Fine-Grain Parallel RRB-Solver for 5-/9-Point Stencil Problems Suitable for GPU-Type Processors

de Jong, Martijn; van der Ploeg, Auke; Ditzel, Auke; Vuik, Kees

Publication date
2017

Document Version
Final published version

Published in
Electronic Transactions on Numerical Analysis

Citation (APA)
de Jong, M., van der Ploeg, A., Ditzel, A., & Vuik, K. (2017). Fine-Grain Parallel RRB-Solver for 5-/9-Point Stencil Problems Suitable for GPU-Type Processors. *Electronic Transactions on Numerical Analysis*, *46*, 375-393.

Electronic Transactions on Numerical Analysis.
Volume 46, pp. 375–393, 2017.

ISSN 1068–9613.

ETNA
Kent State University and
Johann Radon Institute (RICAM)

FINE-GRAIN PARALLEL RRB-SOLVER FOR 5-/9-POINT STENCIL PROBLEMS SUITABLE FOR GPU-TYPE PROCESSORS*

MARTIJN DE JONG†, AUKE VAN DER PLOEG‡, AUKE DITZEL‡, AND KEES VUIK†

Abstract. Preconditioners based on incomplete factorization are very popular for a fast convergence of the PCG-algorithm. However, these preconditioners are hard to parallelize since most operations are inherently sequential. In this paper we present the RRB-solver, which is a PCG-type solver using an incomplete Cholesky factorization based on the Repeated Red-Black (RRB) method. The RRB-solver scales nearly as well as Multigrid, and in this paper we show that this method can be parallelized very efficiently on modern computing architectures including GPUs. For an efficient parallel implementation a clever storage scheme turns out to be the key. The storage scheme is called $r_1/r_2/b_1/b_2$ and it ensures that memory transfers are coalesced throughout the algorithm, yielding near-optimal performance of the RRB-solver. The $r_1/r_2/b_1/b_2$-storage scheme in combination with a CUDA implementation on the GPU gives speedup factors of more than 30 compared to a sequential implementation on one CPU core for 5-/9-point stencils problems. A comparison with algebraic Multigrid further shows that the RRB-solver can be implemented very efficiently on the GPU.

Key words. repeated red-black, conjugate gradient, incomplete Cholesky, parallelization, CUDA, 2D Poisson

AMS subject classifications. 35J05, 65F08, 65F10, 65F50, 65Y05, 68W10

1. Introduction.

Poisson-type problems given by 5- or 9-point stencils arise in many applications and in many research fields. For example, in our real-time ship simulator a Poisson-type problem given by a 5-point stencil needs to be solved multiple times per second in order to compute realistic, interactive waves and realistic ship movements [8]. The size of this Poisson-type problem is directly related to the size of the computational domain of interest, e.g., a harbour, a river, or a sea. To fulfil the requirements of a real-time simulation, a very fast solver is required; simply stated: the faster the solver, the larger the domain that can be computed in real-time.

The proposed solution method in this paper can be applied to solve Poisson-type problems given by 9-point stencils as well. Therefore, throughout this paper we consider a linear system of the form

$$A\mathbf{x} = \mathbf{b}, \tag{1.1}$$

where $A \in \mathbb{R}^{n\times n}$ is a large symmetric positive definite (SPD) matrix given by the stencils,

$$\begin{bmatrix} \cdot & N & \cdot \\ W & C & E \\ \cdot & S & \cdot \end{bmatrix}^{(i,j)} \quad \text{or} \quad \begin{bmatrix} NW & N & NE \\ W & C & E \\ SE & S & SW \end{bmatrix}^{(i,j)}$$

for each node (i, j) in a rectangular $n \times n$ grid. Here N refers to 'North', NE to 'North-East', and so on. There are several methods that can be used to solve (1.1):

1. A direct method, for example a construction of a complete Cholesky decomposition. All fill-in occurring during the Gaussian elimination is within a relatively small band around the main diagonal. An advantage is that with this method relatively high speed-ups can be obtained on parallel architectures with block versions; the

*Received November 11, 2016. Accepted July, 19, 2017. Published online on October 3, 2017. Recommended by Y. Saad.
†Delft University of Technology, Mekelweg 4, 2628 CD Delft, Netherlands
({m.a.dejong-2, c.vuik}@tudelft.nl).
‡Maritime Research Institute Netherlands, Haagsteeg 2, 6708 PM Wageningen, Netherlands
({a.v.d.ploeg, a.ditzel}@marin.nl).

ETNA
Kent State University and
Johann Radon Institute (RICAM)

drawback, however, is that the number of operations increases rather fast with grid refinement. For example, when a rectangular $n \times n$ grid is used, the size of the matrix is n^2, and the width of the above-mentioned band is n. In that case, the number of required operations is $\mathcal{O}(n^4)$. Special reordering methods can be used to speed up the computations; however, these techniques have poor parallelization opportunities. Examples of software: Mumps [1], Umfpack [7], Pardiso [15].

2. A spectral method (e.g., Fishpack [16]) is very fast and parallelizable. However, it is limited to matrices having constant coefficients.
3. Iterative methods. They may be classified into stationary and gradient methods. The stationary methods of Jacobi and Gauss-Seidel are the earliest iterative methods and they are based on a splitting of the matrix. For many cases they can be accelerated by using relaxation techniques, leading, for example, to the well-known successive over-relaxation (SOR) method. In case of an SPD matrix, one can also use the Conjugate Gradient (CG) method. In exact arithmetic, this method finds an approximate solution in such a way that the A-norm of the error is minimized over the Krylov subspace

$$\mathcal{K}_k(A, \mathbf{r}_0) = \operatorname{span}\{\mathbf{r}_0, A\mathbf{r}_0, A^2\mathbf{r}_0, \ldots, A^{k-1}\mathbf{r}_0\},$$

where $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$ is the initial residual. Hence the residual at step k can be written as $P_k(\mathbf{r}_0)$, in which P_k is the 'optimal' polynomial of degree less than $k-1$ such that $P_k(0) = I$.
It is possible to use Chebychev polynomials to give bounds for the convergence rate of the CG method based on the fact that they cannot produce better reductions of the error than the optimal polynomial. A well-known upper bound for the error is [17]:

$$\|\mathbf{x}_k - \mathbf{x}\|_A \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k \|\mathbf{x}_0 - \mathbf{x}\|_A, \tag{1.2}$$

where $\| \cdot \|_A$ denotes the A-norm, and $\kappa(A) := \lambda_{\max}/\lambda_{\min}$ the spectral condition number, i.e., the ratio of the largest to the smallest eigenvalue of A. The convergence behaviour of CG thus strongly depends on the spectral condition number. The convergence rate can often strongly be improved by applying CG to the preconditioned system

$$M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}. \tag{1.3}$$

The SPD matrix M is called the preconditioner. There is a wide choice of preconditioners, see, for example, [5] and [14].
The matrix M should be a proper approximation of A, such that the spectral condition number of $M^{-1}A$ is much smaller than that of A, and solving the systems $M\mathbf{z} = \mathbf{r}$ should be cheap. One possibility is to try to make a SParse Approximation of the Inverse (SPAI-methods). However, even when the coefficient matrix itself is very sparse, its inverse is often not, and therefore it is questionable if a proper SPAI-preconditioner can be constructed. For symmetric matrices many preconditioners are based on an Incomplete Cholesky decomposition. In this method, a lower-triangular matrix L is constructed such that $A \approx LL^T$, and where the factor L is sparse. In that case: $M := LL^T$ and systems $M\mathbf{z} = \mathbf{r}$ can be readily solved by two triangular sweeps.

A good solution method should have a limited number of operations per node, and this number should not increase too fast with mesh refinement. In addition, it should be possible to

ETNA
Kent State University and
Johann Radon Institute (RICAM)

exploit modern computer architectures in order to obtain a high flop rate. Suppose that one has to solve a Poisson equation on a rectangular mesh with constant mesh size $h = 1/(n+1)$. In that case, one can show that SOR takes $\mathcal{O}(n^3)$ operations, whereas preconditioned CG, in which the diagonal of the preconditioner is modified according to Gustafsson [10], the computational complexity is $\mathcal{O}(n^{5/2})$.

