Science & Technology Facilities Council

Technical Report
RAL-TR-2017-010

Sparse direct solution on parallel computers

I Duff, F Lopez, S Nakov

September 2017

Enquiries concerning this report should be addressed to:

RAL Library
STFC Rutherford Appleton Laboratory
Harwell Oxford
Didcot
OX11 0QX

Tel: +44(0)1235 445384
Fax: +44(0)1235 446403
email: libraryral@stfc.ac.uk

Science and Technology Facilities Council reports are available online at: http://epubs.stfc.ac.uk

ISSN 1358-6254

Sparse direct solution on parallel computers

Iain Duff, Florent Lopez, and Stojce Nakov

ABSTRACT

We describe our recent work on designing algorithms and software for solving sparse systems using direct methods on parallel computers. This work has been conducted within an EU Horizon 2020 Project called NLAFET. We first discuss the solution of large sparse symmetric positive definite systems. We use a runtime system to express and execute a DAG-based Cholesky factorization. The runtime system plays the role of a software layer between the application and the architecture and handles the management of task dependencies as well as task scheduling and maintaining data coherency. Although runtime systems are widely used in dense linear algebra, this approach is challenging for sparse algorithms because of the irregularity and variable granularity of the DAGs arising in these systems. We have implemented our software using the OpenMP standard and the runtime systems StarPU and PaRSEC. We compare these implementations to `HSL_MA87`, a state-of-the-art DAG-based solver for positive definite systems. We demonstrate comparable performance on a multicore architecture. We also consider the case when the matrix is symmetric indefinite. For highly unsymmetric systems we use a completely different approach based on developing a parallel version of a Markowitz threshold ordering. This work is less advanced but we discuss some of the algorithmic challenges involved. Finally, we briefly discuss using a hybrid direct-iterative solver that combines the best of the two approaches and enables the solution of even larger problems in parallel.

Keywords: sparse Cholesky, symmetric indefinite, runtime systems, highly unsymmetric matrices, hybrid methods, block Cimmino

AMS(MOS) subject classifications: 65F30, 65F50

Scientific Computing Department, STFC Rutherford Appleton Laboratory, Harwell Campus, Oxfordshire, OX11 0QX, UK.
Correspondence to: iain.duff@stfc.ac.uk

NLAFET Working Note 17.

This work is supported by the NLAFET Project funded by the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement 671633.

August 25, 2017

Contents

1 Introduction

We discuss recent work on the solution of large sparse equations of parallel computers using direct methods in the context of an EU Horizon 2020 Project called NLAFET (Parallel Numerical Linear Algebra for Future Extreme Scale Systems)[1]. This is the H2020 FET-HPC Project 671633 and it involves only four partners. The coordinator is Bo Kågström of Umeå University in Sweden, and the other Principal Investigators are Iain Duff (STFC, UK), Jack Dongarra (University of Manchester, UK) and Laura Grigori (INRIA, Paris). The Project started on 1 November 2015 and will finish on the 31st October 2018.

A major aim of the project is to enable a radical improvement in the performance and scalability of a wide range of real-world applications relying on linear algebra software for future extreme-scale systems.

The key goals are:

- Development of novel architecture-aware algorithms that expose as much parallelism as possible, exploit heterogeneity, avoid communication bottlenecks, respond to escalating fault rates, and help meet emerging power constraints.
- Exploration of advanced scheduling strategies and runtime systems, focusing on the extreme scale and strong scalability in multi/many-core and hybrid environments.
- Design and evaluation of novel strategies and software support for both offline and online auto-tuning.
- The results will appear in an open source NLAFET software library.

1.1 NLAFET workpackage overview

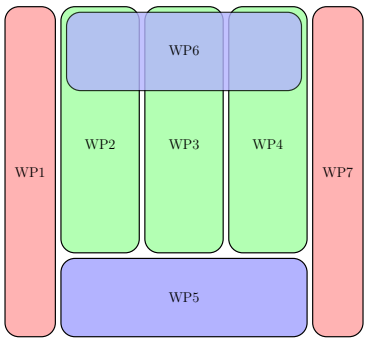

Figure 1.1: Overview of NLAFET Project.

The NLAFET Project consists of seven interlocking workpackages as shown in Figure 1.1. The three workpackages WP1, WP5, and WP7 concern administration, applications, and dissemination while the other four workpackages define our research agenda. We list the workpackages below.

- WP1: Management and coordination.
- WP2: Dense linear systems and eigenvalue problem solvers.
- WP3: Direct solution of sparse linear systems.
- WP4: Communication-optimal algorithms for iterative methods.
- WP5: Challenging applications– a selection.
  Material science, power systems, study of energy solutions, and data analysis in astrophysics.

[1] `www.nlafet.eu`

- WP6: Cross-cutting issues.
  Scheduling and runtime systems, auto-tuning, fault tolerance.
- WP7: Dissemination and community outreach.

This paper is primarily concerned with WP3 in which we have four tasks, namely

T3.1 Lower bounds on communication for sparse matrices.

T3.2 Direct methods for (near-)symmetric sparse systems.

T3.3 Direct methods for highly unsymmetric sparse systems.

T3.4 Hybrid direct-iterative methods.

We will primarily be discussing the work in tasks T3.2 and T3.3 but, before we do so, we will first define what we mean by a direct method.

2 Direct solution of sparse equations

When solving the linear system

$$Ax \;=\; b,$$

where the sparse matrix A is of large dimension, typically 10^6 or greater, by direct methods we will consider the factorization:

$$P_r A P_c \;=\; LU,$$

where L is a sparse lower triangular matrix and U is a sparse upper triangular matrix and the permutations P_r and P_c are chosen to preserve sparsity and maintain stability. When A is symmetric, then $P_r = P_c^T$ and the factorization can be written as LL^T (Cholesky) or LDL^T; this latter factorization is needed when A is not positive definite.

There are several points to note about sparse direct methods:

- Black box solvers are available.
- Routinely solving problems of order in millions.
- Complexity can be low.
- Storage requirements can be very high. Although almost linear storage in 2D.
- Target is half asymptotic speed of `GEMM`.

We now discuss how we implement this factorization in the case of Task 3.2.

3 Task 3.2 Direct methods for (near-)symmetric systems

Task 3.2 considers the factorizations LL^T, LDL^T, and LU and all the algorithms will be based on a tree-based representation of the factorization that we describe in Section 3.1. We will follow the approach taken by the code `HSL_MA87` [19] for obtaining more fine-grained parallelism by using directed acyclic graphs (DAGs) rather than trees. The use of a DAG as opposed to a tree is illustrated in Section 4.4. `HSL_MA87` is based on a low level synchronization API for handling the parallelism. The main novelty in our approach is to use runtime systems to both express and execute the DAG. We do this working closely with our NLAFET partners in WP6.

A key point about sparse factorization methods is that the kernels involve operations on small dense matrices and we design energy-efficient, low-communication dense kernels for use within our sparse factorizations both for positive definite and indefinite systems.

