditioner

We initially set out to test the original conjugate gradient method, and the first part of this process involves producing some A. As we stated previously, the main benefit of this algorithm is that it requires little in the way of memory, to continue with this trend, we choose to produce our matrix from a kernel, so that entries can be computed on the fly and do not need to be stored in memory.

This kernel requires that, if $A \in \mathbb{R}^{n\times n}$, then it must be the case that $n = k^2$, for some $k \in \mathbb{N}$. This is because we map each element of $[n] = \{1, 2, ..., n\}$ to a distinct point on a tensor product mesh on $[0, 1]^2$. Specifically, we have a function $\gamma : \{1, 2, ..., n\} \to [0, 1]^2$ defined to be $\gamma(x) = (\gamma(x)_1, \gamma(x)_2) := (q/(k-1), r/(k-1))$, where $x = qk + r$ by the division algorithm. This mapping from an index value exists to provide an input value to another necessary function: the euclidian distance function in two dimensions. In other words, we compute $\sqrt{(\gamma(i)_1 - \gamma(j)_1)^2 + (\gamma(i)_2 - \gamma(j)_2)^2}$. This gives a kernel function:

$$A_{i,j} = K(\gamma(i), \gamma(j)) = e^{-\sqrt{(\gamma(i)_1-\gamma(j)_1)^2+(\gamma(i)_2-\gamma(j)_2)^2}}.$$

There are a few things to discuss about this kernel. First note that it is indeed an SPD matrix. It is also worth mentioning that, because the value decreases as the distance between $\gamma(x)$ and $\gamma(y)$ increases, it can be intuitively interpreted as a measurement of dissimilarity. This was chosen because of its intuitive and practical simplicity, as well as the fact that it is a commonly used kernel in many other learning algorithms.

The subject of interest in this paper is the idea of a preconditioned system that can be computed in parallel, which we will develop now. First, consider the standalone additive Schwarz method, which is an analytic generalization of the Block-Jacobi algorithm [3]. This algorithm considers a system $A\mathbf{u} = \mathbf{f}$ with $A \in \mathbb{R}^{n\times n}$ and the residual $\mathbf{r}^k$ defined as usual. Further, partition $[n]$ into the blocks $\mathcal{N}_1 = \{1, 2, ..., l\}$, $\mathcal{N}_2 = \{l+1, l+2, ..., n\}$, and let $\mathcal{N} = \mathcal{N}_1 \cup \mathcal{N}_2$. Let the matrix R_i be the restriction operator from $\mathcal{N}$ to $\mathcal{N}_i$ and R_i^T the extension operator from $\mathcal{N}_i$ to $\mathcal{N}$. Finally, define $\mathbf{u}^i := \mathbf{u}|_{\mathcal{N}_i}$, $\mathbf{f}^i := \mathbf{f}|_{\mathcal{N}_i}$, and $A_{ij} := A|_{\mathcal{N}_i x \mathcal{N}_j}$. This partitions the system in the following block form

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{u}^1 \\ \mathbf{u}^2 \end{pmatrix} \begin{pmatrix} \mathbf{f}^1 \\ \mathbf{f}^2 \end{pmatrix}$$

Beginning with some approximation of the solution $\mathbf{u}_k$ define the following iteration:

$$\mathbf{u}^{k+1} = \mathbf{u}^k + (R_1^T(R_1AR_1^T)^{-1}R_1 + R_2^T(R_2AR_2^T)^{-1}R_2)\mathbf{r}^k$$

The preconditioner in question (actually its inverse), however, will be defined as

$$M^{-1} = \sum_{i=1}^{p} R_i^T A_{ii}^{-1} R_i, \text{ where } A_{ii} = R_i A R_i^T.$$