The so-called Repeated Red-Black (RRB) method, described in [3] combines a reordering of the rows and columns of the matrix with an Incomplete Cholesky decomposition. This method has two important advantages:

1. For the test case with uniform mesh mentioned above, an upper bound of the computational complexity is given by (cf. [12], eq. (1.8)):

$$\kappa(M^{-1}A) \leq \frac{\sqrt{5}(\sqrt{5}-1)^{\ell-1}}{1+(-1)^{\ell}\left(\frac{3-\sqrt{5}}{2}\right)^{\ell-1}} \approx 1.8 \cdot 1.23^{\ell},$$

   where ℓ is the number of consecutive red-black levels. Hence the number of iterations is expected to increase only mildly with mesh refinement.
2. The reordering strategy based on repeated red-black guarantees that large parts of the preconditioner can be built in parallel, and large parts of the triangular sweeps can be performed in parallel as well.

In this paper we provide an efficient implementation, that is, an implementation of the RRB-method that can fully exploit the parallelism of modern architectures, such as GPUs. Key to this is a new reordering strategy such that all global memory operations can be performed in a coalesced manner.

The remaining sections are organized as follows. In Section 2 the RRB-solver and its aspects are presented. In Section 3 it is explained which techniques can be used to obtain an efficient parallel implementation of the RRB-solver on both multi-core CPU and GPU systems. In Section 4 the experimental setup and the test problem are discussed. In Section 5 several performance results are presented, as well as a detailed throughput analysis and a speed comparison between the RRB-solver and algebraic Multigrid. In Section 6 the conclusions can be found.

2. The RRB-solver. The RRB-solver is a PCG-type solver where the RRB-method serves as the preconditioner M. RRB stands for 'Repeated Red-Black' (or 'Recursive Red-Black') and refers to how the nodes in a 2D grid are colored and numbered. The RRB-method found its origin in the late eighties where multigrid V-cycles with intermediate skew meshes were investigated. Since then the method has been further investigated by Axelsson and Eijkhout [2] and Brand and Heinemann [3, 4]. They showed that the RRB-method can be used as a preconditioner in Krylov methods leading to a method with nearly optimal scaling. In [6, 12] additional information can be found as well as derivations for upper bounds for the condition number κ.

In order to show how the RRB-method can be parallelized, the RRB-method is described in this section. First we discuss RRB-numbering, then we present the RRB-factorization algorithm and, finally, we explain how the RRB-method can be used as a preconditioner.

2.1. The RRB numbering procedure. Let

$$G = \{(i,j) \mid 1 \leq i \leq N_x, 1 \leq j \leq N_y\}$$

ETNA
Kent State University and
Johann Radon Institute (RICAM)

be the set of all nodes in an $N_x \times N_y$ grid. If $(1,1)$ is chosen to be a black node, then a standard red-black ordering is given by:

$$\begin{aligned} R^{[1]} &= \big\{(i,j) \in G \mid \mathrm{mod}(i+j,2) = 1\big\}, \\ B^{[1]} &= G \setminus R^{[1]}, \end{aligned}$$

where $R^{[1]}$ denotes the set of first level red nodes and $B^{[1]}$ denotes the set of first level black nodes. Next, a standard red-black ordering is reapplied to the $B^{[1]}$-nodes as follows:

$$\begin{aligned} R^{[2]} &= \big\{(i,j) \in B^{[1]} \mid \mathrm{mod}(j,2) = 0\big\}, \\ B^{[2]} &= B^{[1]} \setminus R^{[2]} \\ &= G \setminus (R^{[1]} \cup R^{[2]}). \end{aligned}$$

The second level black nodes, i.e., the $B^{[2]}$-nodes, are thus the nodes in G that neither belong to the sets $R^{[1]}$ nor $R^{[2]}$. Generally,

$$R^{[k]} = \begin{cases} \left\{(i,j) \in G \setminus \left(\cup_{p=1}^{k-1} R^{[p]}\right) \;\middle|\; \mathrm{mod}\big(i+j, 2^{\frac{k+1}{2}}\big) = 2^{\frac{k-1}{2}}\right\}, & k \text{ odd}; \\ \left\{(i,j) \in G \setminus \left(\cup_{p=1}^{k-1} R^{[p]}\right) \;\middle|\; \mathrm{mod}\big(j, 2^{\frac{k}{2}}\big) = 0\right\}, & k \text{ even}. \end{cases}$$

The maximum number of levels that the $N_x \times N_y$-grid allows for is given by

$$\ell_{\max} = 2\lceil \log_2(\max\{N_x, N_y\})\rceil + 1. \tag{2.1}$$

EXAMPLE 2.1. In this example the RRB-numbering procedure is applied to a matrix $A \in \mathbb{R}^{64\times 64}$ resulting from an 8×8 grid of unknowns. For this matrix A the maximum number of levels is $\ell_{\max} = 2\lceil \log_2(8)\rceil + 1 = 7$ according to equation (2.1). The effect of the RRB-numbering on the ordering can be seen in Figure 2.1. For readability the black nodes are represented by gray squares and the red nodes by white squares. The effect of the RRB-numbering on the sparsity pattern of the matrix $A \in \mathbb{R}^{64\times 64}$ belonging to the 8×8 grid of unknowns is shown in Figure 2.2 for the first four levels.

2.2. The RRB-factorization algorithm. The RRB-method factorizes the matrix A into

$$A = LDL^T + R, \tag{2.2}$$

where L is a lower triangular matrix with unitary diagonal entries, D a diagonal matrix, and R a matrix that contains adjustments resulting from lumping procedures. A detailed description of the RRB-method can be found in [9]. Starting from a 9-point stencil, the factorization (2.2) is performed as follows:

1. Renumber the points and equations corresponding to a red-black ordering; first number all red points. The coefficient matrix then has a 2 by 2 block structure, in which the upper-left block A_{11} gives the interaction between the red points only.
2. The nonzero off-diagonal elements of block A_{11} are first added to the main diagonal and then put to zero (lumping). The modified matrix obtained in this way can be represented by a 5-point stencil.
3. In the modified system of linear equations all red points can be eliminated, which gives again a system of linear equations given by a 9-point stencil. This system of equations has only half the number of unknowns as in Step 1.
4. Go to Step 1, and repeat until only 1 node remains.

ETNA
Kent State University and
Johann Radon Institute (RICAM)

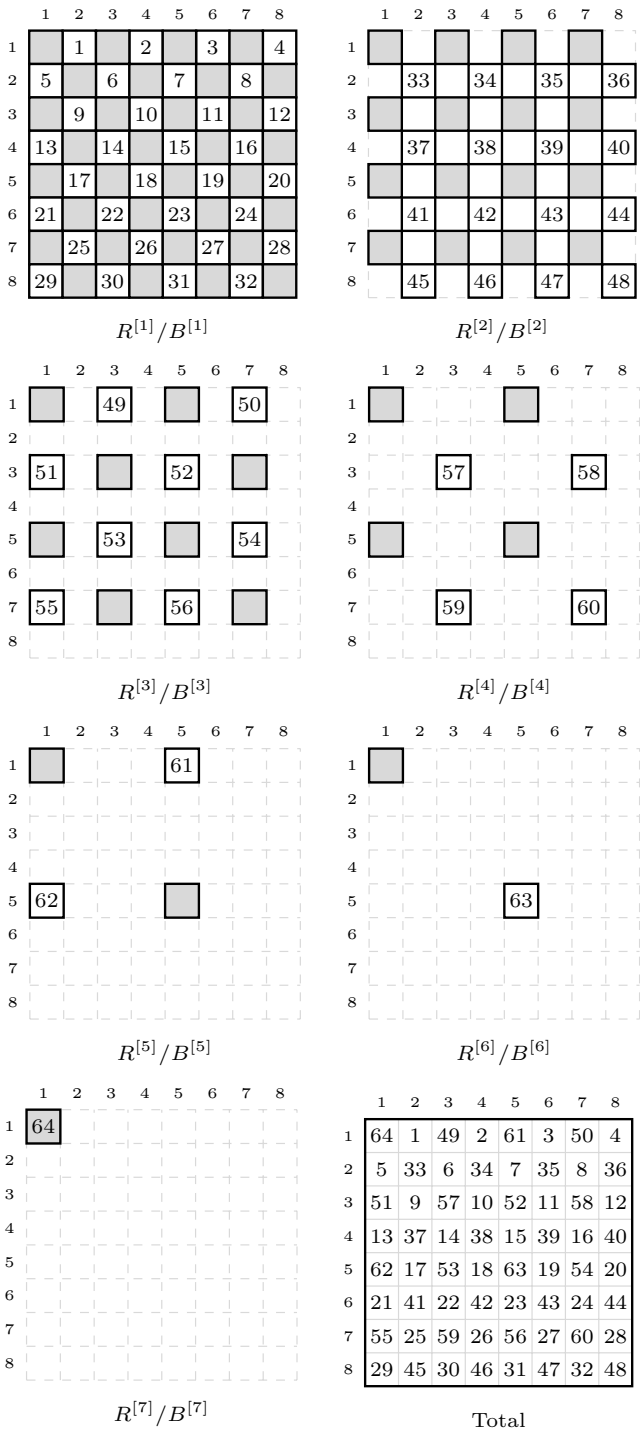
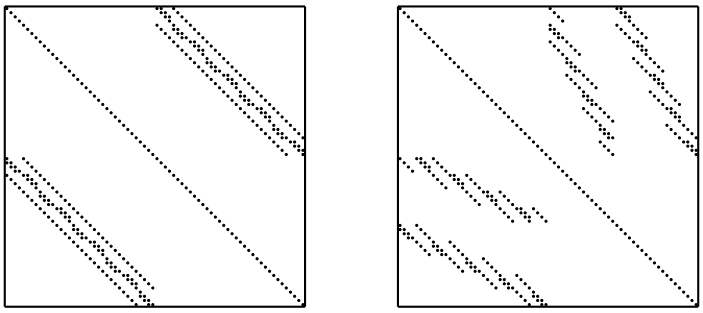
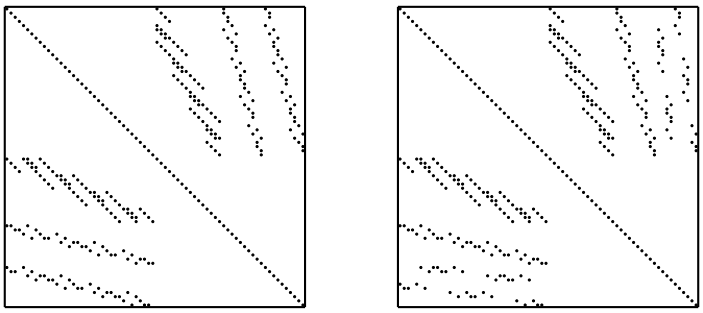