Within the framework of NLAFET, we are primarily concerned with the runtime systems StarPU and OpenMP (using task features in Version 4.0 or above) both using an STF (sequential task flow) model, and PaRSEC using a PTG (parametrized task graph) model. In all cases, the structure supplied to the runtime system is a DAG.

3.1 Tree-based factorization

We feel it is useful to illustrate our approach to sparse factorization using the small 7×7 matrix:

$$\begin{array}{c|ccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & \times & \times & \times & \bullet & \bullet & \times & \bullet \\ 2 & \times & \times & \times & \bullet & \bullet & \bullet & \times \\ 3 & \times & \times & \times & \bullet & \bullet & \bullet & \bullet \\ 4 & \bullet & \bullet & \bullet & \times & \times & \times & \bullet \\ 5 & \bullet & \bullet & \bullet & \times & \times & \bullet & \bullet \\ 6 & \times & \bullet & \bullet & \times & \bullet & \times & \times \\ 7 & \bullet & \times & \bullet & \bullet & \bullet & \times & \times \end{array} \tag{3.1}$$

where the entries marked $\times$ are nonzero and those marked $\bullet$ are zero. Then, if we consider eliminating the first three rows and columns at the first step, the matrix on which we perform the eliminations is

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 6 & 7 \\ 1 & \times & \times & \times & \times & \\ 2 & \times & \times & \times & \times & \times \\ 3 & \times & \times & \times & \times & \times \\ & & & & & \\ 6 & \times & \times & \times & \times & \times \\ 7 & & \times & \times & \times & \times \end{array}$$

because the first three entries in rows and columns 4 and 5 are zero.

If we eliminate rows and columns 4 and 5 at step 2, then this can be performed within the matrix:

$$\begin{array}{c|ccc} & 4 & 5 & 6 \\ 4 & \times & \times & \times \\ 5 & \times & \times & \times \\ & & & \\ 6 & \times & \times & \times \end{array}$$

The remaining part of the original matrix can then be factorized, but we need first to update the entries according to the first two pivot steps. Thus step 3 can be expressed as:

$$\begin{array}{c|cc} & 6 & 7 \\ 6 & \times & \times \\ 7 & \times & \times \end{array} + \begin{array}{c|cc} & 6 & 7 \\ 6 & \times & \times \\ 7 & \times & \times \end{array} + \begin{array}{c|c} & 6 \\ 6 & \times \end{array} \longrightarrow \begin{array}{c|cc} & 6 & 7 \\ 6 & \times & \times \\ 7 & \times & \times \end{array}$$

where the three matrices summed on the left are from the original matrix and from the Schur complements of the factorizations at steps 1 and 2, respectively. The resulting dense matrix on the right is then factorized to complete the factorization of the 7×7 matrix (3.1).

The factorization can be represented by the tree shown in Figure 3.1 where the number in each node corresponds to the pivot step eliminating the variables shown in parenthesis and data must be passed along the tree edges from nodes 1 and 2 to node 3. At each node, a small dense matrix, called a frontal matrix, is partially factorized and the Schur complement is passed for assembly at the parent node of the

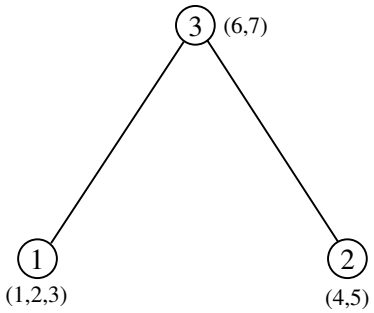

Figure 3.1: Assembly tree for 7×7 example.

tree. The reader is referred to [10] for a longer and more detailed discussion of the use of assembly trees in sparse factorization.

This tree representation can be extended to any symmetric matrix and the sparse factorizations will have similar kernels to those used in the small example. The computation at a node involves dense factorization. Pivots are chosen from the top left block in Figure 3.2, but elimination operations are performed on the whole frontal matrix. Rows and columns of the factors can be stored and the resulting Schur complement (bottom right block) is passed up the tree for future assemblies.

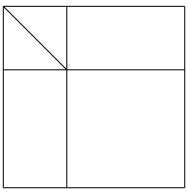

Figure 3.2: Generic frontal matrix showing fully summed block.

4 Parallelism in sparse direct methods

There are several levels of parallelism in solving sparse systems that we discuss in this section. These levels can all contribute to an efficient parallel solution and can result in potentially high levels of exploitation of extreme scale computers. It is the intention of our project to investigate the combination of these techniques to obtain good scalability.

We list the levels below and discuss each in the subsections of this section.

- Partitioning.
- Tree level parallelism.
- Node parallelism (including multi-threaded BLAS).
- Inter-node parallelism.

4.1 Partitioning

In many applications, the matrix can be reordered and partitioned so that all the nonzero entries lie in blocks. For example, if the symmetric system is reducible then the resulting form is block diagonal and the factorization of each block can proceed independently in parallel.

For unsymmetric matrices, reducibility corresponds to a block triangular form and a generalization of this is a bordered block diagonal form, shown in Figure 4.1, that we exploit using the methods discussed in Section 8.

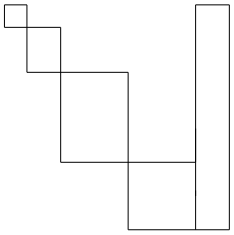

Figure 4.1: A singly bordered block diagonal form.

4.2 Tree level parallelism

We already showed an assembly tree for our 7×7 example (3.1) in Figure 3.1. In the more general case, we can represent the factorization by a similar assembly tree with many more nodes and edges. We show an example of a larger tree in Figure 4.2.

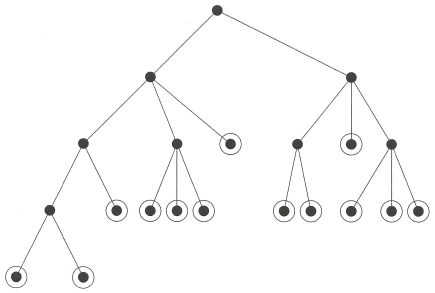

Figure 4.2: Start of factorization. Work corresponding to leaf nodes (circled) can proceed immediately and independently.

As we see in this figure, there is plenty of parallelism available from the tree at the start of the factorization. As the elimination operations at a node complete, other nodes receive the Schur complement information from their children and factorizations can proceed when the data has been received. We show the situation part-way through the factorization in Figure 4.3.

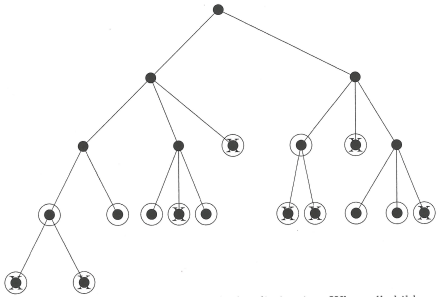

Figure 4.3: Part-way through the factorization. When the dense factorization at a node is finished a cross is superimposed on the circle.

4.3 Node parallelism

Node parallelism comes from the parallel execution of kernels effecting the dense factorization at a node of the tree. Indeed, we can potentially use any tricks developed for parallel dense factorization so that we see immediately the much greater potential for parallelism in the sparse case.

