The definitive version was published in Parallel Computing Volume 57, September 2016
http://dx.doi.org/10.1016/j.parco.2016.05.014

Parallel Iterative Refinement Linear Least Squares Solvers based on All-Reduce Operations

Karl E. Prikopa, Wilfried N. Gansterer, Elias Wimmer

University of Vienna, Faculty of Computer Science, Vienna, Austria

Abstract

We present the novel parallel linear least squares solvers ARPLS-IR and ARPLS-MPIR for dense overdetermined linear systems. All internode communication of our ARPLS solvers arises in the context of all-reduce operations across the parallel system and therefore they benefit directly from efficient implementations of such operations. Our approach is based on the semi-normal equations, which are in general not backward stable. However, the method is stabilised by using iterative refinement. We show that performing iterative refinement in mixed precision also increases the parallel performance of the algorithm. We consider different variants of the ARPLS algorithm depending on the conditioning of the problem and in this context also evaluate the method of normal equations with iterative refinement. For ill-conditioned systems, we demonstrate that the semi-normal equations with standard iterative refinement achieve the best accuracy compared to other parallel solvers.

We discuss the conceptual advantages of ARPLS-IR and ARPLS-MPIR over alternative parallel approaches based on QR factorisation or the normal equations. Moreover, we analytically compare the communication cost to an approach based on communication-avoiding QR factorisation. Numerical experiments on a high performance cluster illustrate speed-ups up to 3820 on 2048 cores for ill-conditioned tall and skinny matrices over state-of-the-art solvers from DPLASMA or ScaLAPACK.

Keywords: parallel least squares solver, semi-normal equations, normal equations, iterative refinement, mixed precision, tall and skinny matrices, all-reduce

1. Introduction

In scientific applications, a typical problem is fitting the parameters of a mathematical model to observations which are subject to errors. Performing linear regression analysis on these observations requires efficient linear least squares (LLS) solvers. We consider the problem of solving the dense LLS problem

$$\min_x \|b - Ax\|_2 \tag{1}$$

in parallel, where $A \in \mathbb{R}^{n\times m}$ with $n \geq m$ and $b \in \mathbb{R}^n$. Of special interest are strongly overdetermined LLS problems where the matrix A has many more rows than columns ($n \gg m$), also known as *tall and skinny* LLS problems. Many big data applications naturally exhibit such a strongly rectangular structure, having billions of data points with only a few hundred descriptors. For example, monitoring seismological activity generates massive amount of data. In [1], a wireless sensor network with only three nodes was deployed around a volcano and returned millions of rows of sensed data. Tall and skinny problems also arise in partial differential equations [2]. Another field of application where high dimensional regression is required is genetics [3]. Single nucleotide polymorphisms (SNPs) can help exhibit a human's susceptibility

Email addresses: karl.prikopa@univie.ac.at (Karl E. Prikopa), wilfried.gansterer@univie.ac.at (Wilfried N. Gansterer), elias.wimmer@univie.ac.at (Elias Wimmer)

The definitive version was published in Parallel Computing Volume 57, September 2016
http://dx.doi.org/10.1016/j.parco.2016.05.014

to different diseases. Millions of SNPs are known today, but the number of subjects for a study of a certain disease is often very low, often limited to a few thousand due to high costs.

We present and evaluate the novel *all-reduce parallel least squares* solvers ARPLS-IR and ARPLS-MPIR for solving problem (1) which are based on the method of semi-normal (SNE) or normal equations (NE) with (mixed precision) iterative refinement (IR). A and b are distributed row-wise across all N processes and the solution $x \in \mathbb{R}^m$ is replicated across the processes. All internode communication in the ARPLS algorithms is contained in all-to-all reduction operations across the participating processes. We consider different variants of the ARPLS algorithm depending on the conditioning of the problem. We show that the application of *mixed precision* iterative refinement (MPIR) in the context of parallel LLS solvers not only reduces the amount of computation but also the communication costs. To the best of our knowledge, the combination of MPIR with LLS solvers has not been studied so far. The mixed precision SNE approach is limited to systems with a condition number up to $\kappa(A) \approx 10^7$ due to single precision being used throughout the majority of the algorithm. In the case of NE, the mixed precision approach is further limited because the normal equations square the condition number of A. Therefore, mixed precision NE does not work for ill-conditioned systems. However, we demonstrate that the accuracy of the standard precision IR method is comparable to existing methods also for higher condition numbers of A. For ill-conditioned systems, unlike some other methods, the approach using SNE with IR (ARPLS-SNE-IR) can still solve the LLS problem and in all cases returns the highest accuracy among the compared algorithms. Moreover, we provide an analysis and comparison of the communication cost of different parallel LLS solvers. Like many standard parallel solvers for dense LLS problems, many ARPLS variants require a parallel QR factorisation algorithm. We thoroughly compare an all-reduce-based parallel version of modified Gram-Schmidt with Tall Skinny QR [4] in terms of computation and communication cost and show how to optimise the communication cost of the all-reduce-based QR factorisation.

The paper is organised as follows. Section 2 summarises related work on parallel and distributed LLS solvers. Section 3 describes the mathematical basis for our approach. Section 4 introduces and discusses different variants of the ARPLS method and a parallel QR factorisation method based on modified Gram-Schmidt. Section 5 provides an analysis of the communication cost and a comparison with an LLS solver based on the communication-avoiding QR (CAQR) algorithm [4] which achieves the theoretical minimum in terms of communication cost. In Section 6 we summarise numerical experiments conducted on a large scale cluster for comparing the performance of ARPLS methods and the LLS solvers from DPLASMA and ScaLAPACK. Section 7 concludes our paper.

2. Related Work

Many parallel algorithms for solving LLS problems have been studied in the literature and are available in high-performance libraries like ScaLAPACK [5], aimed at distributed memory parallel computers, PLASMA [6], designed for shared-memory multi-core machines, MAGMA [6], considering heterogeneous and hybrid architectures with multi-core and GPU systems, or DPLASMA [7], which extends PLASMA to distributed heterogeneous systems. PLASMA and DPLASMA use specialised dynamic scheduling systems (QUARK and PaRSEC, respectively) based on building a direct acyclic graph of parallel tasks and considering the data dependencies of these tasks.

The basic building block of many LLS solvers is a QR factorisation. In PLASMA this is implemented using the tiled QR factorisation algorithm [8], which divides the matrix into small square tiles instead of using rectangular panels seen in block algorithms. The finer granularity achieved by the square tiles is better suited for multicore architectures [9]. Demmel et al. [4] introduced communication-avoiding QR factorisation (CAQR) and proved that its communication cost is optimal up to polylogarithmic factors. The CAQR algorithm factorises block-columns of A, called panels, in parallel using the TSQR algorithm, which is designed for tall and skinny matrices. The panels are divided into block-rows, called domains, which are factorised independently and then merged using a binary tree strategy. In [10], a systolic QR factorisation algorithm is implemented for a distributed memory machine using the PaRSEC parallel scheduler. The authors target a 3D torus topology and limit the communication of the algorithm to neighbouring nodes, aiming to minimise the amount of communication in the reduction trees.

The definitive version was published in Parallel Computing Volume 57, September 2016
http://dx.doi.org/10.1016/j.parco.2016.05.014

An example for a parallel LLS solver is the parallel multisplitting method by Renaut [11], which uses the well-known fixed-point iteration methods Jacobi, Gauss-Seidel and successive over-relaxation to solve the LLS problem by forming the normal equations. The matrix A is distributed column-wise over the network nodes and weighting matrices are used to recombine the local problems, which are independent problems resulting from the linear multisplitting of A. In each iteration a vector of size n has to be broadcast to all other nodes in the network.

A slightly different emphasis is pursued in *distributed* LLS solvers, which limit their communication to the immediate neighbourhood and are therefore of particular interest for loosely coupled networks. Sayed et al. [12] have proposed a diffusion-based least mean square estimator (diffLMS) using normal equations and steepest-descent iterations. Diffusion strategies are seen as an alternative to consensus strategies for distributed optimisation problems, both aiming at limiting the communication to the neighbourhood. The data A and b are both distributed row-wise. The distributed least mean squares method (D-LMS) described in [13] also only uses neighbourhood communication. The method is based on Lagrange multipliers and uses the least squares residual and the difference between the estimates of x from the neighbourhood in a correction step to compute the LLS solution iteratively. The data distribution of A and b is again row-wise. At each step an estimate for the solution x is available in each node. D-LMS communicates twice in each iteration step, once to broadcast the current estimate to the neighbourhood and a second time to send individual correction vectors to each node in the neighbourhood (one-hop unicast). However, first observations indicate that D-LMS has higher communication cost to reach an accuracy comparable to the ARPLS algorithms presented in this paper (for details, see [14]).

3. Iterative Refinement for Linear Least Squares Problems

Mathematically, our approach for solving problem (1) in parallel is either based on the semi-normal equations (SNE) or the normal equations (NE) in combination with iterative refinement (IR).

3.1. Normal Equations

The method of normal equations (NE) solves problem (1) by forming and solving the normal equations:

$$A^\top Ax = A^\top b \tag{2}$$

Assuming A has full rank, the LLS problem has a unique solution. The normal matrix $C := A^\top A \in \mathbb{R}^{m\times m}$ is symmetric and positive definite. Therefore, (2) can be solved using the Cholesky factorisation $C = LL^\top$:

$$LL^\top x = A^\top b$$

The main drawback of NE is that the method is not necessarily backward stable [15]. Forming the cross product squares the condition number: $\kappa(A^\top A) = \kappa(A)^2$. The best forward error bound that can be expected is

$$\frac{\|x^* - x\|}{\|x^*\|} \lesssim nm\kappa(A)^2\varepsilon_{\text{mach}}$$

with x^* being the exact solution and $\varepsilon_{\text{mach}} = \mathit{b}^{1-t}$ being the machine epsilon with b defining the base of the floating-point representation and t the precision. If $\kappa(A) \geq \varepsilon_{\text{mach}}^{-1/2}$, C can be singular or indefinite and the Cholesky factorisation of C will break down. Only if A is well-conditioned, the approach based on the NE is guaranteed to be backward stable.

3.2. Semi-Normal Equations

The method of semi-normal equations (SNE) for solving LLS problems is derived from the normal equations using a QR factorisation $A = QR$:

$$A^\top Ax = A^\top b \quad \Leftrightarrow \quad R^\top Rx = A^\top b.$$

Note that the factor Q is not needed to compute the solution, due to $Q^\top Q = I$. For a large and sparse matrix A, storing and accessing Q is often uneconomical and therefore Q is often discarded. Having the original matrix A, it is still possible to solve multiple right-hand sides b with SNE without Q.

The stability of the SNE method for the LLS problem is analysed extensively in [16]. It has been shown that SNE are not backward stable and that the error in x is similar to the error for the method of normal equations. They have the same numerical properties as the normal equations, even though the factor R from the QR factorisation is of better quality than a Cholesky factorisation of $A^\top A$. The dominating error arises from the rounding errors in the computation of the right hand-side $A^\top b$. However, adding an iterative refinement correction step, as shown in [17] and [18], leads to much more satisfactory results. As long as A does not have widely differing row norms, the SNE with IR become backward stable under certain conditions which will be discussed in Section 3.3.1.

3.3. Iterative Refinement

Iterative refinement (IR) [19] is a strategy for improving the accuracy of a computed solution by trying to reduce round-off errors. The method iteratively computes a correction term to an approximate solution by solving a system using the residual of the result. The correction term is then added to the result to correct the solution. This process is repeated until the accuracy of the solution is sufficiently improved. The cost of the iterative improvement is very low compared to the cost of the factorisation of the matrix but results in a solution which can be accurate to machine precision. Iterative refinement was first analysed in detail by Wilkinson in [19] for linear systems using a scaled fixed point arithmetic and was later expanded by Moler [20] to cover floating-point arithmetic.