FIG. 2.1. *RRB-numbering for an* 8×8 *grid.*

ETNA
Kent State University and
Johann Radon Institute (RICAM)



(a) RRB-1. (b) RRB-2.



(c) RRB-3. (d) RRB-4.

FIG. 2.2. *Effect of RRB-numbering on the sparsity pattern of A. Here (a) shows the pattern after one level of red-black ordering, (b) shows the pattern after applying two levels of red-black ordering, etc.*

2.2.1. Full RRB versus RRB-ℓ. By reapplying RRB-iterations over and over again, after a certain number of iterations only a single node remains. This is called full RRB. However, it is not needed to go all the way down. After each level of red-black numbering, lumping and elimination of the red nodes in the respective level, one can stop at level $\ell \leq \ell_{\max}$. For the remaining black nodes $B^{[\ell]}$, which have a 9-point dependency structure, a full Cholesky decomposition

$$M_\ell = L_\ell D_\ell L_\ell^T \tag{2.3}$$

is computed. This is called ℓ-step RRB, denoted by RRB-ℓ. Obviously, by stopping earlier the factorization (2.2) becomes more exact. However, the disadvantage is that constructing the Cholesky factor L takes more effort and L has more fill-in. Therefore, this should be done only when the size of the remaining system of equations becomes so small that the costs of a direct solver are not much larger than the costs of the rest of the incomplete factorization.

2.2.2. Starting with a 5-point stencil. In case the RRB-method is applied to a matrix A given by a 5-point stencil, e.g., the 2D Poisson problem, then the elimination of $R^{[1]}$ is exact and accordingly $R = 0$ for the first level.

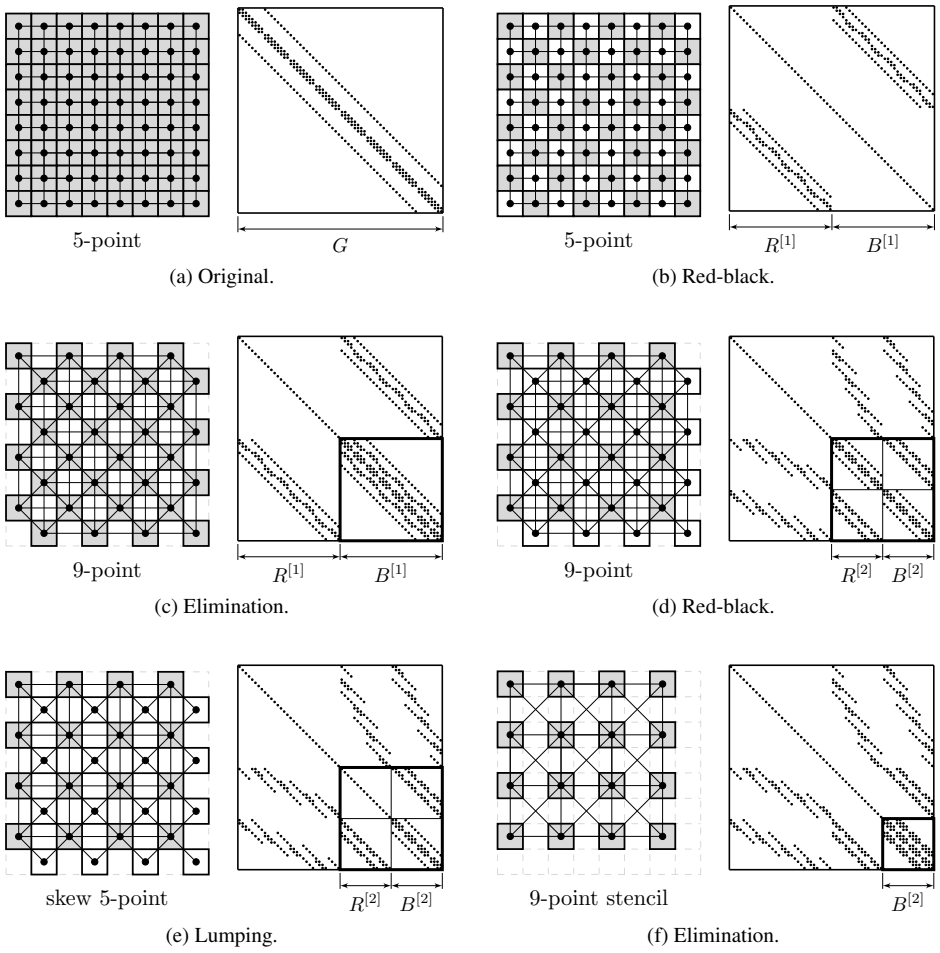
2.2.3. Algorithm. Pseudo code for the RRB-method is provided in Algorithm 1.

EXAMPLE 2.2. In this example the RRB-method is applied to a matrix $A \in \mathbb{R}^{64\times 64}$ resulting from an 8×8 grid, see Figure 2.3. The figure shows the effects of consecutive red-black orderings, lumping, and elimination of red nodes on the dependency structure and sparsity pattern of $L + D + L^T$.

ETNA
Kent State University and
Johann Radon Institute (RICAM)

Algorithm 1 The RRB-method starting with a 9-point stencil.

1. Choose the number of levels $\ell \leq \ell_{\max}$
2. Set $k = 0$
3. **While** $(k \leq \ell)$ **do**
4. Apply red-black ordering: $R^{[k]}/B^{[k]}$
5. Apply lumping procedure to $R^{[k]}$-nodes
6. Eliminate $R^{[k]}$-nodes using 5-point stencils for $R^{[k]}$-nodes
7. $k = k + 1$
8. **End while**
9. Make Cholesky decomposition for remaining 9-point stencil: $M_\ell = L_\ell D_\ell L_\ell^T$



(a) Original. (b) Red-black. (c) Elimination. (d) Red-black. (e) Lumping. (f) Elimination.

FIG. 2.3. *Effects of the RRB-method for $A \in \mathbb{A}^{64\times 64}$ with $\ell = 2$ on the dependency structure and the sparsity pattern of $L + D + L^T$. For the remaining $B^{[2]}$-nodes a full Cholesky decomposition is computed.*

2.3. The RRB-method used as a preconditioner in PCG. The matrix A is factorized as $A = LDL^T + R$. As preconditioner for the PCG-algorithm the matrix

$$M = LDL^T \approx A$$

ETNA
Kent State University and
Johann Radon Institute (RICAM)

is taken. The smaller the numbers in R in absolute value are, the better M resembles A. The combination of the PCG-algorithm and the RRB-method as preconditioner is called the RRB-solver.