		Tree	Leaf nodes		Top 3 levels		
Matrix	Order	nodes	No.	Av. size	No.	Av. size	% ops
`bratu3d`	27 792	12 663	11 132	8	296	37	56
`cont-300`	180 895	90 429	74 673	6	10	846	41
`cvxqp3`	17 500	8 336	6 967	4	48	194	70
`mario001`	38 434	15 480	8 520	4	10	131	25
`ncvxqp7`	87 500	41 714	34 847	4	91	323	61
`bmw3_2`	227 362	14 095	5 758	50	11	1 919	44

Table 4.1: Statistics on front sizes in assembly tree. From Duff, Erisman, Reid [10].

We show some tree statistics in Table 4.1 for some medium sized problems from a range of applications. Although there are many tasks near the leaf nodes, we note that the dimensions of the matrices are small so there is little node parallelism but plenty of tree parallelism. Near the root node, there is not much tree parallelism but the nodes are large, that is the dense matrices at these nodes are of large dimension and so there is plenty of node parallelism. It is this balance that encourages us to believe that good levels of scalability can be obtained and that our target of half the asymptotic speed might be achievable.

4.4 Inter-node parallelism

The bottleneck in parallel algorithms just based on the assembly tree and employing the parallelism discussed in Sections 4.2 and 4.3 is that a node has to wait for all its children to complete before it is available to start its own processing. We follow previous work on qr_mumps [1, 2, 6] and `HSL_MA87` [19] to overcome this limitation by dividing the factorization into subblocks (called tiles) as shown in Figure 4.4 and then defining the dependency of the resulting task-based approach by using a directed acyclic graph or DAG. We illustrate such a DAG in Figure 4.5.

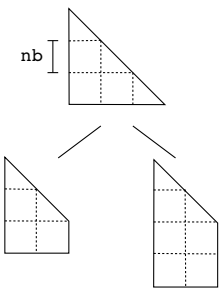

Figure 4.4: Part of tree.

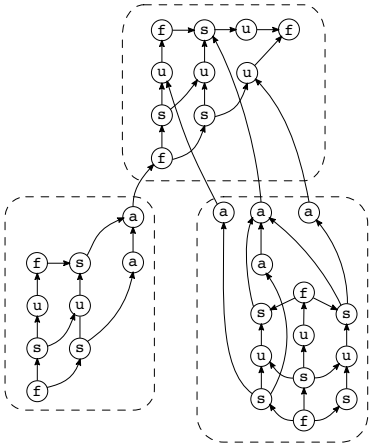

Figure 4.5: Directed acyclic graph.

In the DAG shown in Figure 4.5 the letters in the circles correspond to kernels involved in the factorization of the nodes of the tree. These kernels are:

- the tasks `f` correspond to the Cholesky factorization of a block corresponding to a tile on the diagonal,

- the tasks `s` represent a triangular solve on a subdiagonal block using a factor computed by a task `f`,
- the tasks `u` perform an update of a block within the node using the blocks created by the kernels `s`, and
- the tasks `a` represent an update between nodes with factorizations at one node updating a block in an ancestor node.

5 Experiments on symmetric positive definite systems

We run our experiments on a multicore Haswell machine equipped with two Intel(R) Xeon(R) E5-2695 v3 CPUs with fourteen cores each (twenty eight cores in total). Each core is clocked at 2.3 GHz. The asymptotic performance of DGEMM using PLASMA was 768 Gflop/s. We have implemented a sparse factorization routine, SpLLT, exploiting the parallelism described in Sections 4.2 to 4.4 and have used both OpenMP and the StarPU runtime system [12] using the STF model. We compare runs of this code with `HSL_MA87` on a range of test problems whose characteristics are given in Table A.1 in the Appendix. We see, in Figure 5.1, that the performance is generally comparable to the hand-coded HSL code although rather poor performance is seen in a few matrices, for example matrices 15, 19, and 24. A problem with using the runtime systems is that when the tasks are small, the overhead in setting them up in the runtime system predominates and the overhead is particularly high for StarPU. Another reason why the STF model can show poor performance is that the DAG is traversed sequentially and so some parallelism is missed. To avoid this issue with STF models, we also implemented our algorithm using the PaRSEC runtime

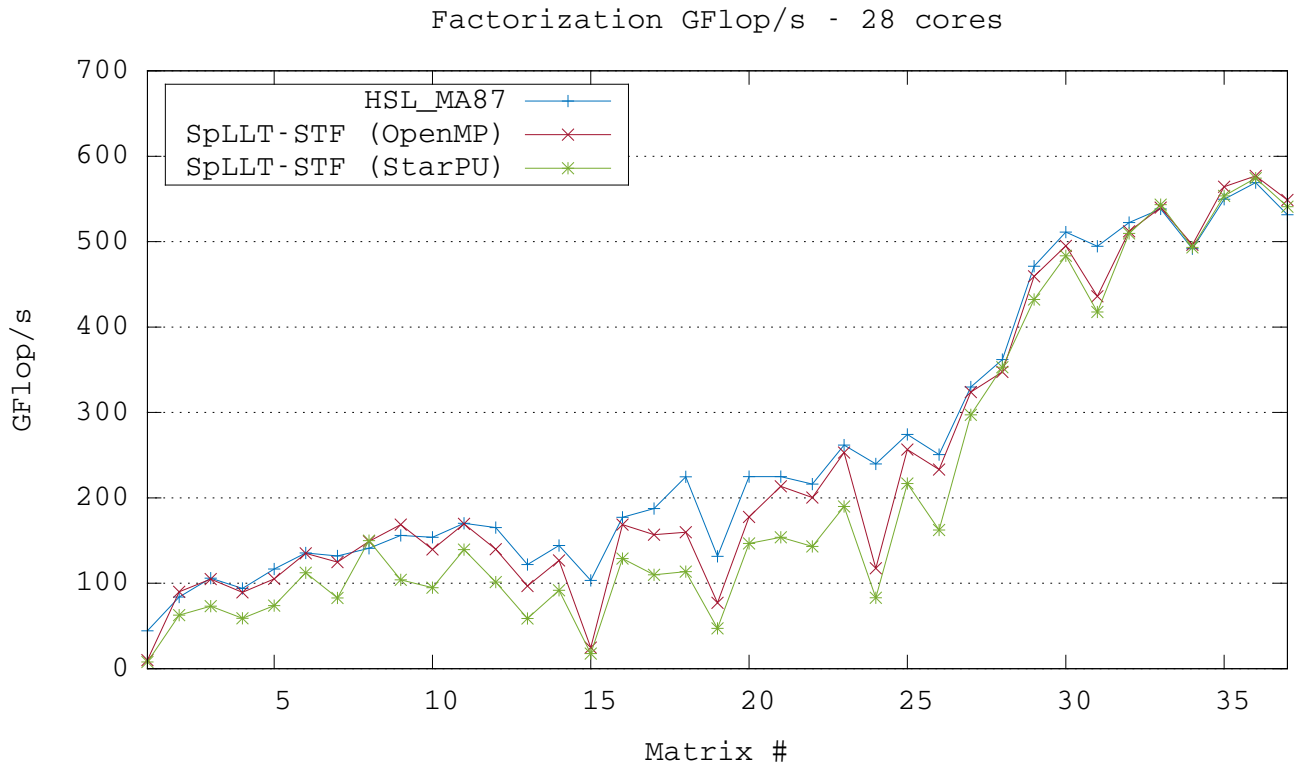

Figure 5.1: Performance of sparse Cholesky code SpLLT using STF runtime systems.

system [13] that implements a PTG model. We see in Figure 5.2 that PaRSEC performs much better than the STF models on the matrices that we just identified, but the cost of using a runtime system still

penalizes cases where there are many small tasks. We can avoid some of this by grouping together into a single task nodes near the leaves of the tree. We call this tree pruning and discuss this in the next section.