A wide range of variations of IR exist which mainly differ in the precisions used for computing the different steps in the process. The terms *target precision* p_α and *working precision* p_β will be used here to distinguish between the targeted precision of the solution and the precision used for the majority of the computation steps, respectively. The standard IR method performs all computations using the same floating-point precision $p_\alpha = p_\beta$. In [21], standard IR has been expanded to use a higher working precision than the target precision to compute the critical steps of IR. This method has been called *extra precise iterative refinement* (EPIR).

Mixed precision iterative refinement (MPIR) [22, 23] is a special performance-oriented case of IR which has been studied for solving linear systems of equations. The majority of operations, the matrix factorisation and solving the linear systems, is computed in single precision (SP) and only the critical parts, computing the residual and updating the solution, are performed in double precision (DP), operations of low complexity compared to the factorisation. This can lead to a result which is accurate to the targeted double precision while the performance has been greatly improved, since the iterative process incurs only very low additional cost. As stated in [23], MPIR using single and double precision achieves at least the same and often higher accuracy than a double precision direct solver as long as the matrix is not too badly conditioned. Some very ill-conditioned systems may never converge and others may need a high number of iterations until they converge to the correct solution. The number of iterations required for convergence directly relates to the condition number of the system matrix, as can be seen in Section 3.4.

Using the lower working precision has many benefits. Modern processors support vector instruction sets, which enables the processor to compute multiple double precision operations in one clock cycle. With SP, the processor can perform double as many operations in one cycle, which significantly increases the performance. SP data also uses less storage, which results in a lower number of cache misses. Furthermore, moving SP data through the memory is faster due to the lower storage requirements. MPIR computes the entire solution of the system with increased performance as long as the performance of SP and DP is significantly different on the used hardware [24].

In [25], it was shown that IR can be used to stabilise almost any linear equation solver and that there is no need to compute the residual in higher precision to stabilise the method but using the same floating-point precision as the solver is sufficient. Numerical stability is proved for $\kappa(A)\varepsilon_{\text{mach}} \leq 1$, where $\kappa(A)$ is the condition number of A and $\varepsilon_{\text{mach}}$ is the machine epsilon.

3.3.1. Corrected Semi-Normal Equations

For LLS problems, the iterative refinement approach [17], which is also referred to in the literature as *corrected SNE* (CSNE), is defined by

$$\|b - Ax\|_2 = \|r - A\Delta x\|_2$$

where $x = \hat{x} + \Delta x$ with $\hat{x}$ being the initial solution, Δx the correction term and $r = b - A\hat{x}$ is the residual vector of the LLS problem. The least squares residual satisfies $A^\top r = 0$ [26]. The correction term Δx is itself the solution to a linear least squares problem. The steps of IR for the SNE method are as follows:

1. Factorise $A = QR$
2. Solve $R^\top R\hat{x} = A^\top b$ for $\hat{x}$
3. $x_0 := \hat{x}$
4. For $i = 0, 1, 2, \ldots$
   (a) Compute $r_i = b - Ax_i$
   (b) Solve $R^\top R\Delta x_i = A^\top r_i$ for Δx_i
   (c) Update $x_{i+1} = x_i + \Delta x_i$

In Step 1, a QR factorisation of A is computed and then used in Step 2 to compute an initial solution $\hat{x}$. Subsequently the iterative refinement algorithm tries to increase the accuracy of the solution by computing the residual r_i of the result in Step 4a and using $A^\top r_i$ as the right-hand side to solve the system for the correction term Δx_i in Step 4b using the already available factor R. Finally, the correction term is added to the result to improve the solution of the LLS problem in Step 4c. This process is repeated until the accuracy of the solution is satisfactory. The rate of convergence is shown in [18] to be roughly linearly dependant on the condition number $\kappa(A)$.

Summarising the analysis in [16], assuming that R is non-singular, then the bound of the estimate of the absolute error for SNE with a single step of IR is

$$\|x^* - x_i\|_2 \leq \sigma\kappa\varepsilon_{\text{mach}} \left(c_2 \|x^*\|_2 + m^{1/2} n \frac{\|b\|_2}{\|A\|_2} \right) + m^{1/2}\kappa\varepsilon_{\text{mach}} \left(m \|x^*\|_2 + n\kappa \frac{\|r\|_2}{\|A\|_2} \right) + m^{1/2}\varepsilon_{\text{mach}} \|x\|_2$$

with $\kappa := \kappa(A)$, x^* being the exact solution and x_i the solution after the i^{th} refinement step. In this bound, $c_2 = 2m^{1/2}(c_1 + m)$ and $c_1 = c_1(n, m)$ is a polynomial in n and m and depends on the method used to compute the QR factorisation. Moreover,

$$\sigma = c_3\kappa^2\varepsilon_{\text{mach}}, \text{ with } c_3 \leq 2m^{1/2} \left(c_1 + 2m + \frac{n}{2} \right) \quad .$$

The combination of SNE with IR is not in general backward stable, but for $\sigma < 1$ it is more accurate than the QR method (a QR factorisation followed by triangular solve) and less accurate if $\sigma > 1$.

For multiple iterative refinement steps, the error bound of a single step is given in [16] by

$$\|x^* - x_i\|_2 \leq m^{3/2}\kappa\varepsilon_{\text{mach}} \left(\|x^*\|_2 + \frac{n}{m}\kappa \frac{\|r\|_2}{\|A\|_2} \right) [1 + O(\kappa\varepsilon_{\text{mach}})]$$

As long as $O(\kappa\varepsilon_{\text{mach}})$ is negligible compared to 1, this error estimate also holds for further steps of IR. If the refinement converges initially, the limiting accuracy does not depend strongly on the starting vector x_0. Therefore, SNE with IR is backward stable if

$$2c_1 m^{1/2}\kappa\varepsilon_{\text{mach}} < 1 \quad .$$

In [16], the author also provides an example for the very ill-conditioned Hilbert matrices, where SNE with IR still produces useful results whereas normal equations (without IR) fail completely and SNE without IR is unstable. The results from SNE with IR are shown to be comparable to the QR method. In Section 6.3, we experimentally investigate the numerical stability of SNE with IR.

3.3.2. Other Iterative Refinement Approaches for LLS Problems

The method of normal equations (NE) can also be used instead of a QR factorisation. Step 1 is then replaced by forming the normal equations $C := A^\top A$ followed by a Cholesky factorisation $C = LL^\top$. To solve the systems in Steps 2 and 4b, the factor L is used to compute $LL^\top y = A^\top b$, where y is either $\hat{x}$ or Δx_i, respectively. As stated in [15], for NE the rate of convergence depends on $\kappa(A)^2$ instead of $\kappa(A)$.

In [27], an extra precise IR (EPIR) method for LLS problems is proposed, where the critical parts of computing the residual and updating the solution are performed in extended precision, which refers to using double-double precision with a 106 bit significand for intermediate results. This leads to a reduction of the forward norm-wise and component-wise errors to $O(\varepsilon_{\text{mach}})$ for the solution x and the residual r. The LLS problem can also be formulated as an augmented linear system of dimension $n + m$, as shown in [18]:

$$\begin{pmatrix} I_n & A \\ A^\top & 0 \end{pmatrix} \begin{pmatrix} r \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix} \tag{3}$$

According to [27], the analysis for EPIR for linear systems [21] can be applied directly to the augmented system (3) for the LLS problem.

To the best of our knowledge, the application of *mixed precision* IR in the context of LLS solvers has not been studied so far. However, formulating the LLS problem as an augmented linear system (3) makes all IR methods developed for linear systems applicable to LLS problems. Therefore the same proofs of convergence and numerical improvement can be applied to MPIR for LLS problems.

3.4. Number of Iterations

In [28], the number of iterations required by iterative refinement is used to estimate the condition number of a linear system $Ax = b$. For IR and MPIR with standard IEEE precisions, the number of iterations for well-conditioned and mildly ill-conditioned problems is in general very low (mostly around two or three). The logarithm to base β of the condition number $\kappa(A)$ returns an estimate of the number of base-β digits that are lost while solving the system. Through simple transformations one arrives at the following estimate for $\kappa(A)$ based on the number of iterations k required by IR for a precision p and gaining s digits of accuracy in each iteration:

$$\kappa(A) \approx \beta^{p-s} = \beta^{p-p/k}$$

From this relationship, it is possible to estimate the number of iterations of IR based on the knowledge of $\kappa(A)$. By setting the precision in the numerator to the target precision and the precision in the denominator to the working precision, the model results in

$$k \approx \frac{p_\alpha}{p_\beta - log_2(\kappa(A))} \tag{4}$$

where p_α and p_β are specified in number of bits stored in the mantissa of a floating-point number. For standard IEEE single and double precision, this would be 24 and 53, respectively.

4. All-Reduce Parallel Linear Least Squares Solvers – ARPLS

In this section, we discuss a parallelisation strategy for an LLS solver based on SNE or NE and IR (as summarised in Section 3). All communication of the resulting algorithms is contained in all-reduce operations of the participating processes, and we therefore call our algorithms *all-reduce parallel least squares* (*ARPLS*) solver. These algorithms comprise three main components: (*i*) a parallel QR factorisation or a parallel matrix multiplication, (*ii*) a parallel matrix-vector multiplication followed by one (without SNE or NE) or two (with SNE or NE) local triangular solves to compute the solution to the LS problem, and (*iii*) IR to stabilise and improve the solution computed in the previous step (this will only be applied to algorithms based on SNE or NE), also requiring a parallel matrix-vector multiplication.

4.1. Parallel QR Factorisation

The first component required by most variants of the ARPLS algorithm is a parallel QR factorisation. In this paper, we will consider two parallel methods, a variation of the distributed modified Gram-Schmidt orthogonalisation (dmGS) [29] and the Tall Skinny QR (TSQR) [4].

4.1.1. All-reduce Modified Gram-Schmidt

A distributed modified Gram-Schmidt orthogonalisation (dmGS) was presented in [29] using a gossip-based reduction algorithm. The parallel variant, the *all-reduce modified Gram-Schmidt* (armGS) algorithm, is outlined in Algorithm 1. armGS assumes that the matrix A is distributed row-wise across the processes. The part of A available locally at process u is therefore denoted by $A^{(u)}$. armGS returns the factor Q distributed row-wise (the same distribution as A) and the upper-triangular factor R which is fully available on every process. The armGS algorithm only differs from sequential mGS in the parallel computation of two sums using two reduction operations. The first one, `arsum` in line 3, is a reduction of the local sums to compute the 2-norm of column j and the second one is a parallel matrix-vector multiplication `argemv` of the transpose of column j of Q with A. `argemv` first computes the product using the locally available factors and then forms the sum of the local results using a parallel reduction operation. No additional communication is necessary and the rest of the computations are performed locally.

Algorithm 1 All-reduce Modified Gram-Schmidt (armGS)

Input: $A \in \mathbb{R}^{n \times m}$ with $n > m$ distributed row-wise over N processes
Output: $Q \in \mathbb{R}^{n \times m}$ distributed row-wise over N processes, $R \in \mathbb{R}^{m \times m}$ on every process

1: **in each** process u **do**
2: **for each** column $j = 1..m$ **do**
3: $v \leftarrow \texttt{arsum}(\langle {A^{(u)}}^\top(:,j), A^{(u)}(:,j)\rangle)$
4: $R(j,j) \leftarrow \sqrt{v}$ ▷ norm of column j
5: $Q^{(u)}(:,j) \leftarrow A^{(u)}(:,j)/R(j,j)$
6: $R(j, j+1:m) \leftarrow \texttt{argemv}({Q^{(u)}}^\top(:,j), A^{(u)}(:, j+1:m))$
7: $A^{(u)}(:, j+1:m) \leftarrow A^{(u)}(:, j+1:m) + Q^{(u)}(:,j)R^\top(j, j+1:m)$ ▷ rank-1 update on $A^{(u)}$
8: **end for**

For the SNE, the factor Q is not required. The Q-less mGS approach (denoted as *armGSR* in the following) therefore only returns the full factor R of the QR factorisation and discards the computed columns of Q. This reduces the memory requirements compared to the armGS algorithm by $n(m-1)$ scalars as only one vector of Q of length n is needed for the computation.