Above, each $\mathcal{N}_i$ can be any random subset of $[n]$ of chosen size m (such that $mp \geq n$). Further, when the system $M\mathbf{z} = \mathbf{r}$ is solved at each iteration, we will not be interested in $\mathbf{z}$ explicitly but instead $\tilde{\mathbf{r}} = \sum_{i=1}^{p} R_i^T(R_iAR_i^T)^{-1}R_i\mathbf{r}$, which can be thought of as a rough approximation of $\mathbf{z}$. This preconditioner is used because it can be computed in parallel, thus satisfying the requirement that $M\mathbf{z} = \mathbf{r}$ is quickly solvable, and it serves as a crude approximation of A^{-1}, thereby aiding in finding the next search direction. In this experiment, we set the size of the sub problems $m = \sqrt{n}$. This lets us define the following algorithm:

```
Initialize $\mathbf{x} = 0$, $\mathbf{r} = \mathbf{b}$, $\tilde{\mathbf{r}} = 0$, and $iter = 0$.
Create $p$ (possibly overlapping) subsets of $[n]$ of size $m$ $\{\mathcal{N}_1, ..., \mathcal{N}_p\}$.
Compute $\tilde{\mathbf{r}} = \sum_{i=1}^{p} R_i^T(R_iAR_i^T)^{-1}R_i\mathbf{r}$ in parallel.
Set $\mathbf{v} = \tilde{\mathbf{r}}$.
Set $\delta_0 = \tilde{\mathbf{r}}^T\mathbf{r}$ and $\delta = \delta_0$.
while($iter \leq iterMax$ and $\delta \geq \epsilon^2\delta_0$)
    (1) Compute $\mathbf{g} = A\mathbf{v}$
    (2) Compute $\tau = \mathbf{v}^T\mathbf{g}$
    (3) Compute $\alpha = \delta/\tau$
    (4) Update $\mathbf{x} := \mathbf{x} + \alpha\mathbf{v}$
    (5) Update $\mathbf{r} := \mathbf{r} - \alpha\mathbf{g}$
    (6) Construct $m$ blocks $\{\mathcal{N}_1, ..., \mathcal{N}_p\}$
    (7) Solve each $(R_iAR_i^T)^{-1}$ in parallel and compute $\tilde{\mathbf{r}} = \sum_{i=1}^{p} R_i^T(R_iAR_i^T)^{-1}R_i\mathbf{r}$
    (8) Set $\delta_{\text{Old}} = \delta$
    (9) Compute $\delta = \tilde{\mathbf{r}}^T\mathbf{r}$
    (10) Increase the iteration counter $iter := iter + 1$
    (11) Compute the next search direction: $\mathbf{v} := \tilde{\mathbf{r}} + \beta\mathbf{v}$, where $\beta = \delta/\delta_{\text{Old}}$.
```

0.5 The empirical data

The experiment was done on Portland State University Linux server and using mostly my own C++11 library, made specifically for this project [4], but for all computations done in parallel, the OpenMP library was used [5]. Stephan Gelever of Portland State aided in debugging and trouble-shooting during the development of the software used to run the experiment.

The process was straightforward: generate a random vector $\mathbf{x}_* \in \mathbb{R}^n$, and create a system $A\mathbf{x}_* = \mathbf{b}$ using the actions of A generated by the kernel described in the previous section. From here, apply the conjugate gradient and preconditioned conjugate gradient algorithms to the system, beginning by setting $\mathbf{x} = 0$, and collecting the resulting iteration and time data. We did this for nine different values of n.

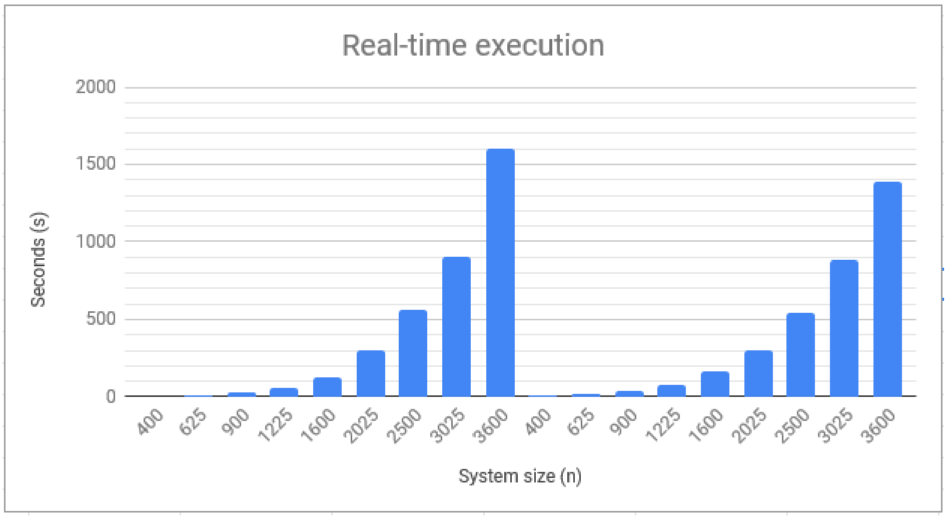

Figure 1: Execution time data for the CG method (left) and preconditioned CG method (right).