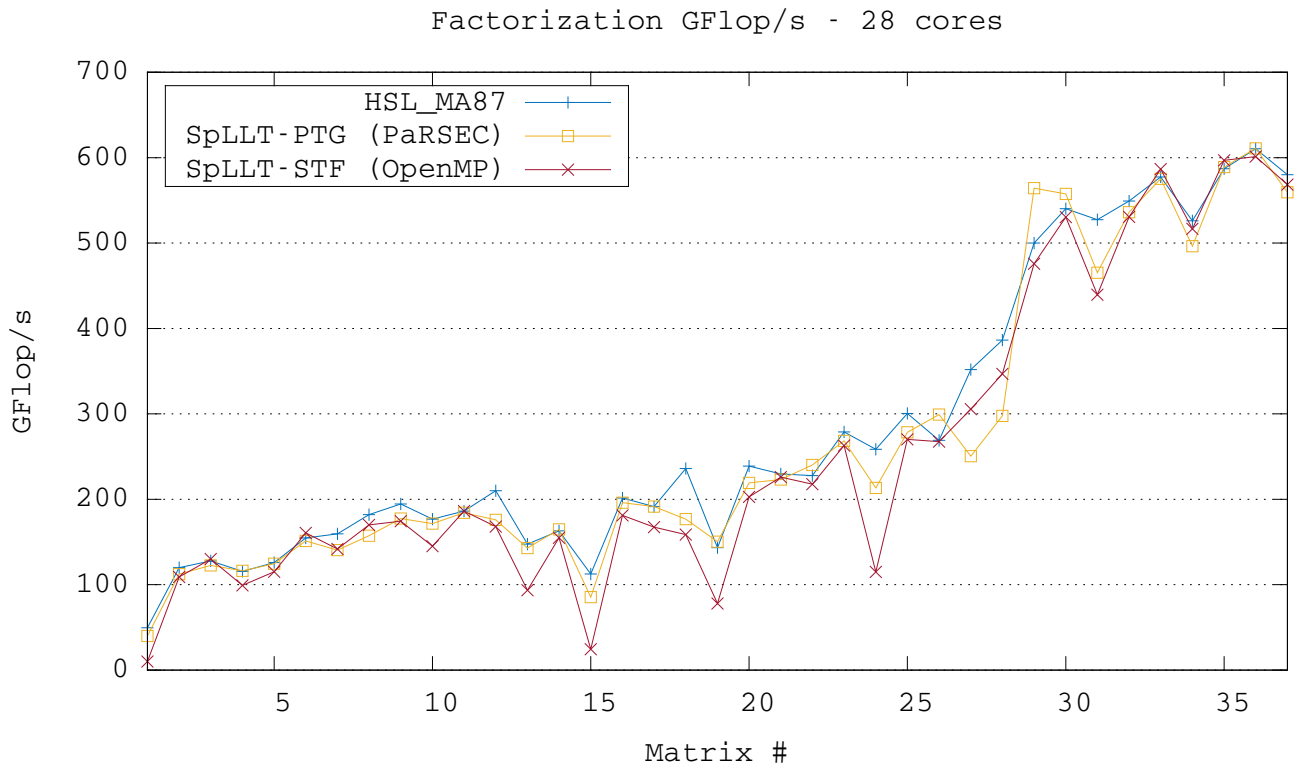

Figure 5.2: Performance of sparse Cholesky code SpLLT using PaRSEC.

5.1 Tree pruning strategy

What we mean by tree pruning is that we do not pursue full tree parallelism by allocating tasks right down to the level of the leaves but we combine the operations in a subtree as a single task so that the granularity of work in that task is increased. This is illustrated in Figure 5.3 where instead of processing the tree right down to the six leaf nodes, we only exploit tree parallelism down to the level of the four heavily shaded nodes and the subtree, for which each is a root, is processed as a single task. This strategy is commonly used when performing a static distribution of work in a multiprocessor environment, for example by MUMPS [3] and, in a shared memory environment, by qr_mumps [6].

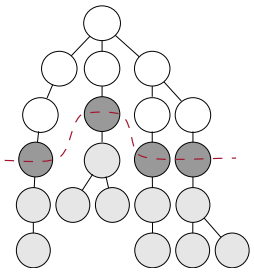

Figure 5.3: An illustration of tree pruning.

We see the effect of doing this pruning in Figure 5.4 where the performance on matrices 15, 19, and 24 is clearly much better. We note that the version with pruning is not always better because of the aforementioned loss of parallelism.

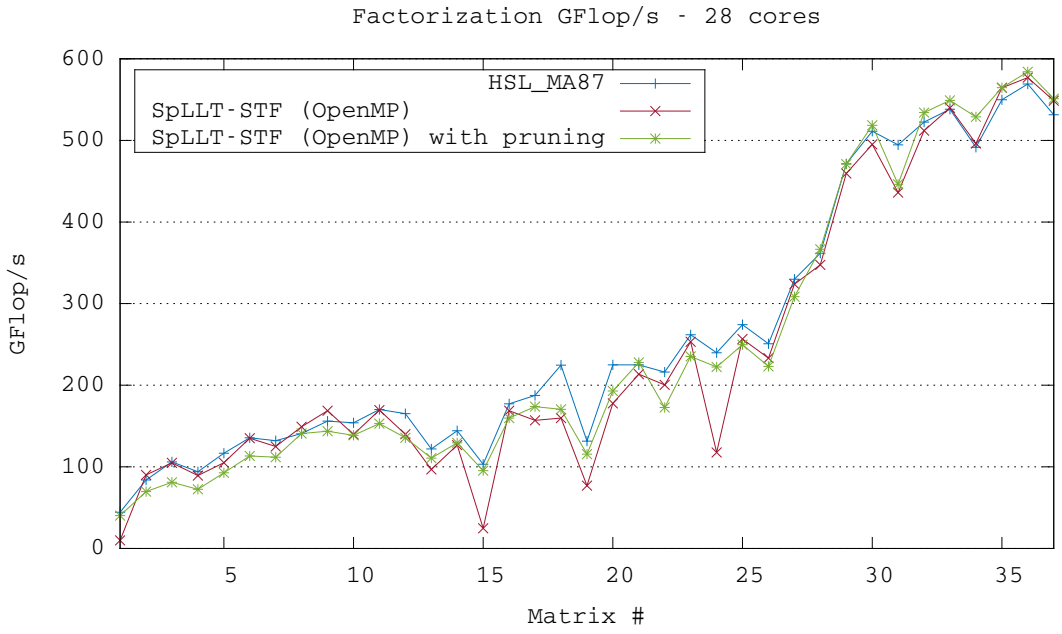

Figure 5.4: Effect of tree pruning on OpenMP version.