Both methods can be further improved in terms of communication cost by postponing the scaling of the column of A by the diagonal element of R after the computation of the second parallel summation in line 6. The first summation of a scalar in line 3 can be combined with the second parallel reduction operation by appending a single value to the vector. This reduces the communication cost from $2m-1$ to m messages and eliminates the overhead caused by the communication of a scalar value. In the following, we will refer to this method as armGSR-Opt.

4.1.2. Tall Skinny QR (TSQR)

The parallel Tall Skinny QR (TSQR) method [4] is aimed at narrow matrices with $n \gg m$ which are distributed row-wise over all processes. As mentioned before, in the context of semi-normal equations, only the factor R is required. Therefore we only outline the algorithm for this case and do not consider the computation of Q. However, the Q factor is implicitly represented by the intermediate parts of Q computed along the reduction path and can be reconstructed if needed.

TSQR first computes a local QR factorisation of the locally available rows and then performs a reduction operation of the local R factors (denoted by $R^{(u)}$ at process u) to compute the full R factor of A. For example, if the input matrix A is split into two block-rows $A^{(0)}$ and $A^{(1)}$, the local triangular factors $R^{(u)}$ would be

$R^{(0)} = \mathrm{qr}(A^{(0)})$ and $R^{(1)} = \mathrm{qr}(A^{(1)})$. Performing a QR factorisation of the local factors $R^{(u)}$ stacked on top of one another leads to the R factor of the entire matrix A:

$$R = \mathrm{qr}\begin{pmatrix} R^{(0)} \\ R^{(1)} \end{pmatrix}$$

For more than two processes, this approach can be applied recursively to all $R^{(u)}$ to compute the QR factorisation in parallel along a reduction tree. The method is associative and can also be made commutative by ensuring that the diagonal of each computed R factor only contains positive entries and is therefore unique (provided A is non-singular). This can be achieved by multiplying the rows of R having a negative diagonal element with -1. The algorithm is outlined in Algorithm 2.

Algorithm 2 Tall Skinny QR (TSQR)

Input: $A \in \mathbb{R}^{n \times m}$ with $n \gg m$ distributed row-wise over N processes
Output: $R \in \mathbb{R}^{m \times m}$ on every process
1: **in each** process u **do**
2: $R^{(u)} \leftarrow \mathrm{qr}(A^{(u)})$ ▷ local QR factorisation
3: $R \leftarrow$ all-reduce$(R^{(u)})$

TSQR, like armGS, can be implemented using only all-reduce operations, which also replicates the final R factor over all processes. In MPI, the collective operations can be used directly with a user-defined reduction operation, benefiting from the reduction trees available in the MPI implementations, which are usually optimised for the targeted architecture.

TSQR has been shown to be communication optimal [30], only requiring $O(\log N)$ messages for the reduction operation of R. However, even though the algorithm is communication optimal, in each step of the reduction tree, a QR factorisation has to be computed, which can have a significant performance impact (depending on m). The number of flops is $O(m^3 \log(N))$ along the critical path [31]. The performance can be improved by exploiting the triangular structure of the stacked matrices and using a custom local QR factorisation. In [30], the recursive approach of Elmroth and Gustavson [32] has been used to achieve a reduction of the number of flops by a factor of 5 compared to a standard QR factorisation with $\frac{10}{3}m^3$ flops.

4.2. Parallel LLS Solver

One of the standard methods for solving the LLS problem (1) is the use of the QR factorisation to solve the system $Rx = Q^\top b$. The ARPLS-QR algorithm (shown in Algorithm 3) computes the QR factorisation of A in parallel, either using armGS or TSQR, followed by a parallel matrix-vector multiplication `argemv` of ${Q^{(u)}}^\top$ and $b^{(u)}$ in line 3 in Algorithm 3. The final step in ARPLS-QR is the local back substitution using the locally available factor R and the result z of `argemv`. Each process then holds the solution x.

Algorithm 3 All-Reduce Parallel Least Squares Solver based on QR Factorisation (ARPLS-QR)

Input: $A \in \mathbb{R}^{n \times m}$ with $n > m$, $b \in \mathbb{R}^n$, both distributed row-wise over N processes
Output: $x \in \mathbb{R}^m$ on every process
1: **in each** process u **do**
2: $[Q^{(u)}, R] \leftarrow \mathtt{qr}(A^{(u)})$ ▷ parallel QR factorisation (armGS or TSQR)
3: $z \leftarrow \mathtt{argemv}({Q^{(u)}}^\top, b^{(u)})$
4: $x \leftarrow$ solve $Rx = z$ ▷ local linear system solve

For ARPLS-SNE (see Algorithm 4), whose mathematical basis has been reviewed in Section 3, the process is identical in terms of communication. The Q-less mGS method, armGSR, requires the same amount of computation and communication as armGS, but only returns the factor R. `argemv` is used to compute

$A^{(u)\top} b^{(u)}$ in parallel, with $A^{(u)\top}$ having the same row-wise distribution as $Q^{(u)\top}$ in ARPLS-QR. Finally, the system is solved using a forward and back substitution using R and $R^\top$.

Algorithm 4 ARPLS based on Semi-Normal Equations (ARPLS-SNE)

Input: $A \in \mathbb{R}^{n \times m}$ with $n > m$, $b \in \mathbb{R}^n$, both distributed row-wise over N processes
Output: $x \in \mathbb{R}^m$ on every process
1: **in each** process u **do**
2: $R \leftarrow \texttt{qr}(A^{(u)})$ ▷ parallel QR factorisation (armGSR or TSQR)
3: $z \leftarrow \texttt{argemv}(A^{(u)\top}, b^{(u)})$
4: $x \leftarrow$ solve $R^\top R x = z$ ▷ local linear system solve

As described in Section 3.1, an alternative to solving the LLS problem using the QR factorisation is the computation of the normal equations (NE). ARPLS-NE (shown in Algorithm 5) first computes $C = A^\top A$ in parallel, which only requires a single reduction operation to compute the sum of $O(m^2)$ local values. Therefore, this approach is as communication optimal as TSQR, only requiring $O(\log N)$ messages for the reduction operation of C but performing a much simpler operation than TSQR along the critical path. The following Cholesky factorisation of C is performed locally. The subsequent steps are the same as in ARPLS-SNE. `argemv` is used to compute $A^{(u)\top} b^{(u)}$ in parallel and the system is solved using a local forward and back substitution using the local factor L.

Algorithm 5 ARPLS based on Normal Equations and Cholesky Factorisation (ARPLS-NE)

Input: $A \in \mathbb{R}^{n \times m}$ with $n > m$, $b \in \mathbb{R}^n$, both distributed row-wise over N processes
Output: $x \in \mathbb{R}^m$ on every process
1: **in each** process u **do**
2: $C \leftarrow \texttt{arsum}(A^{(u)\top} \cdot A^{(u)})$ ▷ parallel computation of the normal equations
3: $L \leftarrow \texttt{cholesky}(C)$ ▷ local Cholesky factorisation
4: $z \leftarrow \texttt{argemv}(A^{(u)\top}, b^{(u)})$
5: $x \leftarrow$ solve $L L^\top x = z$ ▷ local linear system solve

4.3. Iterative Refinement

We denote our ARPLS solvers using IR as ARPLS-IR and ARPLS-MPIR (see Algorithm 6). They first compute an initial solution by using ARPLS-SNE or ARPLS-NE and then improve the solution by IR. They compute the residual locally (line 5 in Algorithm 6) and then in parallel apply $A^{(u)\top}$ using `argemv`. Subsequently, a correction term Δx is computed by solving the system using the already computed factor R. Finally, the approximate solution is updated by the correction term. The process continues until a convergence criterion is met.

ARPLS-IR uses the same precision throughout the process ($p_\alpha = p_\beta$). The mixed precision approach ARPLS-MPIR computes the initial solution completely in the lower working precision p_β. Computing the residual and applying $A^{(u)\top}$ to r (lines 5 and 6 in Algorithm 6) have to be computed in p_α. The correction term is computed in the lower precision using the factor R or L, which is only available in p_β. The final step, updating the solution, is again performed in the higher target precision p_α. The majority of the computations, i. e. the factorisation or forming the normal equations, and of the communication are both performed in the lower working precision, leading to improved performance for the local computations and smaller message sizes during the communication.

Algorithm 6 Mixed Precision ARPLS-IR (ARPLS-MPIR)

Input: $A \in \mathbb{R}^{n \times m}$ with $n > m$, $b \in \mathbb{R}^n$, both distributed row-wise over N processes
Output: $x \in \mathbb{R}^m$ on every process

1: **in each** process u **do**
2: $[R, x] \leftarrow$ ARPLS-SNE$(A^{(u)}, b^{(u)})$ or $[L, x] \leftarrow$ ARPLS-NE$(A^{(u)}, b^{(u)})$ ▷ parallel and local, p_β
3: $i = 0$
4: **while** $i <$ maxiter **do**
5: $r \leftarrow b^{(u)} - A^{(u)}x$ ▷ local, p_α
6: $s \leftarrow \texttt{argemv}({A^{(u)}}^\top, r)$ ▷ p_α
7: **if** $\|s\| <$ tolerance **then**
8: **break** → converged
9: **end if**
10: $\Delta x \leftarrow$ solve $R^\top R \Delta x = s$ (ARPLS-SNE) or solve $LL^\top \Delta x = s$ (ARPLS-NE) ▷ local, p_β
11: $x \leftarrow x + \Delta x$ ▷ local, p_α
12: $i = i + 1$
13: **end while**

4.4. Combining Iterative Refinement with Other LLS Solvers

One could consider the use of iterative refinement (IR or MPIR) with other LLS solvers. Using the QR factorisation to solve the initial LLS system with the factor Q does not make any difference in the amount of communication compared to SNE and NE. All these approaches require a parallel matrix-vector product for forming $z = {Q^{(u)}}^\top b^{(u)}$ (line 3 in Algorithm 3) or $z = {A^{(u)}}^\top b^{(u)}$ (line 3 in Algorithm 4 and line 4 in Algorithm 5). The local linear system solve is more expensive for ARPLS-IR since it requires a backward *and* forward substitution instead of only one backward substitution in ARPLS-QR. However, those operations are only of order m^2 and therefore have a very low impact on the overall computation time, which is dominated by the QR factorisation or by forming the normal equations, both being of order $O(nm^2)$.

The main advantage of using SNE or NE appears in the iterative refinement. To compute $s = A^\top(b - Ax) = A^\top r$ (line 6 in Algorithm 6), a parallel matrix-vector operation is needed in each iteration of IR. This parallel computation is required by all variants of the LLS solver but at different steps of the algorithm. For SNE or NE, s is required to compute the correction term and at the same time provides the accuracy of the current result to determine if the method has already converged. Not using SNE or NE, this step is only required to check for convergence after the computation of the correction term. To solve the system for Δx, ARPLS-QR would multiply r by $Q^\top$, which would incur an additional parallel matrix-vector operation. This step is not required by the SNE or NE approach in ARPLS-IR because all information needed to solve the system (the R factor from the QR factorisation or the L factor from the Cholesky factorisation) is already available locally and no further communication is necessary. Combining Algorithm 3 with IR would also require more memory for storing $Q \in \mathbb{R}^{n \times m}$ than SNE or NE with IR.

Another disadvantage arises when mixed precision IR is being considered for ARPLS-QR (Algorithm 3). When applying $Q^\top$ to r, the factor Q has to be available in the higher target precision p_α. However, to exploit the potential performance benefits of MPIR, the initial QR factorisation has to be computed in the lower working precision p_β. In this case, the factor Q is not available in p_α and without the SNE or NE, MPIR will not be able to improve the result beyond the lower precision p_β.

Overall, we can conclude that combining SNE or NE with IR leads to the best results in terms of communication and computation compared to other dense LLS solvers.