As remarked earlier, starting from a 5-point stencil, e.g., the 2D Poisson problem, the elimination of $R^{[1]}$-nodes is exact. Hence, in this case PCG can be applied to the resulting 1st Schur complement S_1 instead of the entire matrix A. This is beneficial for the total amount of computational work. Since S_1 consists of only the $B^{[1]}$-nodes, the number of flops for computing the vector updates and inner products in the PCG-algorithm is reduced by a factor two. The number of flops in the matrix-vector product $\mathbf{q} = S_1\mathbf{p}$ remains the same as the matrix-vector product with A, because the matrix S_1 is now given by a 9-point stencil instead of by a 5-point stencil.

2.4. Spectral condition number of RRB-l. The convergence rate of PCG depends on the spectrum of matrix $M^{-1}A$. Since the preconditioner M is an approximation of the system matrix A, the condition number of $M^{-1}A$ is smaller than that of A. This gives a sharper upper bound of the error (1.2) and therefore most likely a faster convergence.

In [3], [6], and [12] the RRB-ℓ preconditioner is investigated in detail for the Poisson problem with Dirichlet boundary conditions on a 2D uniform grid with $n \times n$ unknowns and where n is of the form $n = 2^\ell - 1$. Different upper bounds can be found in the aforementioned literature. Notay [12] gives an upper bound:

$$\kappa(M^{-1}A) \leq \frac{\sqrt{5}(\sqrt{5}-1)^{\ell-1}}{1+(-1)^\ell\left(\frac{3-\sqrt{5}}{2}\right)^{\ell-1}} \approx 1.8 \cdot 1.23^\ell.$$

Since the mesh spacing $h = 1/(n+1)$ and $\ell = \log_2(1/n) \approx \log_2 h^{-1}$, alternatively,

$$\kappa(M^{-1}A) \leq 1.8\, h^{-0.306}.$$

2.5. Application of the preconditioner. At each iteration of the PCG-algorithm, the system $M\mathbf{z} = \mathbf{r}$ needs to be solved for $\mathbf{z}$. The preconditioning matrix M is factorized as $M = LDL^T$. Therefore, $M\mathbf{z} = \mathbf{r}$ can be solved efficiently in three steps as follows. Set $\mathbf{y} := L^T\mathbf{z}$ and $\mathbf{x} := DL^T\mathbf{z} = D\mathbf{y}$, then:

1. solve $\mathbf{x}$ from $L\mathbf{x} = \mathbf{r}$ using forward substitution;
2. compute $\mathbf{y} = D^{-1}\mathbf{x}$;
3. solve $\mathbf{z}$ from $L^T\mathbf{z} = \mathbf{y}$ using backward substitution.

Each of the three steps is computationally cheap.

2.5.1. Algorithm. For memory efficiency a single vector $\mathbf{z}$ is used instead of the three vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$. Moreover, if $\ell \leq \ell_{\max}$ then at the final level ℓ a full Cholesky decomposition (2.3) is made for the remaining level of black nodes $B^{[\ell]}$. Pseudo code is provided below, see Algorithm 2.

EXAMPLE 2.3. For a matrix $A \in \mathbb{R}^{64\times 64}$ resulting from an 8×8 grid of unknowns the forward substitution steps are visualized in Figure 2.4 and Figure 2.5.

As indicated by Algorithm 2, both the forward and backward substitution are performed level-wise in two phases. First, the $\mathbf{z}$-values at $B^{[k]}$-nodes are updated using the skew 5-point stencils ($\times$) at the $R^{[k]}$-nodes, see Figure 2.4. Second, the $\mathbf{z}$-values at $B^{[k+1]}$-nodes are updated using the straight 5-point stencils ($+$) at the $R^{[k+1]}$-nodes, see Figure 2.5. The backward substitution is done level-wise in the same manner, yet in the reverse order.

ETNA
Kent State University and
Johann Radon Institute (RICAM)

Algorithm 2 Application of the RRB preconditioner.

1. Given number of levels $\ell \leq \ell_{\max}$
2. Set $k = 2$
3. **While** $(k \leq \ell)$ **do** % forward subs.
4. Update $\mathbf{z}$-values at $B^{[k]}$-nodes using $R^{[k]}$-nodes % skew 5-point ($\times$)
5. **If** $(k + 1 == \ell)$ **then**
6. **break**
7. **End if**
8. Update $\mathbf{z}$-values at $B^{[k+1]}$-nodes using $R^{[k+1]}$-nodes % 5-point ($+$)
9. $k = k + 2$
10. **End while**
11. Update $\mathbf{x}$-values at $B^{[1]}$-nodes % diagonal scaling
12. Solve $L_\ell D_\ell L_\ell^T \mathbf{x}_\ell = \mathbf{z}_\ell$ and update $\mathbf{x}$-values in level ℓ % final level exact
13. Set $k = \ell$
14. **While** $(k \geq 2)$ **do** % backward subs.
15. Update $\mathbf{z}$-values at $R^{[k]}$-nodes using $B^{k]}$-nodes % 5-point ($+$)
16. **If** $(k - 1 == 2)$ **then**
17. **break**
18. **End if**
19. Update $\mathbf{z}$-values at $R^{[k-1]}$-nodes using $B^{[k-1]}$-nodes % skew 5-point ($\times$)
20. $k = k - 2$
21. **End while**

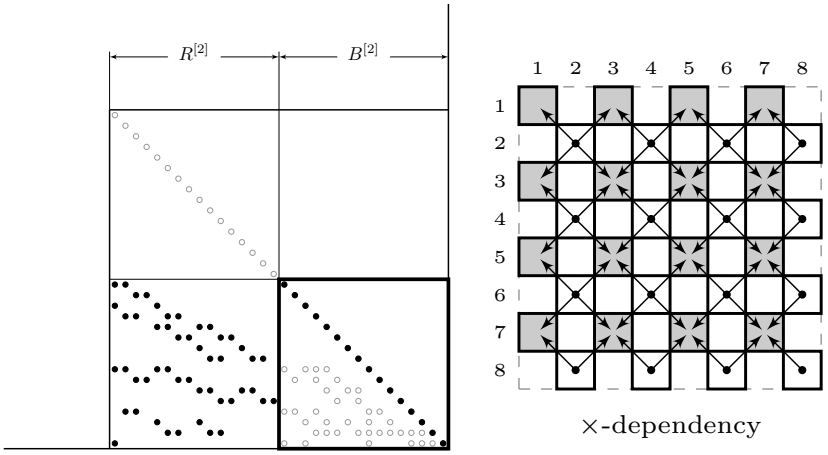


FIG. 2.4. *Forward substitution phase 1: skew 5-point stencils ($\times$).*

3. Parallel implementation of the RRB-solver. To obtain a parallel implementation of the RRB-solver, all operations of the CG-algorithm, i.e., the inner products, the vector updates, and the matrix-vector $\mathbf{q} = S_1\mathbf{p}$, where S_1 is the first Schur complement, can readily be parallelized on shared memory machines [11]. Secondly, the construction of the preconditioner, i.e., $M = LDL^T$, can be performed *level-wise* in parallel on shared memory machines. From Algorithm 1 it can be seen that per level each of the operations lumping, elimination, and substitution can be performed fully in parallel. Finally, the application of the preconditioner, i.e., solving $M\mathbf{z} = \mathbf{r}$ for $\mathbf{z}$, can be performed *level-wise* in parallel on shared

ETNA
Kent State University and
Johann Radon Institute (RICAM)

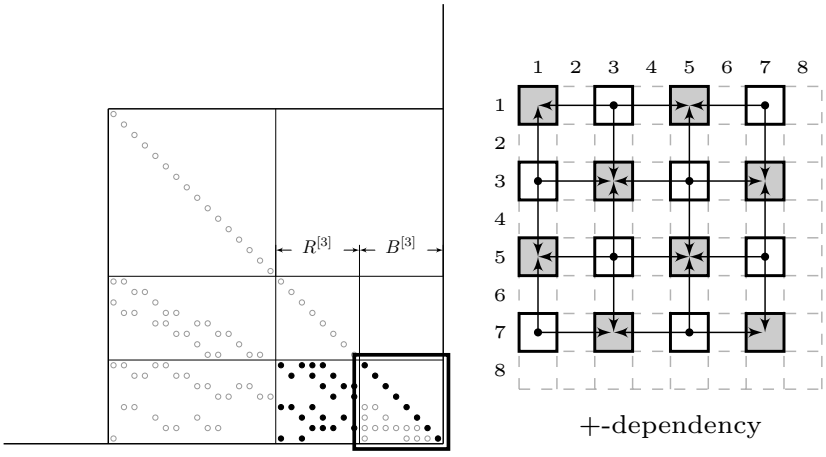


FIG. 2.5. *Forward substitution phase 2: straight 5-point stencils (+).*

memory machines as well. This can be seen from Algorithm 2 as well as from Figure 2.5, both indicate that at each level the gray nodes can be updated fully in parallel. Figure 3.1 illustrates this even more clearly. The gray areas indicate the blocks where fill-in has been lumped, and where the computations can be done fully in parallel.