6 Symmetric indefinite matrices

If the matrix is indefinite then numerical pivoting is needed. A simple example is the matrix

$$\begin{bmatrix} 0 & \times \\ \times & 0 \end{bmatrix}$$

where it is clear that, whatever, the value of the entries $\times$, a Cholesky factorization will fail because of the zeros on the diagonal. We note that, if $\times$ is nonzero, the matrix is nonsingular. The good news is that we can stably factorize an indefinite matrix using only 1×1 and 2×2 pivots [4, 5].

As is standard in sparse factorization, we use threshold rather than partial pivoting so we want

$$\|Pivot\| \geq u \times \|Largest\ entry\ in\ column\| \tag{6.1}$$

where u is the threshold parameter ($0 < u \leq 1$) so that a value of 1.0 for the threshold parameter would be equivalent to partial pivoting while relaxing this value gives us more scope to choose pivots on sparsity grounds while still retaining some control over the stability of the factorization. This trade-off is described in detail by [10].

For the symmetric indefinite case, where 2×2 pivots may be needed to preserve stability and symmetry, Duff and Reid [15] recommend, for each 2×2 pivot, the test

$$\left| \begin{bmatrix} a_{kk}^{(k)} & a_{k,k+1}^{(k)} \\ a_{k+1,k}^{(k)} & a_{k+1,k+1}^{(k)} \end{bmatrix}^{-1} \right| \begin{bmatrix} \max\limits_{j \neq k,k+1} |a_{kj}| \\ \max\limits_{j \neq k,k+1} |a_{k+1,j}| \end{bmatrix} \leq \begin{bmatrix} u^{-1} \\ u^{-1} \end{bmatrix}, \tag{6.2}$$

where $|A|$ is the matrix with each entry the modulus of the corresponding entry of A and $u \leq 0.5$. This corresponds to

$$|a_{kk}^{(k)}|^{-1} \max_i |a_{ik}^{(k)}| \ \leq \ u^{-1} \tag{6.3}$$

for the 1×1 case.

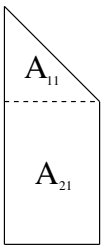

Figure 6.1: Frontal matrix showing fully summed columns.

If we look at the frontal matrix shown in Figure 6.1 then pivots can only be chosen from A_{11}. Various pivoting strategies can be adopted, some of which we now discuss.

6.1 Threshold partial pivoting

We can use the threshold partial pivoting (TPP) algorithm defined by equations (6.1) to (6.3). The problem is that, in Figure 6.1, entries in A_{21} may be large enough to prevent a potential pivot from A_{11} satisfying the threshold test (6.1). In a normal dense matrix factorization such large entries could be used as the off-diagonal entry in a 2×2 block pivot but that is not possible in this context since we do not have available all the entries in the rows of the matrix corresponding to rows in A_{21} as they need contributions from further up the tree. Pivots (with associated row and column) that cannot be used and are passed to ancestors in the tree are called delayed pivots.

We are thus faced with a choice of just restricting pivoting to A_{11} and hoping that large entries in A_{21} do not cause us problems. This results in an unstable algorithm so normally some precaution is taken to guard against that. Thus it is common to do some steps of iterative refinement that will either give an accurate answer or will alert the user to the fact that there are problems with the factorization. This is the approach adopted by PARDISO [23].

The alternative is to conduct the elimination column by column and, if a column has too large an entry in the part of the column in A_{21}, to avoid pivoting on that column and to leave it in the uneliminated Schur complement or contribution block for passing to the ancestors in the tree. This does not compromise our threshold pivoting algorithm since the said column will eventually be available for pivoting, certainly at the root node if not before. The problem with this approach is that it does not lend itself to good exploitation of parallelism. We are working a column at a time and need to update later columns before we can test them for pivoting. We must search for the maximum entry in the column that requires access to memory and potentially memory on different processors in a distributed memory environment. However, this is usually the algorithm of choice for serial codes and we can obtain some degree of parallelism by careful implementation. This is the approach adopted by the HSL code `HSL_MA97` [20].

In fact, for many indefinite systems, if the matrix is scaled well beforehand and the threshold parameter is not set too high, there are few instances when a large entry in A_{21} prevents a pivot being chosen [21]. We discuss an algorithm that exploits this in the next section.

6.2 A posteriori threshold pivoting

The intention of a posteriori threshold pivoting (APTP) [18] is to concentrate on obtaining good parallelism in the case when no pivots are delayed but to be able to continue with a stable factorization when there are delayed pivots albeit with a slight loss in efficiency.

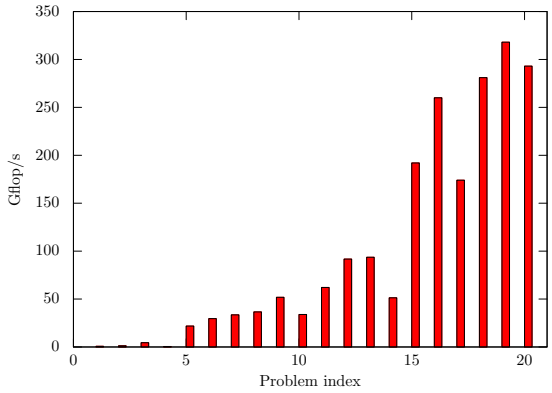

Figure 6.2: Performance of SSIDS factorization for easy indefinite matrices on the 28-node Haswell machine.

In this approach we first tile the frontal matrix and compute the factors for a block of columns in A_{11} working only within this block. We then use these factors to update in parallel tiles in A_{21} within this block of columns. When doing this, we monitor the size of the entries of L that we are creating and check that their absolute value is less than u^{-1}. If that is so then we use the blocks to update the rest of the frontal matrix. If it is not, then we do not trust any columns in the block to the right of the failed entry and we backtrack to the situation at that time, flag the column as being non-pivotal and try to continue the factorization. The main penalty here is that we must also store the unmodified data so that a backtrack can be performed. At worst, the failed columns will correspond to delayed pivots in the TPP algorithm and will be passed with the Schur complement to ancestor nodes.

This algorithm is used in the SSIDS code [18] and the kernel that implements it is used in the NLAFET code SpLDLT. We illustrate its performance on a set of matrices of increasing size in Figure 6.2 on our 28-node Haswell machine. The attributes of the test matrices are shown in Table A.2 in the Appendix and they are called *easy* indefinite because they do not have any delayed pivots if the matrices are scaled beforehand. On the Haswell machine, the `GEMM` kernel runs at 768 Gflop/s so that, even with pivoting, we are close to achieving our target of half the asymptotic speed.