4.5. Extensions of ARPLS-IR and ARPLS-MPIR

All communication in our ARPLS solvers is concentrated on reduction operations. Through the use of fault tolerant reduction operations, the algorithms have the potential to become fault tolerant against silent communication failures, such as message loss. Fault tolerant reduction operations include gossip-based,

randomised algorithms like push-flow [33], which limit their communication to the immediate neighbourhood of a computing node. This approach is well suited for extreme scale systems and loosely coupled systems (e. g., wireless sensor networks).

In order for parallel iterative refinement to work, it is important that all nodes use the same x to compute the residual in the first step of each IR iteration (line 5 in Algorithm 6). In the case of `MPI_Allreduce`, this is already guaranteed by the MPI standard which requires all processes to receive identical results [34]. For other types of reduction algorithms, for example, gossip-based algorithms, which compute the sum iteratively and therefore produce different approximations of the sum at each node, it is necessary to ensure that each node u has the same approximation x_u, at least to the accuracy targeted for the solution. Otherwise the computation of the correction term Δx will fail. This can either be achieved by averaging the approximate solution vectors over all nodes, accurate to the targeted accuracy, or by selecting one node to distribute its result to all other nodes in the network. For ARPLS-MPIR, if x_u has to be averaged over all nodes, this operation has to be computed in the higher target precision $p_\alpha = DP$.

5. Analysis of Communication Cost

In this section, we first analyse the communication cost for the different variants of the ARPLS method (see Table 1) and then compare the costs to a parallel LLS solver based on CAQR. Denoting N as the number of processes, a single reduction operation requires about $2 \log N$ messages [35]. For reasons of simplicity, we assume that the number of messages is independent of the size of the data being reduced.

armGS or armGSR can be used by the ARPLS variants having to compute a QR factorisation. Both methods require $2m-1$ sum reduction operations and send the same amount of data $\frac{m(m+1)-2}{2}+2m = O(m^2)$ per process. In armGSR-Opt the number of reduction operations is further reduced to m, as described in Section 4.1.1. We will append *OGSR* to the name of the methods using this optimisation. Solving the LLS problem requires one additional reduction operation for the matrix-vector product, resulting in a total of $2m$ reduction operations for armGS and armGSR or $m+1$ reduction operations for armGSR-Opt.

The TSQR method is communication optimal, as shown in [4], requiring only one reduction operation and therefore being of order $O(\log N)$. It has to be noted, that the reduction operation is not a simple summation, but a more complex and computationally intensive task of a QR factorisation of two $R^{(u)}$ factors at every step of the reduction. Depending on the width of the matrix m, this can have a significant impact on the performance of the algorithm compared to simple sum reduction operations. TSQR reduces the amount of communication by increasing the number of flops by an additional $O(m^3 \log(N))$ along the critical path [31]. In terms of amount of data, each process sends an upper triangular matrix of size $m \times m$ per process, leading to $m(m+1)/2$ values.

Forming the normal equations in parallel is also communication optimal, again only requiring one reduction operation and therefore also being of order $O(\log N)$. However, in contrast to TSQR, the reduction operation is much simpler and only has to compute the sum of symmetric matrices. Therefore only $m(m+1)/2$ elements have to be sent per process.

Adding iterative refinement slightly increases the communication cost due to the additional sum reduction operation for each iteration in line 6 of Algorithm 6. Compared to an armGS QR factorisation, this increase is negligible, since the number of iterations k is very small for well-conditioned matrices (usually, 2-3 iterations suffice). Each reduction operation sums vectors of length m. MPIR requires the same number of reduction operations as IR, but sends less data and therefore smaller messages for the bulk of the communication performed in the QR factorisation or in the computation of the normal equations because of its use of single precision. This halves the amount of data sent per process in all parallel QR factorisation methods and in the summation of the local symmetric matrices to compute the NE.

The main part of the communication cost originates in the parallel QR factorisation, especially if the modified Gram-Schmidt method is used. The TSQR method, which is only intended for tall and skinny matrices, has been shown to be optimal with $2 \log N$ messages [4]. Therefore, the communication cost of the communication-avoiding QR (CAQR) algorithm, which has been shown to be optimal up to polylogarithmic factors, is a reasonable lower bound for the communication cost of a general parallel LLS solver. CAQR

Table 1: Theoretical communication cost per process for the different ARPLS methods. m denotes the number of columns in A and k the number of iterations required by iterative refinement.

ARPLS method	**Factorisation**	**Number of reduction operations**	**Total amount of data sent per process**
QR (GS)	armGS	$2m$	$\frac{m(m+1)-2}{2} + 2m$
SNE (GSR)	armGSR	$2m$	$\frac{m(m+1)-2}{2} + 2m$
SNE-IR (GSR)	armGSR	$2m + k$	$\frac{m(m+1)-2}{2} + m(2 + k)$
SNE-MPIR (GSR)	armGSR	$2m + k$	$\frac{m(m+1)-2}{4} + m(1 + k)$
SNE (OGSR)	armGSR-Opt	$m + 1$	$\frac{m(m+1)-2}{2} + m$
SNE-IR (OGSR)	armGSR-Opt	$m + 1 + k$	$\frac{m(m+1)-2}{2} + m(1 + k)$
SNE-MPIR (OGSR)	armGSR-Opt	$m + 1 + k$	$\frac{m(m+1)-2}{4} + m(\frac{1}{2} + k)$
SNE (TSQR)	TSQR	2	$\frac{m(m+1)}{2} + m$
SNE-IR (TSQR)	TSQR	$2 + k$	$\frac{m(m+1)}{2} + m(1 + k)$
SNE-MPIR (TSQR)	TSQR	$2 + k$	$\frac{m(m+1)}{4} + m(\frac{1}{2} + k)$
NE	Cholesky	2	$\frac{m(m+1)}{2} + m$
NE-IR	Cholesky	$2 + k$	$\frac{m(m+1)}{2} + m(1 + k)$
NE-MPIR	Cholesky	$2 + k$	$\frac{m(m+1)}{4} + m(\frac{1}{2} + k)$

sends

$$msg_{\mathrm{CAQR}}(n, m, N) \;=\; \frac{1}{4}\sqrt{\frac{mN}{n}}\log^2\frac{nN}{m}\log\left(N\sqrt{\frac{nN}{m}}\right) \tag{5}$$

messages and

$$data_{\mathrm{CAQR}}(n, m, N) \;=\; \sqrt{\frac{nm^3}{N}}\log N - \frac{1}{4}\sqrt{\frac{m^5}{nN}}\log\frac{mN}{n} \tag{6}$$

data per process. For the special case of almost square matrices $n \approx m$, this simplifies to

$$msg_{\mathrm{CAQR}}(n, n, N) = O\left(\sqrt{N}\log^3 N\right) \quad \text{and} \quad data_{\mathrm{CAQR}}(n, n, N) = O\left(\frac{1}{\sqrt{N}}\log(N)\right)$$

Comparing this to the number of messages required by the parallel mGS QR factorisation

$$msg_{\mathrm{armGS}}(n, n, N) \;=\; O\left(n \log N\right)$$

reveals that armGS requires a factor $n/\left(\sqrt{N}\log^2(N)\right)$ messages more than CAQR. Considering the amount of data, armGS requires

$$data_{\mathrm{armGS}}(n, n, N) \;=\; O(n^2 \log N)$$

which is $\sqrt{N}$ more than the communication optimal CAQR method.

The communication cost of CAQR in (5) and (6) assumes that n and m are sufficiently large in comparison with the block size. For the case $n \gg m$, CAQR is reduced to performing TSQR on a single panel and therefore only requires $2 \log N$ messages. For armGS, the number of messages depends on m and for the optimised version of armGSR requires m reduction operations, which results in $2m \log N$ messages. Compared to TSQR, armGSR sends $m - 1$ more messages. The amount of data sent per message is lower for armGSR, which only sends vectors with a length of up to m scalars per reduction operation. TSQR has to send a triangular matrix which has $m(m + 1)/2$ values. However, the total amount of data sent by both methods is identical.

The communication cost is not the only factor that has to be considered. A low number of messages will not guarantee a low runtime, especially if the computation during the reduction operation costs more than

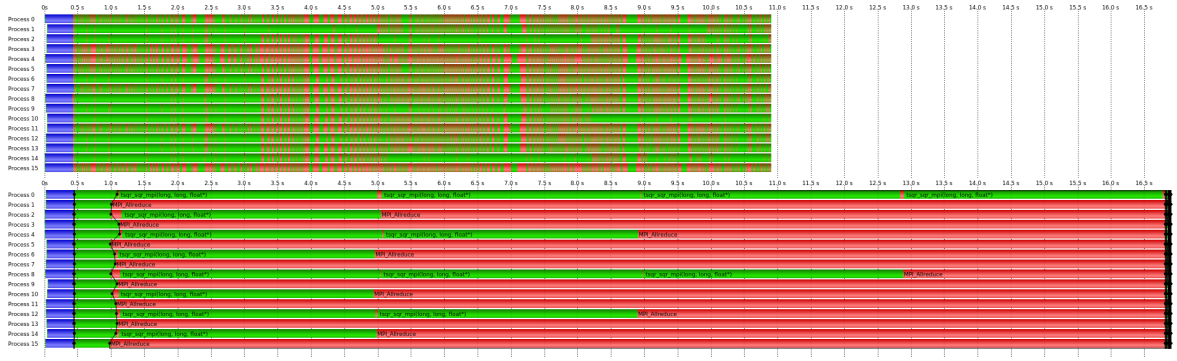


Figure 1: Parallel execution of armGS (top) and TSQR (bottom) on the same time scale for a wider matrix with $n = 16384$ and $m = 2048$. The green blocks represent the computation time, whereas the red blocks show the communication time.

the communication itself. The modified Gram-Schmidt orthogonalisation armGSR requires $\frac{2nm^2}{N}$ flops to compute the QR factorisation. During this operation, m reductions are executed computing a sum of up to m elements, an operation costing $m \log N$ flops, resulting in the total computation costs for the reduction of $O(m^2 \log N)$. The TSQR algorithm also requires $\frac{2nm^2}{N}$ flops to compute the initial QR factorisation of the locally available data. The reduction operation then computes QR factorisations at every reduction step, leading to a computation cost of $O(m^3 \log N)$. The computation costs of this reduction is one order of magnitude higher than the total computation costs for the reduction in armGSR. CAQR uses TSQR for the panel factorisations in CAQR. Therefore, CAQR also computes a local QR factorisation with $\frac{2nm^2}{N}$ flops and additionally uses $O(m^3 \log N)$ operations to compute the solution of the QR factorisation, again a factor m more operations than armGSR.

An analysis of the communication and computation costs is given in Figure 1, which shows a trace of armGS and TSQR using the VampirTrace library [36]. The green fields represent computational tasks and the red fields display the MPI communication. To illustrate the effect, a wider matrix A was used with $n = 16384$ and $m = 2048$. Both methods are displayed on the same time scale and in this example armGS is about 30% faster. Naturally, for wide matrices one would apply TSQR to panels of A (using CAQR) and achieve a much better performance. Furthermore, TSQR also has the advantage of exploiting the performance of level 3 BLAS operations, whereas armGS is limited to level 2 BLAS. However, this toy example is intended to demonstrate the influence of the synchronisation on the communication time. TSQR only requires a single `MPI_Allreduce`, but during this operation most processes are idle, waiting for the MPI call to complete before they can continue with the next panel or with the solution of the LLS problem. With every merge of two processes to compute the QR factorisation of a stacked matrix, fewer processes are involved in the computation. At the end, only a single process is computing the final result, which is then distributed to all other processes.

6. Experiments

In this section, we present performance results for the ARPLS solvers and compare them to state-of-the-art parallel dense LLS solvers.

6.1. Experimental Setup

All experiments were run on the Vienna Scientific Cluster VSC-2[1] consisting of 1314 nodes. Each node holds two AMD Opteron 6132HE processors with eight cores each and has 32 GB of main memory. The nodes are connected through Infiniband QDR using a fat tree topology.