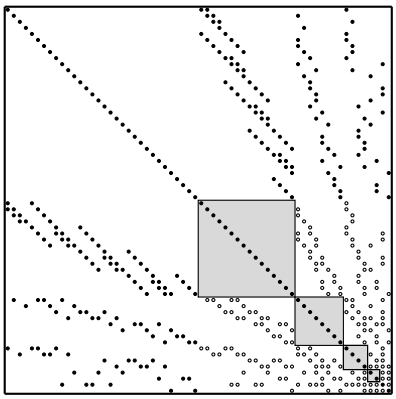


FIG. 3.1. *Sparsity pattern of $L + D + L^T$. The gray areas indicate parallel blocks.*

As forward and backward substitution are inherently sequential, it is not possible to compute the different levels (the gray blocks) in parallel as well. Fortunately, however, as the number of unknowns decreases fast, namely by a factor 2 per level, so does the amount of work. All together, it can be concluded that the RRB-solver can be parallelized very well on shared memory machines.

3.1. A key idea to maximize bandwidth. In this section we discuss how to actually implement the RRB-solver on modern parallel architectures such as the GPU. The RRB-solver consists of BLAS-1 and BLAS-2 routines and is therefore a bandwidth-bound, a data intensive algorithm, i.e., the ratio

$$r = \frac{\text{Number of memory operations}}{\text{Number of floating point operations}}$$

ETNA
Kent State University and
Johann Radon Institute (RICAM)

is large. For already relatively small problem sizes, the amount of data is such that the problem does not fit in the (fast) cache of most modern computers anymore. Hence throughout the RRB-algorithm most data is read from and written to the global memory over and over again. A main concern thus is how to achieve high global memory bandwidth throughout the algorithm.

This is certainly not trivial for the RRB-solver because of the repeated red-black orderings, which can be seen from Figure 2.1. A naïve storage of the data implies that all the data required for the vector-updates, inner products, and the matrix-vector product would be read and written with a stride of two. This follows from the fact that in case of the RRB-solver the underlying PCG-algorithm operates on the $B^{[1]}$-nodes only. Even worse, during application of the preconditioner step, i.e., solving $M\mathbf{z} = \mathbf{r}$ for $\mathbf{z}$, the data would be accessed with increasing stride: $2, 4, 8, 16, \ldots$. It is well-known that reading data from and writing data to the global memory with a stride leads to poor performance. Even though the amount of work decreases fast for each next level, at the first few levels already too much bandwidth would get wasted and the RRB-solver would suffer from poor performance.

Therefore, to guarantee maximal bandwidth throughout the algorithm, for the first few (the finest) levels a different storage scheme is proposed, see Figure 3.2. This storage scheme is called the $r_1/r_2/b_1/b_2$-storage scheme [8].

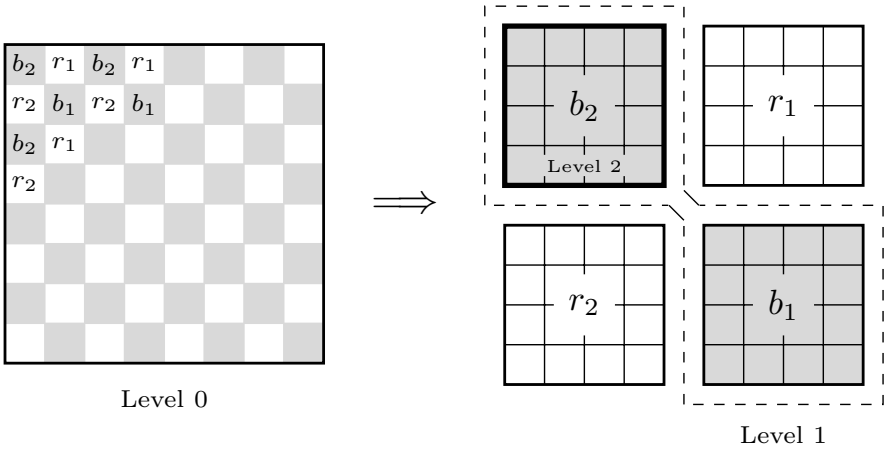


FIG. 3.2. *Restorage of data in four groups: r_1-, b_1-, r_2- and b_2-nodes. The b_1- and b_2-nodes together form the intermediate levels with skew meshes ($\times$). The b_2-nodes form the next coarser level ($+$).*

Comparison of Figure 2.1 and Figure 3.2 shows: the $R^{[1]}$-nodes are divided into r_1- and r_2-nodes and the $B^{[1]}$-nodes are divided into b_1- and b_2-nodes. Hence the b_1- and b_2-nodes are the nodes to which the majority of the PCG-algorithm is applied to. The first important observation is now that, by storing the $B^{[1]}$ nodes into these two new arrays of b_1- and b_2-nodes, the data can always be accessed in a coalesced manner, without a stride.

The second important observation is that the b_2-nodes form the next coarser grid, and that, on this coarser grid, the $r_1/r_2/b_1/b_2$-storage scheme can be reapplied. In general, the odd levels k are stored into $b_1^{[k]}$- and $b_2^{[k]}$-nodes, and the even levels k are stored into the $b_2^{[k]}$-nodes. Given the problem size, the layout of storage schemes can be determined a priori.

It is thus very beneficial for performance to store the data used by the RRB-solver as much as possible in the $r_1/r_2/b_1/b_2$-storage scheme, i.e., the 5 vectors in the PCG-algorithm $(\mathbf{r}, \mathbf{x}, \mathbf{z}, \mathbf{p}, \mathbf{q})$ and the $5 + 9$ vectors that describe the system matrix S_1 and the preconditioner matrix M, respectively. Actually, the matrix M and the vector $\mathbf{z}$ occurring in the preconditioner

ETNA
Kent State University and
Johann Radon Institute (RICAM)

step $M\mathbf{z} = \mathbf{r}$ are stored in a recursive $r_1/r_2/b_1/b_2$-storage scheme. Although the proposed scheme is basically a renumbering and reordering of the grid points, for performance it is needed to use multiple physical arrays to ensure that all data describing M and $\mathbf{z}$ can be accessed in a coalesced manner. This is at most just $1 + 1/4 + 1/16 + \ldots = 4/3$ times as expensive as the naive storage. The $r_1/r_2/b_1/b_2$-storage scheme can be used virtually for free during the preconditioner step $M\mathbf{z} = \mathbf{r}$. This can be seen from Figure 2.5: in case of application of the preconditioner, the updated $\mathbf{z}$-values at level k are directly written into the $r_1/r_2/b_1/b_2$-arrays of level $k+1$ in case of forward substitution and vice versa in case of backward substitution. Hence, by using the $r_1/r_2/b_1/b_2$-storage scheme, maximal bandwidth can be obtained for all routines in the PCG-algorithm.

4. Implementation and experimental setup.

4.1. Hardware. All experiments are performed on a system having a Xeon E5-1620 processor, 32 GB of memory and a GeForce Titan X graphics card. The OS is Linux Debian 8.8 (64-bit) with kernel 4.9.23. The RRB-solver is implemented in C++ with OpenMP for parallelization as well as in CUDA C. The code is compiled with GCC version 4.9.2 and NVVC 7.5.17. All experiments are performed in double-precision.

4.2. Test problem. Similar to [3], [6], and [12], as a test problem the 2D Poisson problem with Dirichlet boundary conditions on a 2D uniform grid is taken:

$$\begin{aligned} -\Delta u &= f(x,y) \quad \text{on} \quad \Omega = (0,1) \times (0,1), \\ u(x,y) &= 0 \quad \text{on} \quad \partial\Omega, \end{aligned}$$

having $n \times n$ unknowns, and where n is of the form $n = 2^\ell - 1$. The right-hand side f is computed such that $u(x,y) = x(x-1)y(y-1)\exp(xy)$.