6.3 Numerical pivoting in the indefinite case

We have discussed three numerical pivoting options for symmetric indefinite factorizations and have identified codes that implement each of these. In this section we run some tests on some highly indefinite systems to see how these strategies compare in practice. Some of the results are from the technical report by Hogg [18]. The codes that we are comparing are: SSIDS using APTP, `HSL_MA97` that uses TPP, and PARDISO as implemented in MKL 11.0.3 that factorizes the block A_{11} without any test against the entries in A_{21}. For these tests we use the set of indefinite matrices from Hogg and Scott [21] that require substantial numerical pivoting identified by having a large number of delayed pivots. Many of these hard indefinite cases are saddle-point matrices. We list these matrices in Table A.3 in the Appendix. We show the comparison in Figure 6.3. We see in this figure that both SSIDS and PARDISO obtain much better speedups than the TPP code `HSL_MA97` and they are roughly comparable, with PARDISO having a slight edge over SSIDS.

However, as we mentioned when discussing the different numerical pivoting strategies, the one used by PARDISO is unstable and iterative refinement is needed. We examine, in more detail in Table 6.1, the performance of the three codes on some matrices from the earlier runs. Here we see that while both SSIDS and PARDISO are significantly faster than `HSL_MA97`, SSIDS has a far better backward error than PARDISO, even though iterative refinement is used with PARDISO. Indeed SSIDS is comparable with the numerical performance of the TPP code as it should be.

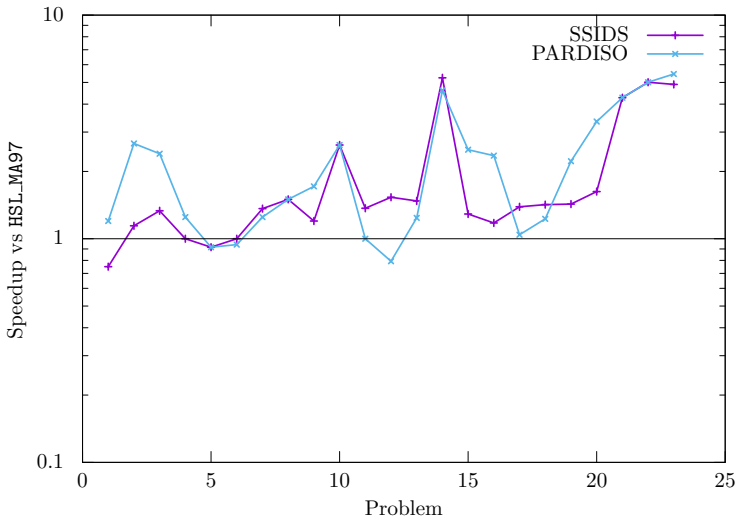

Figure 6.3: Comparison of codes on hard indefinite matrices on the 28-core Haswell machine.

Matrix	stokes128	cvxqp3	ncvxqp7
Order $\times 10^3$	49.7	17.5	87.5
Entries $\times 10^6$	0.30	0.07	0.31
	Factor time		
HSL_MA97	0.15	1.52	8.18
PARDISO	0.12	0.33	1.50
SSIDS V2	0.11	0.29	1.67
	Backward error		
HSL_MA97	$1.6\ 10^{-15}$	$3.1\ 10^{-11}$	$4.4\ 10^{-9}$
PARDISO	$3.9\ 10^{-3}$	$1.1\ 10^{-6}$	$1.4\ 10^{-7}$
SSIDS V2	$1.4\ 10^{-15}$	$2.0\ 10^{-11}$	$7.3\ 10^{-9}$

Table 6.1: Hard indefinite systems on the 28-core Haswell machine.

In our talks on these approaches, we parodize these codes as: with HSL_MA97 we have to *pay*, with PARDISO we have to *pray*, while with SSIDS we can *play*.

7 Task 3.3 Direct methods for highly unsymmetric systems

The algorithms that we will study for highly unsymmetric systems are radically different from those of the previous sections. They are not based on an assembly tree but rather on a right-looking factorization method that uses a Markowitz threshold strategy. Parallelism is obtained largely through the use of extensive blocking.

We define a highly unsymmetric matrix as a matrix whose structure is not well approximated by the structure of $|A| + |A|^T$. Various authors have defined a measure of the asymmetry of a matrix and here we use that defined in [17] which is the proportion of off-diagonal entries for which there is a corresponding entry in the transpose, viz.

$$si(A) = \frac{number_{i \neq j}\{a_{ij} * a_{ji} \neq 0\}}{nz\{A\}},$$

where si is called the symmetry index and $nz\{A\}$ is the number of off-diagonal entries in the matrix A. A symmetric matrix will thus have a symmetry index of 1.0. Matrices with symmetry indices of less than 0.9 can be considered highly unsymmetric and these are the main target of this work in Task 3.3. Such matrices are encountered in applications such as: chemical engineering, linear programming, economic modelling, power systems, and circuit modelling.

Both code and matrices implementing these algorithms can be very unstructured and complicated. We illustrate this in Figure 7.1 with a matrix from an econometric model of SE Asia. It is matrix `ORANI_678`, from the Harwell-Boeing test set and available from the SuiteSparse matrix collection [9].

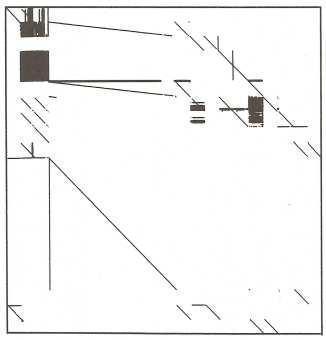

Figure 7.1: Matrix from econometric model of SE Asia.

As an example of how convoluted the coding can be we reproduce the inner-loop from an 1993 HSL code, `MA28`. For many years this was best code available for this type of matrix and was the code available in netlib. The innermost loop in the Fortran 77 version of the HSL code `MA48` [14] that replaced this was only marginally less ugly.

7.1 Markowitz threshold pivoting

Clearly, for any pivot in Gaussian elimination the maximum fill-in[2] that can occur is the product of the number of other entries in the pivot row with the number of other entries in the pivot column. Thus if there are c_j entries in column j and r_i entries in row i, then we define the Markowitz cost for a potential pivot in row i, column j as

$$Mark_{ij} \;=\; (r_i - 1) \times (c_j - 1). \tag{7.1}$$

We choose candidate entries with low or minimum Markowitz count to reduce the amount of fill-in. Of course, such a candidate would be unacceptable if its value was zero or very small relative to other entries.

[2] An entry that is zero in A but is nonzero in the corresponding entry of the factors is termed fill-in.

```
      DO 590 JJ = J1, J2
         J = ICN (JJ)
         IF (IQ(J). GT.0) GO TO 590
         IOP = IOP + 1
         PIVROW = IJPOS - IQ (J)
         A(JJ) = A(JJ) + AU x A (PIVROW)
         IF (LBIG) BIG = DMAXI (DABS(A(JJ)), BIG)
         IF (DABS(A(JJ)). LT. TOL) IDROP = IDROP + 1
         ICN (PIVROW) = ICN (PIVROW)
590 CONTINUE
```

Figure 7.2: Innermost loop of MA28.