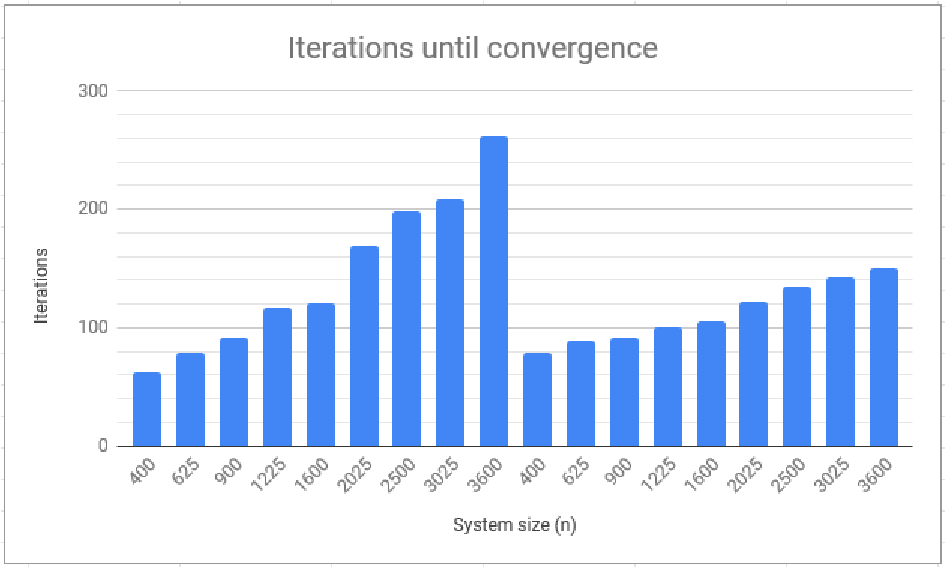

Figure 2: Iteration data for the CG method (left) and preconditioned CG method (right).

0.6 Conclusion

The preconditioned system ends up converging in fewer iterations as n gets large, signifying that search directions found by the parallel Block-Jacobi analogue are more effective than the search directions produced by the standalone conjugate gradient method. This difference becomes more pronounced as n increases, resulting in a real-time speed increase, even though each iteration of the preconditioned method may take longer, on average. Moreover, one would see a greater speed up with access to many cores (or nodes, in the case of a computer cluster), so that computation can be done in parallel.

There are a number of caveats one must be aware of before employing this preconditioner to solve large systems, however. First, notice how the requirement that A was not stored explicitly in memory was relaxed to the requirement that A was not stored in a single node. This is because we require p parallel $m \times m$ systems, and this assumption may be too relaxed for very large systems, but there a number of possible avenues to remedy this. One could restrict the domain decomposition – instead of allowing overlap between blocks – and solve fewer blocks per iteration. Also, the implementation used in this experiment relied on direct Gaussian elimination to solve each block, which will not scale with the linear time of the conjugate gradient. One possible fix for this is to use the symmetric Gauss-Seidel method instead, seeing as each block is also guaranteed to also be symmetric positive definite.

To conclude, we mention that possible applications of interest include linear discriminant analysis and classification problems, including problems in computer vision, marketing, and computational medicine.

Bibliography

[1] Hestenes, M., Stiefel, E. (1952). Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards, 49*(6), 409. doi:10.6028/jres.049.044

[2] Golub, G. H., F., V. L. C. (1996). *Matrix computations* (3rd ed.). Baltimore: The Johns Hopkins University Press.

[3] Victorita Dolean, Pierre Jolivet, Frédéric Nataf. An Introduction to Domain Decomposition Methods: algorithms, theory and parallel implementation. Master. France. 2015. cel-01100932v5

[4] Craig, A. A preconditioned conjugate gradient linear algebra C++ library, (2019), GitHub repository, https://github.com/acraigmath/CGtest

[5] OpenMP Architecture Review Board, "OpenMP Application Program Interface, Version 5.0", November 2018.