[1] http://vsc.ac.at/systems/vsc-2/

The ARPLS variants only use the `MPI_Allreduce` subroutine to perform the parallel summations, which are the only operations which are computed in parallel. For DPLASMA [7] (version 1.2.1), ScaLAPACK [5] and the ARPLS variants the MPI library achieving the best performance on the VSC-2 was chosen. DPLASMA achieved the highest performance using OpenMPI, whereas the best performance of `MPI_Allreduce` and therefore of ARPLS was achieved with MVAPICH2 or Intel MPI. In our setup, the `MPI_Allreduce` subroutine in OpenMPI was on average 100 times slower than the one in the other MPI libraries available on the cluster. As ARPLS strongly depends on an efficient implementation of `MPI_Allreduce`, MVAPICH2 was used for the ARPLS algorithms and OpenMPI for DPLASMA. ScaLAPACK performed best using MVAPICH2.

We compare the performance of our approaches with the routine `pdgels` from ScaLAPACK and with the routine `dplasma_dgels` from DPLASMA, since these two routines represent the state-of-the-art in available high performance implementations of parallel dense LLS solvers. DPLASMA is executed using one process per node, with each node on the VSC-2 having 16 cores available. DPLASMA provides many different parameters to tune its routines for high performance, including various block sizes (tile, supertile and inner blocking), parameters for defining the process grid and the type and size of the high and low-level reduction trees. The high-level trees are specific to the reduction between nodes and the low-level trees take care of the reduction within the nodes. Four types of reduction trees are currently implemented for both levels: a flat tree, a binomial tree, a Fibonacci tree and a greedy tree, which is also the default tree used by DPLASMA. We tested various tile sizes for the different problem sizes and selected the ones delivering the best performance on our test machine. All variants of the reduction trees were also tested using various tree sizes, but a performance increase compared to the default greedy tree was only observed for a single problem size (about 20% performance increase for $n = 2^{22}, m = 16$). In all other cases, any combination of the possible parameters described above had none or a negative effect on the performance. In the following, for DPLASMA we always report the best performance results achieved over many different parameter combinations.

6.2. Generating Test Matrices

For the experiments, due to the dependence of the NE and mixed precision on the conditioning of the input matrices, we require test matrices with specific condition numbers κ to analyse the accuracy of the algorithms. We consider the condition number with respect to the 2-norm, $\kappa(A) = \sigma_{\max}/\sigma_{\min}$, where σ_i are singular values of A. In this section we describe our procedure for generating our test matrices.

The idea of our approach is to modify the singular values of A to receive the desired condition number for the matrix. The algorithm for generating the test matrices is shown in Algorithm 7. The method requires a matrix A and the targeted condition number ϑ as its input and returns a modified matrix $\hat{A}$ where $\kappa(\hat{A}) = \vartheta$. First, a singular value decomposition (SVD) of A is computed on line 1, where Σ holds the singular values σ_i which are sorted in descending order. Depending on the requested ϑ, the singular values have to be modified. We distinguish between the following different cases:

1. The simplest case is $\vartheta = 1$, which can only be reached by setting all singular values to 1 (lines 3-4).
2. If $\kappa(A) > \vartheta$, a pair of singular values is sought for which satisfy $\sigma_i/\sigma_{m-i} \leq \vartheta$, where $i \in [1, m/2]$ (lines 6-10).
   (a) If no pair of singular values matches this criterion, then we fall back to the simplest case of setting all singular values to 1 (lines 11-12). The first singular value σ_1 will then be set to the desired condition number ϑ in line 19.
   (b) Otherwise, the singular values larger than σ_i are set to σ_i and the ones smaller than σ_{m-i} are set to σ_{m-i} (lines 13-15).
3. In the case $\kappa(A) \leq \vartheta$, no specific changes are necessary before the scaling in line 19.

In all cases, the first singular value is then set to the last singular valued scaled by ϑ (line 19). Finally, the new matrix $\hat{A}$ is computed using the factors U and V from the SVD and the modified singular values stored in Σ (line 20), leading to $\kappa(\hat{A}) = \vartheta$.

Algorithm 7 Generating test matrices with prescribed condition number κ

Input: $A \in \mathbb{R}^{n\times m}$ where $n > m$, targeted condition number ϑ
Output: $\hat{A} \in \mathbb{R}^{n\times m}$ where $\kappa(\hat{A}) = \vartheta$

$U, \Sigma, V \leftarrow \texttt{svd}(A)$ ▷ Singular value decomposition of A where $\forall i : \sigma_i \geq \sigma_{i+1}$
$\kappa(A) = \sigma_1/\sigma_m$
if $\vartheta == 1$ **then**
    $\sigma_i = 1 \quad \forall i \in [1, m]$ ▷ Set all singular values (sv) to 1
else
    if $\kappa(A) > \vartheta$ **then**
        $i = 1$
        while $i \leq m/2$ **and** $\sigma_i/\sigma_{m-i} > \vartheta$ **do** ▷ Find a pair of sv with ratio $\leq \vartheta$
            $i = i + 1$
        end while
        if $i > m/2$ **then** ▷ No pair of sv found with ratio $\leq \vartheta$
            $\sigma_i = 1 \quad \forall i \in [1, m]$
        else ▷ Pair of sv found with ratio $\leq \vartheta$
            $\sigma_j = \sigma_i \quad \forall j \in [1, i-1]$
            $\sigma_j = \sigma_{m-i} \quad \forall j \in [m-i+1, m]$
        end if
    end if
end if
$\sigma_1 = \vartheta\sigma_m$
$\hat{A} \leftarrow U\Sigma V^\top$

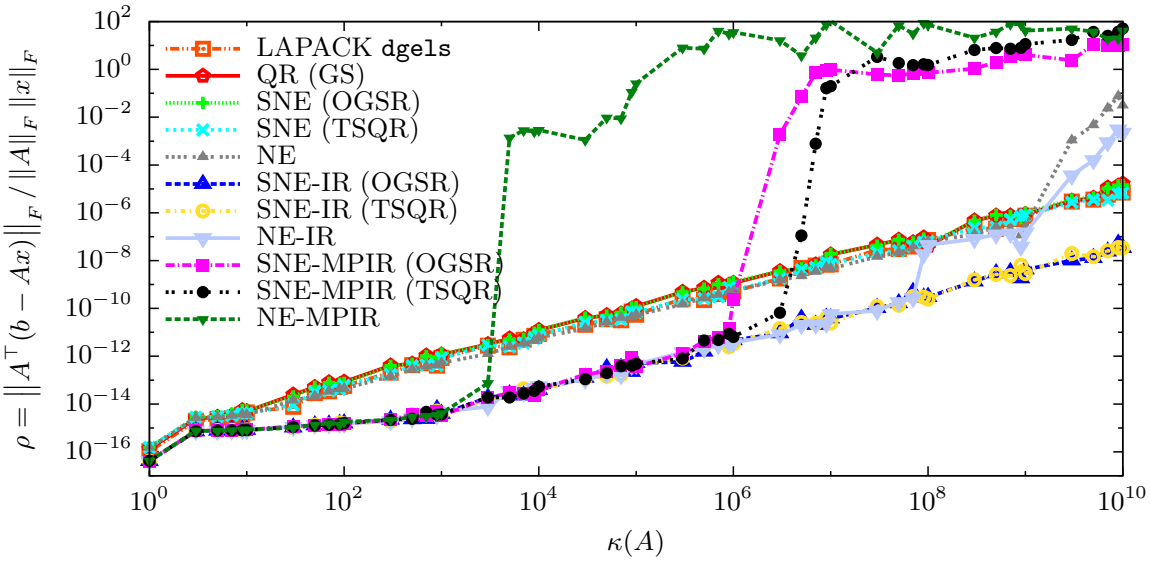


Figure 2: Comparison of the achieved accuracy ρ of all ARPLS methods and LAPACK `dgels` for $n = 1024, m = 64$ and different condition numbers κ. The prefix "ARPLS" is omitted from the legend.

6.3. Numerical Accuracy

The accuracy of the result is determined by considering the relative residual

$$\rho = \frac{\|A^\top r\|_F}{\|A\|_F \|x\|_F}$$

Figure 2 shows the accuracy achieved by the different methods for a set of tall and skinny test matrices generated according to Algorithm 7 with $n = 1024$, $m = 64$ and varying condition numbers $\kappa(A)$. Starting with Figure 2, we omit the prefix "ARPLS" from the legend. The LAPACK subroutine `dgels`, which uses a QR factorisation to solve the LLS problem, is also included. ARPLS-QR and ARPLS-SNE result in the same accuracy, showing that the SNE method has no adverse effect on the numerical accuracy of the result. `dgels` achieves the same accuracy as ARPLS-SNE (TSQR), which is just slightly better than ARPLS-QR (GS) and ARPLS-SNE (OGSR). In general, TSQR achieves the same or slightly better results than armGS. Up until $\kappa = 10^9$, ARPLS-NE also achieves the same accuracy as ARPLS-QR (GS) and ARPLS-SNE. For $\kappa > 10^9$, ARPLS-NE can no longer compute an acceptable result due to the loss of accuracy when forming the normal equations.

Using IR improves the accuracy in almost all cases and reaches relative residuals which can be almost two orders of magnitude lower compared to the methods without IR. The only exception is ARPLS-NE-IR, which only benefits from IR up until $\kappa = 10^8$. For worse conditioned matrices ($\kappa > 10^8$), ARPLS-NE-IR returns residuals close to ARPLS-NE up until $\kappa = 10^9$ and then fails to compute a correct result.

The improvement of the MPIR variants is limited by the accuracy of the working precision $p_\beta = \text{SP}$. If the QR factorisation computed in p_β does not contain any correct digits then MPIR is not able to improve the result. However, for mildly ill-conditioned systems up until $\kappa \approx 10^6$, ARPLS-SNE-MPIR achieves the same accuracy as ARPLS-SNE-IR. As shown in (4), the number of iterations k increases with κ to reach the displayed accuracies. For $\kappa \leq 10^2$ all MPIR variants normally converged after only 2 iterations and for $\kappa \approx 10^5$ ARPLS-SNE-MPIR needed 8 iterations. However, due to the low additional computational complexity of $O(m^2)$ per iteration, the performance of the algorithms is not significantly influenced by the number of iterations. For example, for ARPLS-SNE-MPIR (OGSR) with $n = 2^{22}$ and $m = 256$, an iteration on average only made up 0.005% of the total execution time. Even multiple iterations could be performed at very low cost. When the limiting condition number $\kappa = 2^{23} \approx 8.4 \cdot 10^6$ is reached, the number of iterations grows very fast. As discussed in Section 3.4, the relation p_β/k denotes the number of digits (in bit) which can be improved per iteration. If $k > p_\beta$, the result can no longer be corrected because the improvement per iteration would be less than a bit. In the case of ARPLS-NE-MPIR, forming the normal equations squares the condition number of A and therefore the MPIR method already ceases to achieve an accurate result after $\kappa \approx 10^3$, roughly the square root of the condition number ARPLS-SNE-MPIR is able to handle. Nevertheless, for mildly ill-conditioned systems ARPLS-SNE-MPIR achieves the same accuracy as the double precision IR solvers and additionally benefits from the performance gain due to the use of the lower working precision.

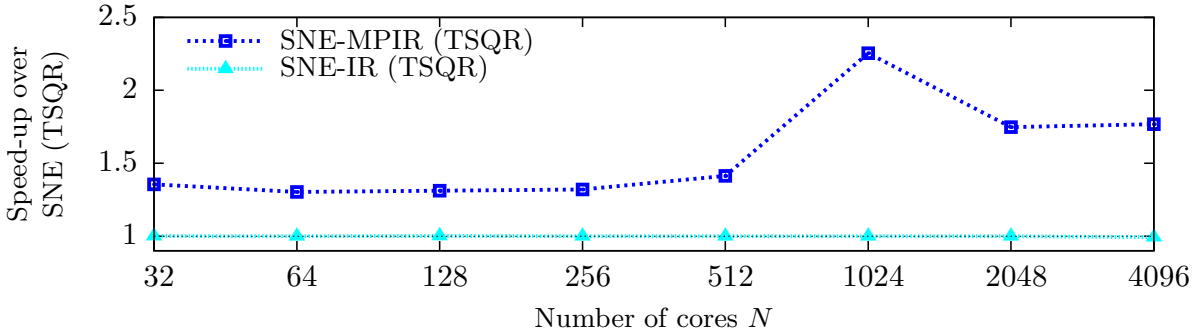
6.4. Runtime Performance and Discussion

In this section, we will consider two different test cases: well-conditioned and ill-conditioned matrices. In the former case, we are interested in the performance benefits achieved through the use of mixed precision in the ARPLS-MPIR variants. In the second test case the performance of the ARPLS-IR solvers is investigated for LLS problems with $\kappa(A) = 10^{10}$. As seen in Figure 2, all ARPLS-NE and ARPLS-MPIR variants are no longer able to achieve an acceptable accuracy for such ill-conditioned systems and are therefore excluded from these experiments.