We therefore introduce a pivot threshold and, analogously to the case for symmetric matrices in equation (6.1), only consider entries a_{ij} that satisfy

$$|a_{ij}| \geq u * \max_k |a_{kj}|, \quad k = 1, \ldots, m \tag{7.2}$$

where u is a threshold parameter $0 < u \leq 1.0$. That is to say we only consider entries that are at least u times as large as the largest entry in modulus of all entries in the column. We call such entries eligible entries. If u were equal to 1.0 then we would be using partial pivoting that is the most common algorithm for dense matrices.

To continue with the factorization we must first update the remaining matrix using the outer product of the pivot row and column, updating the numerical entries and normally introducing fill-in. This is clearly a right-looking algorithm. For selecting the next pivot we perform the threshold Markowitz algorithm on this remaining updated matrix of order one less than the previous one, and we continue in this way until all $\min(m, n)$ pivots have been chosen although it is advantageous to switch to a dense code when the Schur complement becomes denser [14]. The algorithm is simple but the data structures to implement it efficiently, even in serial mode, are not.

7.2 Parallel implementation of threshold Markowitz pivoting

For our parallel implementation, we essentially use the same algorithm, that is a threshold Markowitz algorithm using the same terminology as the previous section. In this implementation, we find a set of independent pivots that can be used and then use these as a block pivot to update the remaining matrix in parallel. We illustrate this in Figure 7.3.

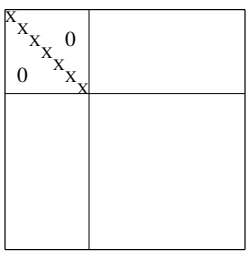

Figure 7.3: Block of independent pivots.

When choosing the set of pivots in parallel, we note that the threshold test only needs information from one column and this is our first observation for obtaining significant parallelism. We thus launch our algorithm by scanning columns of the matrix independently. For each column we compute the largest

entry in modulus and choose as a potential pivot in that column an entry that satisfies the threshold test, that is an entry at least u times the maximum just calculated, and is in a row with the least number of entries over all eligible entries in the column. It is possible that all eligible entries are in rows of high count so, although we still flag the entry as a possible pivot, we will not use it unless there are no other suitable pivots.

Having done this, we then want to construct a set of independent pivots in parallel. We do this by a binary combination illustrated in Figure 7.4. We use a parameter to define a block size and then select

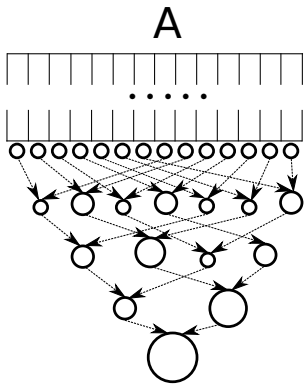

Figure 7.4: Combining pivots to get block pivot.

at random a set of columns of the matrix whose cardinality is the block size. We choose the columns at random because often the structure of the matrix militates against choosing consecutive columns. We then do a cheap scan of columns in the block to identify an independent set. Each block is independent and can be scanned in parallel. If we assume that the column we are seeking to combine with the current block pivot is j_1 and the block pivot is in the set of rows I_2 and columns J_2 and that the potential pivot in column j_1 is in row i_1 then the column is combined with the current block if there are no entries in positions (I_2, j_1) and (i_1, J_2). For checking whether a column yields an independent pivot we use an integer array of length n that flags whether a row has no entries in all the previously chosen columns. A similar flag is set for the columns so that the test comprises just two lookups followed by an update of the flags. This can be done without having to reset the flag array by incrementing flags at each step and only resetting if integer overflow occurs. We show experiments on the effect of the influence of the block size in the next section.

Having done this pass on all the blocks, we have sets of independent pivots of size up to the block size. We then combine these to get larger sets, continuing to do so in a binary tree fashion as shown in Figure 7.4 until we have a single set of independent pivots.

In sequential codes like `MA48` [14], we select the eligible pivot that has the minimum Markowitz count, as defined in equation (7.1). Because we want to get large blocks of independent pivots, we relax this by accepting eligible pivots within a factor of the minimum, that is an entry (i, j) can be chosen as a pivot if its Markowitz cost satisfies the condition

$$Mark_{ij} \leq \alpha_{Mark} \times Best_{Mark} \tag{7.3}$$

where the Markowitz factor α_{Mark} is greater than or equal to one and $Best_{Mark}$ is the lowest Markowitz count.

Our next step is to perform all the pivot operations for the block of independent pivots in parallel. In effect what we have to do is a parallel sparse matrix by sparse matrix multiply to update the Schur complement. Having done this, we then repeat the parallel pivot selection on the reduced matrix corresponding to the Schur complement that we have just computed.

We terminate the algorithm when either the last few steps of our algorithm ("few" is a parameter that we have set to 5 in our experiments) have failed to obtain a number of independent pivots greater than

a preset threshold or the Schur complement reaches a preset density. At that stage, in the present code, we switch to using the `PLASMA` [7] code `GETRF` for parallel dense LU factorization on the remaining Schur complement. We plan in later versions of the code to have a transitional stage where we use a parallel sparse direct code designed for relatively dense sparse matrices, such as the parallel LU factorization that will be developed in Task 3.2.

Because of these various stages, we use four different data structures. The L and U factors continuously grow with the execution of our algorithm without changing the already computed part. For this reason, we use standard CSC storage for the L and U factors. The values in the diagonal matrix D of the LDU factorization are stored in a separate array. On the other hand, the structure of the Schur complement changes because of the update operations. Additionally, at each step we must be able to determine if a set of pivots are mutually independent. For these reasons, we use a flexible CSC/CSR based structure. The numerical values are stored in the CSC fashion, while the CSR part stores only the nonzero structure of the matrix. In total, three large arrays are used, one index and one value array for the CSC part and one index array for the CSR part. Having the matrix structure stored by rows and columns provides an efficient way of updating the structures during pivot selection and Schur update. In order to cope with the dynamic nature of the Schur complement, within the CSR/CSC structure extra space is allocated at the end of each row and each column. Each row and column is represented by an offset from the start of the corresponding array, by the number of entries each contains, with the amount of available free space. Additional memory is allocated at the end of each array which is managed by a garbage collector. For each block of free memory, the garbage collector stores the offset from the beginning of the array and its size. When fill-in to a row or a column consumes its available space, it is moved to the next available space provided by the garbage collector. Its old memory is marked as free memory and added to the garbage collector for future reuse. Similarly, space freed when a row or column becomes pivotal is likewise given to the garbage collector.

7.3 Preliminary results

In this section we present results of some experiments with our solver. The tests were performed on the same multicore Haswell machine that we used for the experiments in Section 5. All the results presented are sequential (mono-threaded execution). The main attributes of the matrices used in this study are given in Table 7.1. The matrix twotone is from the SparseSuite set of test matrices and the other two are from a Power Systems application supplied by Bernd Klöss of DigSILENT GmbH.

Matrix	Order $\times 10^3$	Entries $\times 10^6$	si
twotone	120	1.22	0.26
LoadFlow_Newton_0_4	197	3.70	0.46
Jacobian_unbalancedLdf	203	2.76	0.80

Table 7.1: Some highly unsymmetric matrices.