4.3. Condition number and number of PCG-iterations. The condition number κ of the preconditioned system (1.3) is given by

$$\kappa := \kappa(M^{-1}A) = \frac{\lambda_{\max}(M^{-1}A)}{\lambda_{\min}(M^{-1}A)},$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the largest and smallest eigenvalues, respectively. Since for the RRB-preconditioner $\lambda_{\min} \approx 1$, we have [3]:

$$\kappa \approx \lambda_{\max}(M^{-1}A).$$

The required number of PCG-iterations depends on:

(i) the termination criterion, i.e., the demanded accuracy given by a parameter tol;
(ii) the choice of the initial guess $\mathbf{x}_0$;
(iii) the number of RRB-levels ℓ (accuracy of preconditioner);

In our experiments we have used the termination criterion:

$$\|\mathbf{r}_i\|_{M^{-1}} / \|\mathbf{r}_0\|_{M^{-1}} \leq tol,$$

with $tol := 10^{-6}$. As initial guess we have taken $\mathbf{x}_0 = 0$. The number of RRB-levels ℓ is varied in the experiments.

4.4. Four implementations. Four implementations of the RRB-solver are used:

- **S1:** Sequential solver in C++, uses naïve storage scheme for all ℓ levels;
- **S2:** Sequential solver in C++, uses $r_1/r_2/b_1/b_2$-storage scheme for the first $2g$ levels; calls S1 for the remaining $\ell - 2g$ levels;

ETNA
Kent State University and
Johann Radon Institute (RICAM)

- **S3:** Parallel solver in C++/OpenMP, uses $r_1/r_2/b_1/b_2$-storage scheme for the first $2g$ levels; calls S1 for the remaining $\ell - 2g$ levels;
- **S4:** Parallel solver in CUDA C, uses $r_1/r_2/b_1/b_2$-storage scheme for the first $2g$ levels; calls S1 for the remaining $\ell - 2g$ levels.

For all solvers the desired level ℓ can be set and for the solvers S2, S3, and S4 the desired number of $r_1/r_2/b_1/b_2$-grids g can be set as well. S1 is a complete sequential implementation of the RRB-solver. The solvers S2, S3, and S4 only take care of the first $2g$ levels and rely on solver S1 for the remaining levels. If $\ell \geq \ell_{\max}$ then $\ell := \ell_{\max}$ and the final grid consists of just 1 node; if $\ell < \ell_{\max}$ then a complete Cholesky decomposition is made for the remaining 9-point stencil. On the coarsest grid solver S1 takes care of the remaining problem, see Line 12 in Algorithm 2. This is done exactly with a sequential band solver.

5. Results.

5.1. Condition numbers and number of PCG-iterations. In Figure 5.1 the condition number κ is shown for different numbers of RRB-levels ℓ and different problem sizes. In

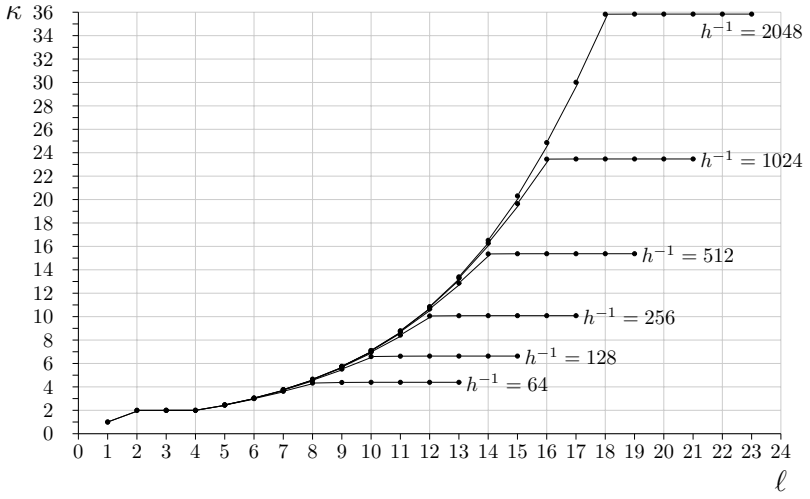


FIG. 5.1. *Condition number κ versus number of RRB-levels ℓ for various problem sizes.*

Figure 5.2 the corresponding number of PCG-iterations is shown. The maximal number of RRB-levels $\ell_{\max}$ depends on the problem size, see Equation (2.1). The figures show that:

1. The larger ℓ, the larger κ and the larger the required number of PCG-iterations. This makes perfect sense as the accuracy of the preconditioner M decreases with the number of RRB-levels ℓ; beyond a certain level the accuracy of the preconditioner M is hardly effected anymore, hence the horizontal slope in the figures;
2. For the 2D Poisson test problem, the RRB-solver scales very well: both the condition number κ and the required number of PCG-iterations hardly increase with mesh refinement. Going from $h = \frac{1}{128}$ to $h = \frac{1}{2048}$ only gives a doubling of iterations.

The RRB-solver is thus very efficient in itself: low condition numbers, low numbers of PCG-iterations, and very good scaling behaviour.

5.2. Timings and speedup. The results in this section are all for the test problem with a fixed problem size of 2047×2047 unknowns. In Figure 5.3 the computation time is shown for the four implementations S1 to S4, for three numbers of RRB-levels ℓ, and for different

ETNA
Kent State University and
Johann Radon Institute (RICAM)

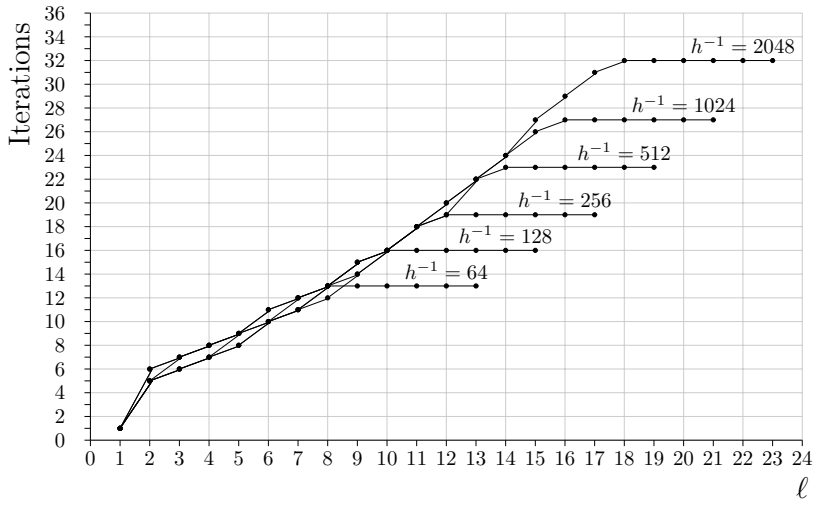


FIG. 5.2. *Number of PCG-iterations versus number of RRB-levels ℓ for various problem sizes.*

numbers of $r_1/r_2/b_1/b_2$-grids. As an example: S2-2 means solver S2 with $g = 2$. In Figure 5.4 the corresponding speedups are shown with respect to solver S1. The figures show:

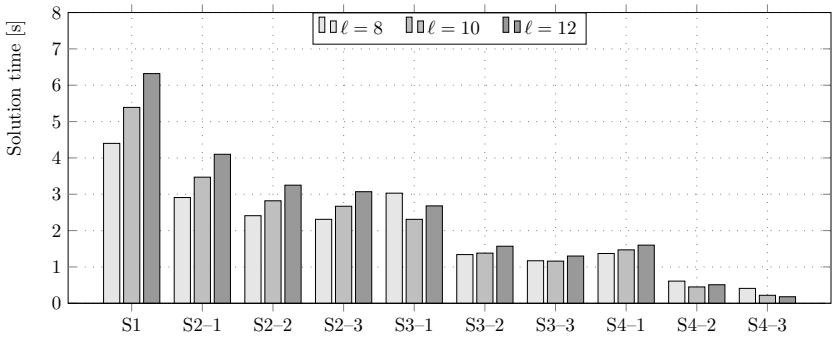


FIG. 5.3. *Solution time of RRB-solver for the four implementations and different number of RRB-levels ℓ.*

1. Introduction of the $r_1/r_2/b_1/b_2$-storage scheme instead of the naïve storage scheme gives already a speedup factor 2.1 (with $\ell = 12$ and $g = 3$);
2. Computation with 4 cores of the Xeon E5-1620 via OpenMP on top of introduction of the $r_1/r_2/b_1/b_2$-storage scheme gives a speedup factor 4.9 (with $\ell = 12$ and $g = 3$);
3. Computation with all cores of the Titan X via CUDA on top of introduction of the $r_1/r_2/b_1/b_2$-storage scheme gives a speedup factor 35.1 (with $\ell = 12$ and $g = 3$).