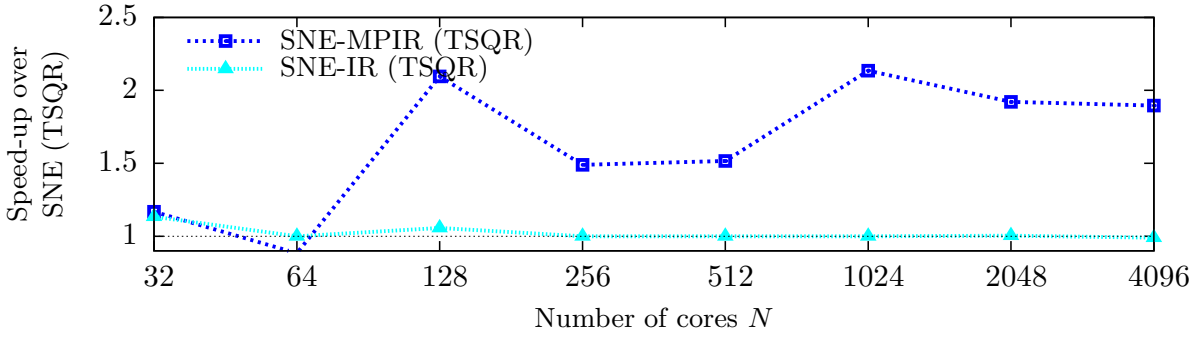
Our experiments have shown that the execution times of the ARPLS methods using armGS and armGSR-Opt to compute the QR factorisation are almost always the same for thin matrices. The optimised version achieves more significant speed-ups for wider matrices due to fewer large messages being sent during the reduction operations. Over all our experiments, ARPLS-SNE-MPIR (OGSR) was faster in 73% of the cases. If we account for fluctuations in the measurements and also integrate almost negligible slow-downs at or above 0.97, then ARPLS-SNE-MPIR (OGSR) was faster in over 87% of the cases. We therefore only show results for the methods using armGSR-Opt.

6.4.1. Well-conditioned LLS Problems

In the following experiments, A and b are initialised with random values between -1 and 1. The IR and MPIR based algorithms are terminated after reaching $\rho \leq 10^{-15}$ which takes place after 2-3 iterations due to the matrices being well-conditioned when generated this way.



(a) Speed-up of the total *execution time*



(b) Speed-up of the *communication time*

Figure 3: Speed-up achieved due to mixed precision iterative refinement for $n = 2^{22}, m = 256$, well-conditioned

ARPLS-SNE using TSQR to compute the QR factorisation benefits from the usage of mixed precision iterative refinement in ARPLS-SNE-MPIR, as shown in Figure 3a. As expected, ARPLS-SNE-IR is slightly slower than ARPLS-SNE due to the additional computations of order $O(n^2)$ required by IR. ARPLS-SNE-MPIR always outperforms ARPLS-SNE and for $n = 2^{22}$ and $m = 256$ achieves speed-ups between 1.3 and 2.3. However, for wider matrices (larger m) the execution time of TSQR increases significantly due to the computation of a QR factorisation at every step of the reduction. Therefore, figures for wide matrices will not include results for any ARPLS methods using TSQR. Focusing only on the communication time, as shown in Figure 3b for the same experiments as in Figure 3a, ARPLS-SNE-MPIR also benefits from the mixed precision approach and achieves speed-ups between 1.5 and 2.1 for $N \geq 128$ due to the data being sent in the lower working precision (SP).

Figure 4 shows the speed-up achieved by different ARPLS algorithms and ScaLAPACK over DPLASMA for very tall and skinny matrices with $n = 2^{22}$ and $m = \{16, 256\}$ for different numbers of cores along the x-axis. For very thin matrices with $m = 16$, ARPLS-SNE-MPIR (OGSR) achieves a speed-up of 2802 and ARPLS-SNE-MPIR (TSQR) of 4477 for the maximum number of cores. For $m = 256$, the speed-ups grow up to 75 for ARPLS-SNE-MPIR (OGSR) and up to 183 ARPLS-SNE-MPIR (TSQR) on 2048 cores. The ARPLS-NE methods outperform the other methods for these well-conditioned matrices, being up to 3.4 times faster than ARPLS-SNE-MPIR (TSQR). For the smaller problem size with $m = 16$, ARPLS-NE-IR and ARPLS-NE-MPIR are slower than ARPLS-NE because the communication dominates their execution time. ARPLS-NE only requires a single all-reduce operation to compute the normal equations, whereas the corresponding IR and MPIR methods additionally require 2-3 iterations to improve the result and have to perform an all-reduce operation in each of those iterations. However, for $m = 256$ ARPLS-NE-MPIR benefits from the use of single precision and achieves the same speed-up as ARPLS-NE (up to 1204 for 2048 cores). ARPLS-NE-IR and ARPLS-NE-MPIR achieve a higher accuracy than ARPLS-NE for these well-conditioned matrices (as seen in Figure 2), but for $m = 256$ ARPLS-NE-MPIR is faster than the standard

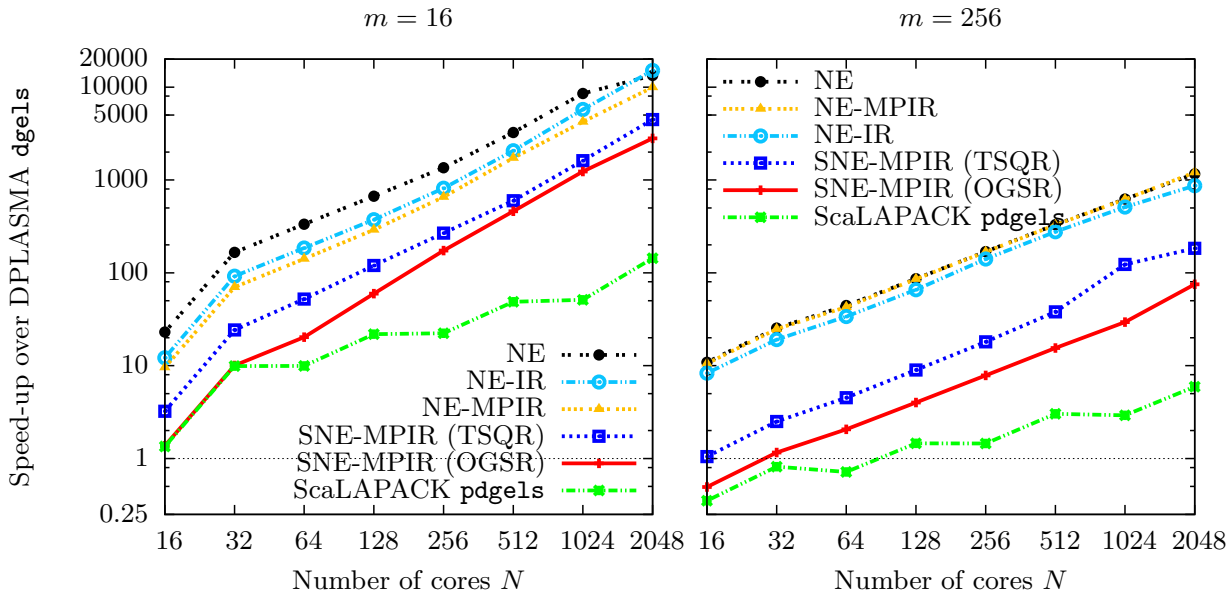


Figure 4: Speed-up over DPLASMA for tall and skinny matrices ($n = 2^{22}$, large n/m, well-conditioned)

IR method and requires the same runtime as ARPLS-NE. The benefit of ARPLS-NE-MPIR continues to increase with growing m and outperforms ARPLS-NE.

6.4.2. Ill-conditioned LLS Problems

The following experiments use worse conditioned matrices by initialising A and b with random values between -1 and 1 and then modifying A to have $\kappa = 10^{10}$ by computing the SVD and scaling the singular values appropriately (as explained in Section 6.2). These experiments were terminated after reaching $\rho \leq 10^{-8}$, which occurred after 3 iterations of standard precision IR. Since these matrices were not well-conditioned, the ARPLS-NE and ARPLS-SNE-MPIR variants could not be used anymore. We therefore focus our attention on the ARPLS-SNE-IR methods.

In Figure 5 the speed-ups achieved with ARPLS-SNE-IR (OGSR) and ARPLS-SNE-IR (TSQR) over DPLASMA are shown for very tall and skinny matrices with $n = 2^{22}$ and $m = \{16, 256\}$ for different numbers of cores along the x-axis. For $m = 256$ the speed-up grows steadily reaching 42 for 2048 cores for ARPLS-SNE-IR (OGSR) and 120 for ARPLS-SNE-IR (TSQR). For even thinner matrices with $m = 16$, the speed-up reaches 3468 for ARPLS-SNE-IR (OGSR) and 3820 for ARPLS-SNE-IR (TSQR) for the maximum number of cores tested on the cluster. ScaLAPACK also achieves speed-ups compared to DPLASMA (up to 143 times faster for $m = 16$) but is generally much slower than both ARPLS-SNE-IR variants (up to 26 times for 2048 cores and $m = 16$).

As also stated in [9], PLASMA or PLASMA-like tiled QR algorithms are more efficient for wide matrices (large m) and are the most efficient for square matrices ($n = m$). These algorithms exploit parallelism to achieve good cache usage and do not perform well on very tall and skinny matrices.

Figures 6 and 7 show the scaling behaviour of the various algorithms with n (for fixed m). For thin matrices with $m = 256$, ARPLS-SNE-IR (OGSR) again displays faster execution times than DPLASMA, for all n being about 5, 12 and 50 times faster for 64, 256 and 1024 cores, respectively. ARPLS-SNE-IR (TSQR) is slower than ARPLS-SNE-IR (OGSR) for smaller n and but does perform better than ARPLS-SNE-IR (OGSR) for $n \geq 2^{16}$. ScaLAPACK is in general slower than DPLASMA and always slower than ARPLS-SNE-IR (OGSR). Analysing the results for wider matrices, as shown for $m = 4096$ in Figure 7, DPLASMA is faster than ARPLS-SNE-IR (OGSR) for 64 cores, but again, due to DPLASMA not scaling further than 256 cores for the tested problem sizes, ARPLS-SNE-IR (OGSR) can achieve a higher performance for 1024 cores most of the time. ScaLAPACK is slower than DPLASMA on 64 cores but performs between 1.3 and

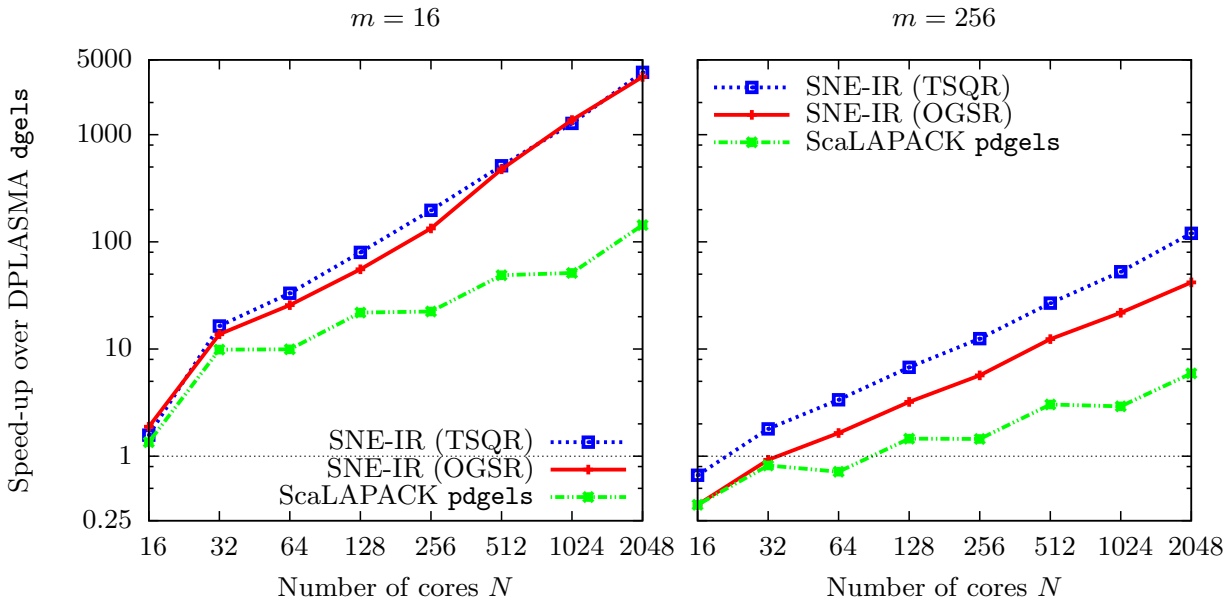


Figure 5: Speed-up over DPLASMA for tall and skinny matrices ($n = 2^{22}$, large n/m, $\kappa = 10^{10}$)

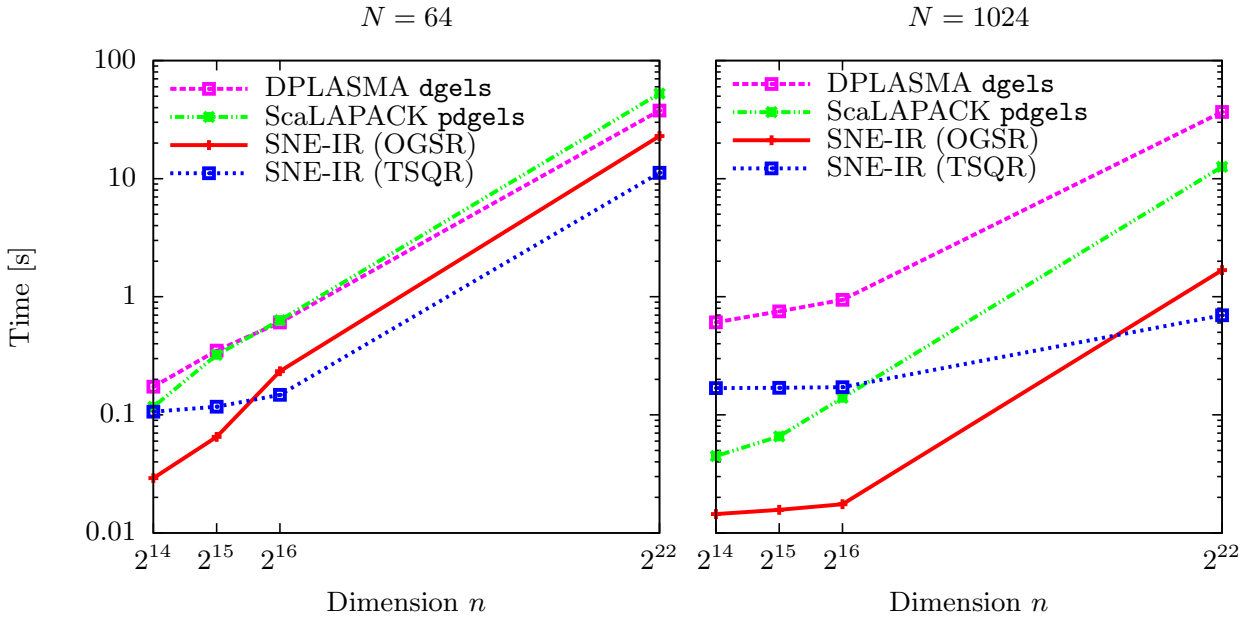


Figure 6: Scaling behaviour for $m = 256$ and growing n ($\kappa = 10^{10}$)

2.4 times better on 1024 cores. Compared to ARPLS-SNE-IR (OGSR), ScaLAPACK is faster on 64 and 1024 cores for these wider matrices.

Figure 8 shows the performance of the tested algorithms for varying number of columns m. With increasing m, DPLASMA comes closer to and also overtakes ARPLS-SNE-IR (OGSR). However, for small values of m ARPLS-SNE-IR (OGSR) performs significantly better than DPLASMA and also exploits the available computing cores more efficiently for skinny matrices. ARPLS-SNE-IR (OGSR) is also faster than ScaLAPACK up until $m = 2048$ on 1024 cores. Considering wider matrices (large m), ARPLS-SNE-IR (OGSR) is in general slower, but with increasing number of cores catches up with and even overtakes

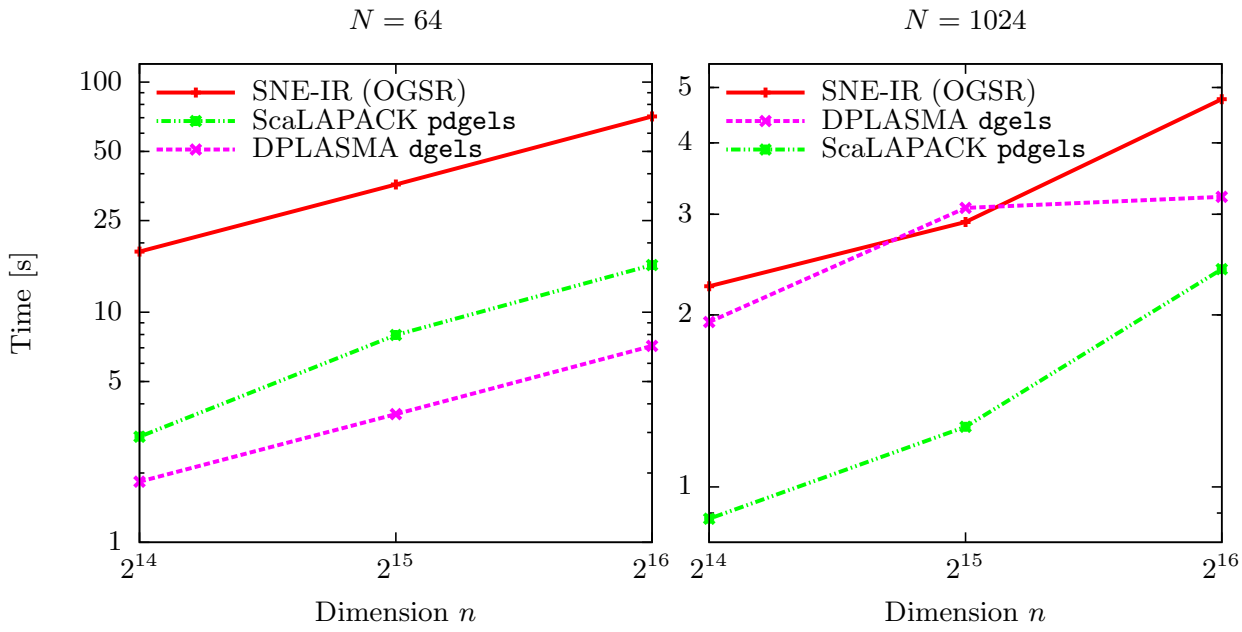


Figure 7: Scaling behaviour for $m = 4096$ and growing n ($\kappa = 10^{10}$)

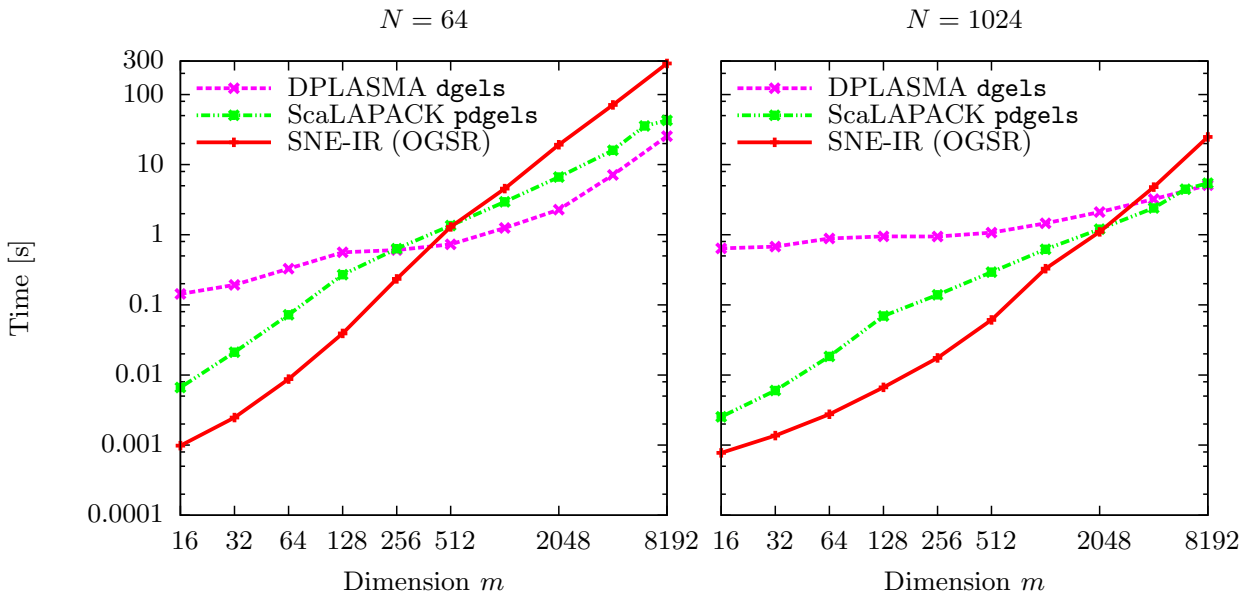


Figure 8: Scaling behaviour for $n = 65536$ and growing m ($\kappa = 10^{10}$)

DPLASMA. ARPLS-SNE-IR (OGSR) profits from the increased number of cores and scales well, reaching a speed-up of up to 1.6 over DPLASMA for $n = 65536$ and $m = 2048$ on $N \geq 512$ cores, as can be seen in Figure 8 for $N = 1024$.