In Figure 5.5 and Figure 5.6 more timings are shown for solver S4. In Figure 5.5 the time is shown to compute the factorization (2.2) for various g and various ℓ. In Figure 5.6 the time to solve the linear system (1.3) is shown. The figures show:

1. The higher the number of $r_1/r_2/b_1/b_2$-grids g the faster the factorization; however, above $g = 3$ and/or $\ell = 13$ the time reduction is negligible;
2. The higher the number of $r_1/r_2/b_1/b_2$-grids g the faster the solution; however, above $g = 3$ the time reduction is negligible; moreover, there is an 'optimal' number of

ETNA
Kent State University and
Johann Radon Institute (RICAM)

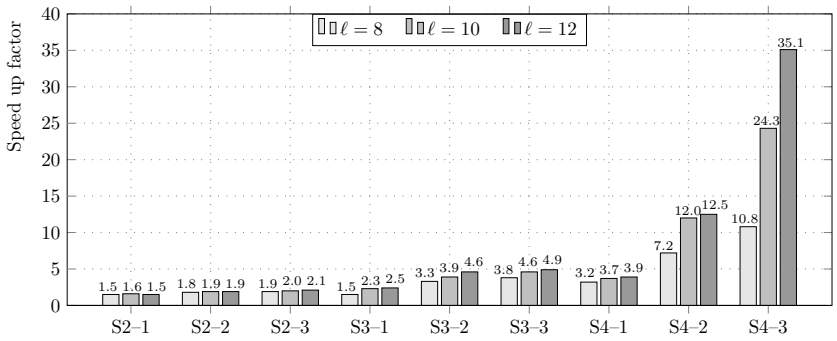


FIG. 5.4. *Speedup relative to S1 for different number of RRB-levels ℓ.*

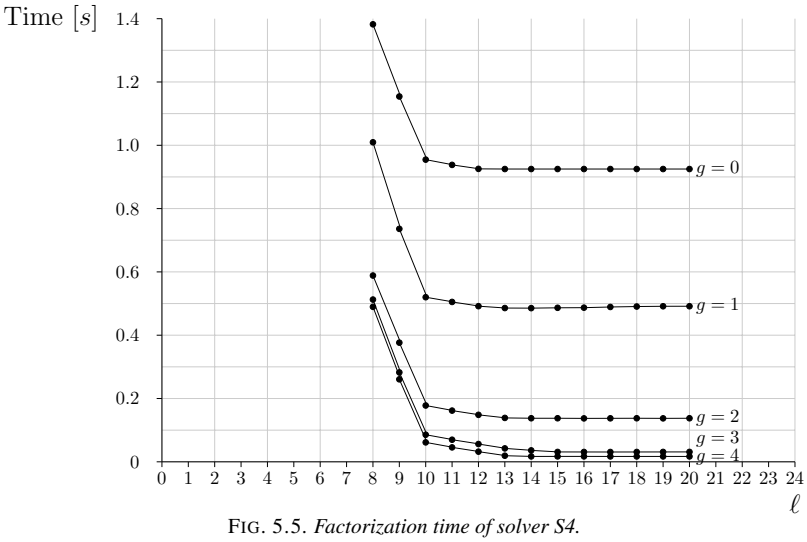


FIG. 5.5. *Factorization time of solver S4.*

RRB-levels ℓ: first the solution time decreases until a certain number of RRB-levels ($\ell = 8$ for $g = 0$ or $\ell = 12$ for $g = 4$) due to the fact that the remainder of the Cholesky factor is smaller; thereafter the solution time goes up again due to the fact that the number of iterations increases.

5.3. Throughput, flop rate, and time breakdown. All results in this section are for the test problem having 2047×2047 unknowns. In Table 5.1 the time breakdown of solver S4 is shown. The table lists the different kernels in the PCG-algorithm and their corresponding time and performance. We observe that:

1. The solver is well-balanced in time; all kernels contribute equally, in the same order, to the overall computation time;
2. The throughput rates are really good: the theoretical peak of the Titan X is 313 GiB/s. We observe that the throughput rates of all kernels are very good (above 80% of the theoretical peak). The `solver` and `matvec` kernels operate even close to and a bit beyond the theoretical peak, respectively.

The data transfers from and to the device (the vectors $\mathbf{x}$ and $\mathbf{b}$) are performed rather slowly (only 2.6 GiB/s) and therefore take 30.4% of the time. Most likely, this is caused by the rather

ETNA
Kent State University and
Johann Radon Institute (RICAM)

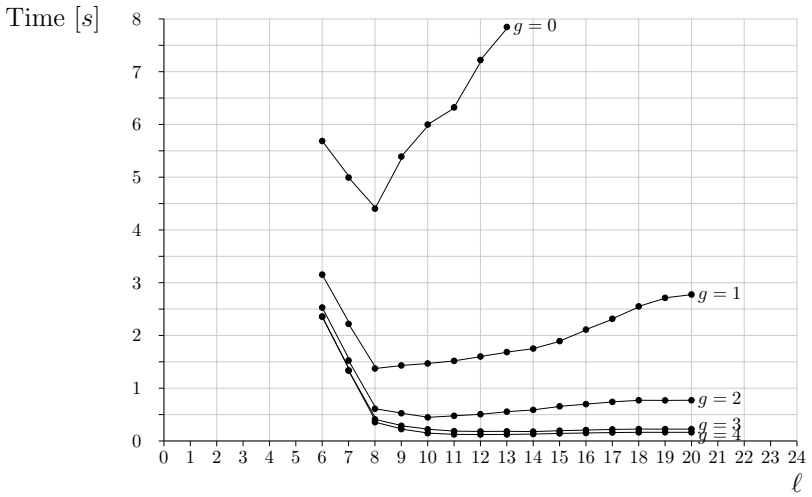
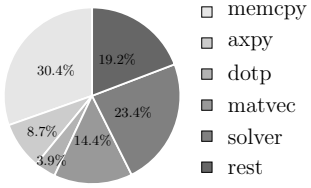


FIG. 5.6. *Solution time of solver S4.*

old motherboard with low PCI-express transfer rates in our test system; an upgrade of the motherboard would most likely lead to a significant reduction in time.

TABLE 5.1
*Time breakdown of solver S4 for $g = 4$ and $\ell = 12$. *Total time for 19 iterations. **GPU part only; CPU part is accumulated in* `rest`.

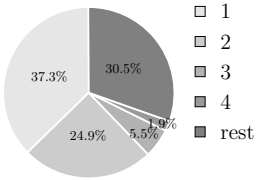
Kernel	Time* (ms)	%	GiB/s	GFlops/s
`memcpy` (3×)	38.7	30.4	2.6	–
`axpy` (3×)	11.0	8.7	246.1	30.9
`dotp` (2×)	4.9	3.9	246.3	31.0
`matvec`	18.3	14.4	325.0	36.4
`solver`**	29.8	23.4	299.1	24.5
`rest`	24.4	19.2	–	–
Total	127.1	100.0		



Finally, in Table 5.2 the time breakdown is shown for the forward substitution part of the `solver` kernel.

TABLE 5.2
Time breakdown of S4's forward substitution (= part of `solver`*) for $g = 5$ and $\ell = 12$. *Levels 9 to 12 and the remainder are performed by S1 (on the CPU).*

Grid	RRB-levels	$cx \times cy$	Time (μs)	%
1	$1 + 2$	1024×1024	258	37.3
2	$3 + 4$	512×512	172	24.9
3	$5 + 6$	256×256	38	5.5
4	$7 + 8$	128×128	13	1.9
rest	$9 - 12$ and rem*		211	30.5
Total			692	100.0



ETNA
Kent State University and
Johann Radon Institute (RICAM)

We observe that:

1. The first two grids 1 and 2 (the first four RRB-levels) take the most computation time (37.3% and 24.9%, respectively). Since the `solver` kernel runs close to the theoretical peak of the device (see Table 5.1) one cannot gain much speed anymore;
2. The last two grids 3 and 4 only take a fraction of the time (5.5% and 1.9% respectively); this shows that the typical Multigrid issue of idle cores on coarser levels does not really have an impact in our solver;
3. The levels 9 to 12 and the remainder are performed by solver S1 on the CPU which takes 30.5% of the overall time. Although this is relative much due to a rather old CPU, it does not really matter, since the finest levels take an equal amount of time; see the observation in item 1.