6.5. Communication Cost Analysis

In Figure 9, the execution time is shown for the different parts of the ARPLS-SNE-MPIR (OGSR) algorithm. As one can see, the iterative refinement shown as a green bar always only accounts for a very small percentage of the computation time due to its low complexity compared to the QR factorisation. In

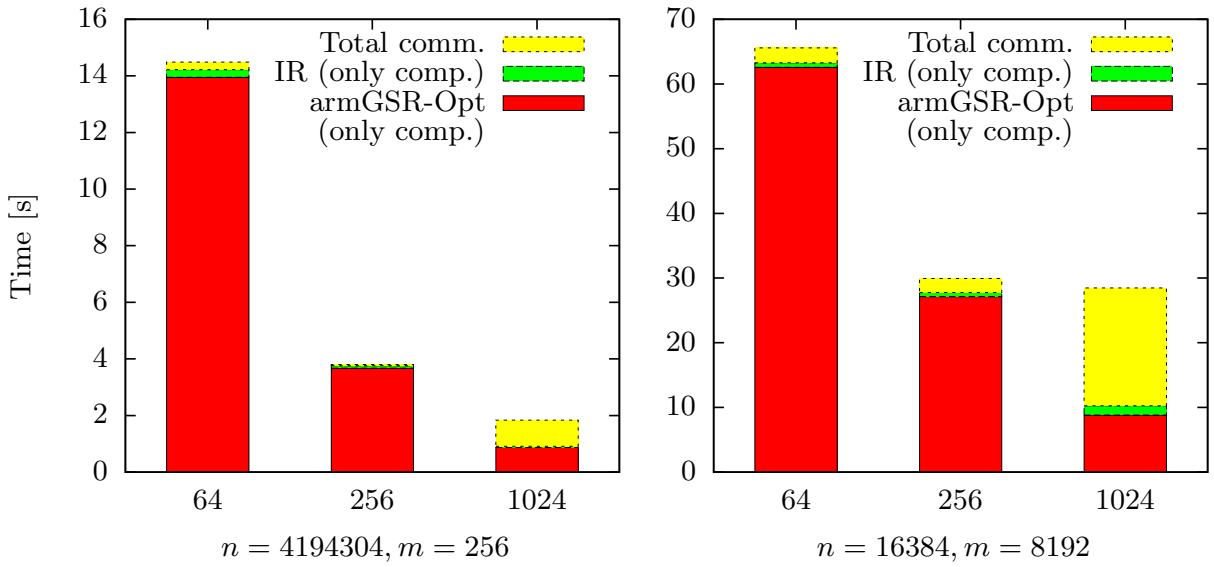


Figure 9: Communication and computation time of ARPLS-SNE-MPIR (OGSR) for different numbers of cores. The matrices are well-conditioned and therefore IR only requires 2-3 iterations to achieve the targeted double precision accuracy.

most cases, armGSR-Opt shown in red makes up for the majority of the execution time. The communication time for both parts shown in yellow depends on the number of columns m. For very skinny matrices it is of course lower than for wider matrices, due to the smaller message sizes. Looking at the communication cost for 1024 cores, it seems that the communication cost suddenly increases and dominates the execution time. However, investigating the communication time more closely and measuring every `MPI_Allreduce` call reveals that only very few `MPI_Allreduce` calls exhibit an above average communication time, leading to a strong increase of the total communication time. The longest `MPI_Allreduce` call for $n = 4194304, m = 256$ took 0.0193 s, whereas the average communication time over all reduction calls is 0.0035 s. For $m = n/2$ on the right side of Figure 9, this behaviour is even more significant: on average only 5 out of 8196 `MPI_Allreduce` calls are responsible for almost 40% of the total communication time. These calls occur randomly throughout the algorithm and are also independent of the message size. The average communication time is 0.0022 s with a standard deviation of 0.0326. Therefore it is fair to assume that this large discrepancy for different `MPI_Allreduce` calls can be accounted to the network infrastructure.

7. Conclusions

We presented the parallel linear least squares solvers ARPLS-IR and ARPLS-MPIR which are based on semi-normal or normal equations and (mixed precision) iterative refinement. We compared two different strategies for the parallel computation of the QR factorisation required in this context (armGS and communication optimal TSQR). In the ARPLS solvers, all communication operations between participating processes are contained in all-reduce operations. Consequently, the ARPLS methods directly benefit from all improvements in such reduction operations (e. g., efficient implementation, optimised communication trees, fault tolerance or variants with localised communication).

The theoretical comparison of the communication cost of the relevant parallel QR factorisation methods revealed an asymptotically higher message count of armGS compared to communication optimal TSQR and CAQR. However, numerical experiments on several thousand cores of a high performance cluster showed competitive runtime performance. We have also shown that the use of *mixed precision* IR in ARPLS-MPIR reduces both, the computation and communication costs. However, mixed precision solvers are limited to matrices with a condition number $\kappa(A) < 10^7$ due to the lower working precision. The experiments confirmed that ARPLS-IR also scales very well with the number of cores and outperforms the parallel

dense LLS solvers in the state-of-the-art libraries DPLASMA and ScaLAPACK for several test cases. In particular, for tall and skinny matrices ARPLS-IR exploits the available computing power efficiently and scales very well with increasing processor count as illustrated by our experiments for up to 2048 cores on a high performance cluster. ARPLS-IR achieves speed-ups up to 3820 on 2048 cores over the state-of-the-art solvers from DPLASMA and up to 26 on 2048 cores over ScaLAPACK for very tall and very skinny matrices. ARPLS-SNE-IR, which uses standard precision IR, achieves better accuracy than all other solvers and the results for $\kappa(A) > 10^8$ are improved by about two magnitudes. In contrast, the normal equation-based solver ARPLS-NE, although faster for well-conditioned problems, fails to compute a correct result for those ill-conditioned systems due to squaring the condition number when forming the cross product.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the paper. This work was partly supported by the Austrian Science Fund (FWF) in project S10608 (NFN SISE). The computational results presented have been achieved using the Vienna Scientific Cluster (VSC)[2].

[1] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, M. Welsh, Monitoring volcanic eruptions with a wireless sensor network, in: Proceeedings of the Second European Workshop on Wireless Sensor Networks, IEEE, 2005, pp. 108–120. doi:10.1109/EWSN.2005.1462003.

[2] Z. Cai, T. A. Manteuffel, S. F. McCormick, First-Order System Least Squares for Second-Order Partial Differential Equations: Part II, SIAM Journal on Numerical Analysis 34 (2) (1997) 425–454. doi:10.1137/S0036142994266066.

[3] M. Lipson, P.-R. Loh, A. Levin, D. Reich, N. Patterson, B. Berger, Efficient Moment-Based Inference of Admixture Parameters and Sources of Gene Flow, Molecular Biology and Evolution 30 (8) (2013) 1788–1802. doi:10.1093/molbev/mst099.

[4] J. Demmel, L. Grigori, M. Hoemmen, J. Langou, Communication-optimal Parallel and Sequential QR and LU Factorizations, SIAM J. Sci. Comput. 34 (1) (2012) 206–239. doi:10.1137/080731992.

[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, ScaLAPACK Users' Guide, SIAM, Philadelphia, PA, 1997.

[6] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov, Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects, Journal of Physics: Conference Series 180 (1) (2009) 012037. doi:10.1088/1742-6596/180/1/012037.

[7] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, J. Dongarra, Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA, in: IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011, pp. 1432–1441. doi:10.1109/IPDPS.2011.299.

[8] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, Parallel tiled QR factorization for multicore architectures, Concurrency and Computation: Practice and Experience 20 (13) (2008) 1573–1590. doi:10.1002/cpe.1301.

[9] B. Hadri, H. Ltaief, E. Agullo, J. Dongarra, Tile QR factorization with parallel panel processing for multicore architectures, in: IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2010, pp. 1–10. doi:10.1109/IPDPS.2010.5470443.

[10] G. Aupy, M. Faverge, Y. Robert, J. Kurzak, P. Luszczek, J. Dongarra, Implementing a Systolic Algorithm for QR Factorization on Multicore Clusters with PaRSEC, in: Euro-Par 2013: Parallel Processing Workshops, Vol. 8374 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014, pp. 657–667. doi:10.1007/978-3-642-54420-0_64.

[11] R. A. Renaut, A Parallel Multisplitting Solution of the Least Squares Problem, Numer. Linear Algebr. 5 (1) (1998) 11–31. doi:10.1002/(SICI)1099-1506(199801/02)5:1<11::AID-NLA123>3.0.CO;2-F.

[12] A. H. Sayed, Diffusion Adaptation over Networks, Academic Press Library in Signal Processing, 1st Edition, Elsevier, 2013, Ch. 9, pp. 323–456.

[13] I. Schizas, Consensus in ad hoc WSNs with noisy links - Part II: Distributed estimation and smoothing of random signals, IEEE T. Signal Proces. 56 (4) (2008) 1650–1666. doi:10.1109/TSP.2007.908943.

[14] K. E. Prikopa, H. Straková, W. N. Gansterer, Analysis and Comparison of Truly Distributed Solvers for Linear Least Squares Problems on Wireless Sensor Networks, in: Euro-Par 2014 Parallel Processing, Vol. 8632 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 403–414. doi:10.1007/978-3-319-09873-9_34.

[15] N. Higham, Accuracy and Stability of Numerical Algorithms: Second Edition, SIAM, 2002.

[16] A. Björck, Stability analysis of the method of seminormal equations for linear least squares problems, Linear Algebra Appl. 48 (1987) 31–48. doi:10.1016/0024-3795(87)90101-7.

[17] G. Golub, Numerical methods for solving linear least squares problems, Numer. Math. 7 (3) (1965) 206–216. doi:10.1007/BF01436075.

[2] http://vsc.ac.at/

[18] A. Björck, Iterative refinement of linear least squares solutions I, BIT 7 (4) (1967) 257–278. doi:10.1007/BF01939321.
[19] J. H. Wilkinson, Rounding Errors in Algebraic Processes, Her Majesty's Stationery Office, 1963.
[20] C. B. Moler, Iterative Refinement in Floating Point, J. ACM 14 (2) (1967) 316–321. doi:10.1145/321386.321394.
[21] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, E. J. Riedy, Error bounds from extra-precise iterative refinement, ACM T. Math. Software 32 (2) (2006) 325–351. doi:10.1145/1141885.1141894.
[22] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, J. Dongarra, Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy (Revisiting Iterative Refinement for Linear Systems), in: Proceedings of the ACM/IEEE SC 2006 Conference, 2006, pp. 50–50. doi:10.1109/SC.2006.30.
[23] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, S. Tomov, Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy, ACM T. Math. Software 34 (4) (2008) 17:1–17:22. doi:10.1145/1377596.1377597.
[24] J. Kurzak, A. Buttari, J. Dongarra, Solving systems of linear equations on the CELL processor using Cholesky factorization, IEEE T. Parall. Distr. 19 (9) (2008) 1175–1186. doi:10.1109/TPDS.2007.70813.
[25] M. Jankowski, H. Woźniakowski, Iterative refinement implies numerical stability, BIT 17 (1977) 303–311.
[26] A. Björck, Numerical methods for least squares problems, SIAM, 1996.
[27] J. Demmel, Y. Hida, E. J. Riedy, X. S. Li, Extra-Precise Iterative Refinement for Overdetermined Least Squares Problems, ACM T. Math. Software 35 (4) (2009) 28:1–28:32. doi:10.1145/1462173.1462177.
[28] J. Rice, Matrix computations and mathematical software, McGraw-Hill, 1981.
[29] H. Straková, W. N. Gansterer, T. Zemen, Distributed QR Factorization Based on Randomized Algorithms, in: Parallel Processing and Applied Mathematics, Vol. 7203 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 235–244. doi:10.1007/978-3-642-31464-3_24.
[30] J. Demmel, L. Grigori, M. F. Hoemmen, J. Langou, Communication-optimal Parallel and Sequential QR and LU Factorizations, Tech. Rep. UCB/EECS-2008-89, EECS Department, University of California, Berkeley (2008).
[31] E. Agullo, C. Coti, J. Dongarra, T. Herault, J. Langem, QR factorization of tall and skinny matrices in a grid computing environment, in: IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2010, pp. 1–11. doi:10.1109/IPDPS.2010.5470475.
[32] E. Elmroth, F. Gustavson, Applying recursion to serial and parallel QR factorization leads to better performance, IBM Journal of Research and Development 44 (2000) 605–624. doi:10.1147/rd.444.0605.
[33] W. N. Gansterer, G. Niederbrucker, H. Straková, S. Schulze-Grotthoff, Scalable and Fault Tolerant Orthogonalization Based on Randomized Distributed Data Aggregation, Journal of Computational Science 4 (6) (2013) 480 – 488. doi:10.1016/j.jocs.2013.01.006.
[34] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard Version 3.0 (2012).
[35] R. Thakur, R. Rabenseifner, W. Gropp, Optimization of Collective Communication Operations in MPICH, International Journal of High Performance Computing Applications 19 (1) (2005) 49–66. doi:10.1177/1094342005051521.
[36] R. Schöne, R. Tschüter, T. Ilsche, D. Hackenberg, The VampirTrace Plugin Counter Interface: Introduction and Examples, in: Proceedings of the 2010 Conference on Parallel Processing, Euro-Par 2010, Springer-Verlag, 2011, pp. 501–511. doi:10.1007/978-3-642-21878-1_62.