5.4. Comparison with multigrid. In this section we compare the RRB-solver with the default algebraic Multigrid (AMG) solver from the PARALUTION [13] library, a high-performance C++ library that offers various sparse iterative solvers and preconditioners for multicore CPU and GPU devices. A comparison with Multigrid is chosen, because Multigrid-type solvers are known to be very efficient as well for the type of problems considered in this paper. It was shown that, throughout, the performance of the RRB-solver is near-optimal, therefore, the only remaining two options to gain speed are when the required number of iterations in the PCG-algorithm can be reduced, or by using a different type of solver. From our experience, most (if not all) other PCG-type solvers will perform (much) worse because they typically require significantly more iterations until convergence. Therefore, we confine ourselves to a comparison with the Multigrid method. It is also important to note that FFT-solvers cannot be used directly as the system matrix A does not have constant diagonals.

Although for structured, rectangular grids, such as our 2D Poisson-type problem, geometric Multigrid (GMG) seems to be the better alternative, its application within PARALUTION comes with several additional requirements, making it rather complicated to use. For GMG it is required that all operations are defined beforehand, such as explicit definition of the restriction/prolongation operations and smoother for each level, and the coarse grid solver. A major advantage of AMG over GMG is that it can be used as a black-box solver, as there is no need to explicitly define all operations. Similarly, the RRB-solver can be used as a black-box solver, and we show here that it can outperform AMG as a black-box solver for Poisson-type problems.

The default AMG solver in PARALUTION uses smoothed aggregation interpolation, a fixed-point iteration solver, and two-color Gauss-Seidel as a smoother. As of today, a GPU implementation is missing for the construction of AMG, and the construction is therefore performed on the host (CPU). The solution steps however are performed on the GPU. PARALUTION's AMG solver also supports the diagonal (DIA) storage format, which is the most efficient format to use in our application. The AMG solver is compared with our S4-4 solver, with $\ell = 12$ fixed. Both solvers were tested on the same hardware, see Section 4.1.

Again the 2D Poisson test problem, see Section 4.2, is solved for different problem sizes and $tol = 10^{-6}$. In Table 5.3 the results are listed. The setup time is the total time taken by the solver to initialize all memory (on both CPU and GPU) and to setup/construct the solver. From the table the following can be observed:

1. Setting up the S4-4 solver on the CPU is 7 times faster than setting up the AMG solver; setting up the S4-4 solver on the GPU is even 10 times faster than setting up the default AMG solver;
2. The S4-4 solver outperforms the default AMG solver by a factor 7 for the solve step;

ETNA
Kent State University and
Johann Radon Institute (RICAM)

TABLE 5.3
Solver comparison: PARALUTION's default AMG solver versus the S4-4 solver.

	AMG			S4-4, $\ell = 12$			
Problem		Setup [s]	Solve [s]		Setup [s]		Solve [s]
size h^{-1}	#Iter	CPU	GPU	#Iter	CPU	GPU	GPU
64	7	0.075	0.085	13	0.005	0.012	0.007
128	8	0.105	0.128	16	0.007	0.028	0.010
256	9	0.121	0.162	19	0.026	0.042	0.010
512	10	0.317	0.241	20	0.076	0.073	0.018
1024	13	1.157	0.498	20	0.308	0.145	0.042
2048	16	4.511	1.084	19	0.725	0.426	0.142

3. The number of iterations taken by S4-4 (with $\ell = 12$) is nearly as low as the number of iterations taken by AMG. This shows that the RRB-solver scales nearly as good as Multigrid;
4. For larger problem sizes ($h^{-1} > 256$) and fixed $\ell = 12$ the number of RRB-iterations does not increase.

Although AMG could be tuned to perform better, and GMG may provide even better performance than AMG (in particular for construction of the solver), our comparison study already clearly demonstrates that the GPU RRB-solver is really fast, and that it can challenge or, for specific type of problems, even outperform existing Multigrid-type solvers from acknowledged high-performance computing libraries.

6. Conclusions. This paper addressed an efficient implementation of the RRB-solver. The RRB-solver is a PCG-type solver based on the RRB-method for the incomplete factorization of the preconditioner. Literature and a comparison study by us shows that the RRB-method is very efficient in itself and that it scales very well with mesh refinement, nearly as good as Multigrid. Besides being very efficient in itself, this paper demonstrates that the RRB-method also allows for an efficient parallelization. A clever storage scheme is key in the efficient parallelization. Using the so-called $r1/r2/b1/b2$-storage scheme both the construction and the application of the preconditioner are efficiently parallelized on both multi-core CPU using C++/OpenMP and the GPU using CUDA C. The $r1/r2/b1/b2$-storage format can be used in different solvers as well to obtain an efficient implementation on the GPU, e.g., (geometric) Multigrid may benefit from this. Performance studies on the 2D Poisson test problem showed that the performance of the RRB-solver is very good overall: no wasted throughput, no bottleneck, and a well-balanced algorithm in terms of time spent per routine in the PCG-algorithm. On the GPU the RRB-solver reaches a throughput close to the theoretical peak of the device and a speedup factor of more than 30 compared to a sequential implementation on the CPU. A comparison with algebraic Multigrid from a recognized high-performance computing library showed that the RRB-solver can outperform it by a factor 7 to 10, which again demonstrates that the RRB-solver can be implemented very efficiently on the GPU.

REFERENCES

[1] P. R. AMESTOY, I. S. DUFF, J. Y. L'EXCELLENT, AND J. KOSTER, *MUMPS: a general purpose distributed memory sparse solver*, in Proc. PARA2000, 5th Int. Workshop on Appl. Parallel Comp., T. Sørevik, F. Manne, A. H. Gebremedhin, and R. Moe, eds., vol. 1947 of Lecture Notes in Comput. Sci., Springer, Berlin, 2000, pp. 122–131.

[2] O. AXELSSON AND V. EIJKHOUT, *The nested recursive two-level factorization method for nine-point difference matrices*, SIAM J. Sci. Statist. Comput., 12 (1991), pp. 1373–1400.

ETNA
Kent State University and
Johann Radon Institute (RICAM)

[3] C. W. BRAND, *An incomplete-factorization preconditioning using repeated red-black ordering*, Numer. Math., 61 (1992), pp. 433–454.
[4] C. W. BRAND AND Z. HEINEMANN, *A new iterative solution technique for reservoir simulation equations on locally refined grids*, SPE Reservoir Eng., 5 (1990), pp. 555–560.
[5] A. M. BRUASET, *Preconditioners for Discretized Elliptic Problems*, PhD. Thesis, Department of Informatics, University of Oslo, 1992.
[6] P. CIARLET, JR., *Repeated red-black ordering: a new approach*, Numer. Algorithms, 7 (1994), pp. 295–324.
[7] T. A. DAVIS, *Algorithm 832: UMFPACK V4.3: an unsymmetric-pattern multifrontal method*, ACM Trans. Math. Software, 30 (2004), pp. 196–199.
[8] M. DE JONG, A. VAN DER PLOEG, A. DITZEL, AND C. VUIK, *Real-time computation of interactive waves on the GPU*, in 15th Numerical Towing Tank Symposium, Cortona, Italy, V. Bertran, E. Campana, eds., Curran Assoc., Red Hook, 2012, pp. 111–116.
[9] M. DE JONG AND C. VUIK, *GPU implementation of the RRB-solver*, Tech. Rep. 16-06, Delft University of Technology, Delft, Netherlands, 2016.
[10] I. GUSTAFSSON, *A class of first order factorization methods*, BIT, 18 (1978), pp. 142–156.
[11] R. LI AND Y. SAAD, *GPU-accelerated preconditioned iterative linear solvers*, J. Supercomputing, 63 (2013), pp. 443–466.
[12] Y. NOTAY AND Z. O. AMAR, *A nearly optimal preconditioning based on recursive red-black orderings*, Numer. Linear Algebra Appl., 4 (1997), pp. 369–391.
[13] PARALUTION LABS, *PARALUTION v1.1.0*, 2016. http://www.paralution.com/
[14] Y. SAAD, *Krylov subspace methods on supercomputers*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 1200–1232.
[15] O. SCHENK AND K. GÄRTNER, *Solving unsymmetric sparse systems of linear equations with PARDISO*, in Computational Science—ICCS 2002, Part II (Amsterdam), P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, eds., vol. 2330 of Lecture Notes in Comput. Sci., Springer, Berlin, 2002, pp. 355–363.
[16] P. N. SWARZTRAUBER AND R. A. SWEET, *Algorithm 541: FISHPACK: efficient fortran subprograms for the solution of separable elliptic partial differential equations [d3]*, ACM Trans. Math. Softw., 5 (1979), pp. 352–364.
[17] H. A. VAN DER VORST, *Iterative Krylov Methods for Large Linear Systems*, Cambridge University Press, Cambridge, 2003.