As discussed in the previous section, at some point in our algorithm we switch to a dense solver. The reason for this is that once the Schur complement becomes too dense, we get only a few pivots at a time and in addition the operations become more and more expensive. At some point, we get sets of size one only and the execution of each step takes a lot of time. This is considerably improved by switching to the dense solver and we show the number of pivots selected and the time in Figure 7.5.

At the beginning of each step, we need to set up the initial sets of pivots that will be merged later. The impact of the number of candidates per initial pivot set (the block size) on the average number of pivots found per step is given in Figure 7.6. We can see that for all the values of this parameter, our algorithm is able to create large enough sets. One interesting thing to notice is that when the LoadFlow_Newton_0_4 matrix is used, the optimal value is 10. That points to the fact that the largest

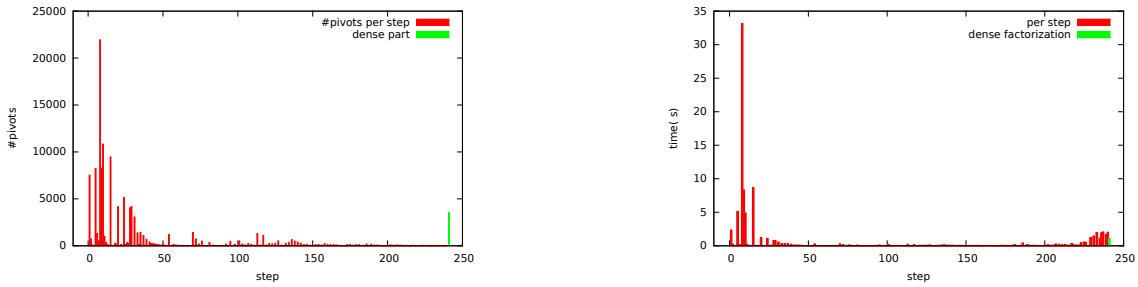

Figure 7.5: The number of pivots (left) and the time spent (right) for each step when the matrix twotone is used. Additionally, the number of pivots handled by the dense solver and the time spent in the dense solver are given at the end.

pivot sets are obtained when we start with a large number of small pivot sets. Since the merge is done using a binary tree and, at each level of the tree, all the merging can be done in parallel, it indicates that our algorithm could potentially be extremely parallel.

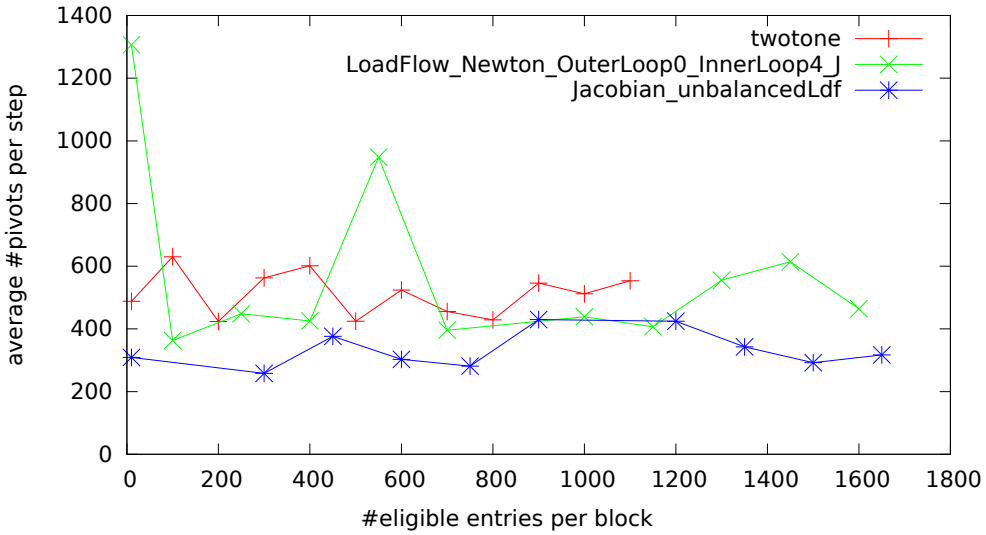

Figure 7.6: Impact of the number of candidates per initial pivot set on the average number of pivots per step.

Each pivot candidate must satisfy the Markowitz test in Equation (7.3). When we relax the constraint on the Markowitz cost, we accept pivots with higher Markowitz cost which will usually introduce more fill-in in the factors. The impact of the Markowitz factor α_{Mark} on the number of entries in the L and U factors is presented in Figure 7.7. In this figure the total number of entries in the L and U factors are shown and include the dense part for each of them. Thus as expected when the Markowitz factor is relaxed, the amount of fill-in increases.

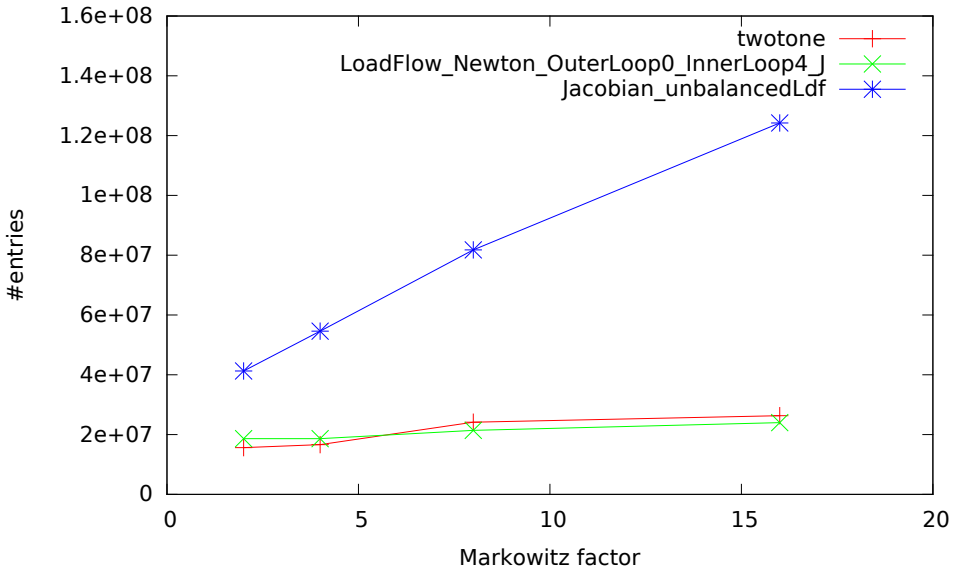

Figure 7.7: The impact of the Markowitz factor on the total number of entries in the L and U factors.

8 Task 3.4 Hybrid direct-iterative methods

We have only just commenced work on Task 3.4. Here we consider the use of a hybrid method to solve the system of equations. This enables us to extend the range of direct solvers to larger matrices and to obtain another level of parallelism. The system we want to solve is

$$Ax = b, \tag{8.1}$$

where A is an $m \times n$ sparse matrix, x is an n-vector and b is an m-vector. In the following, we assume the system is consistent and for simplicity we suppose that A has full rank.

We will study the solution of the system (8.1) using the block Cimmino method, an iterative method using block-row projections. In this method, the system (8.1) is subdivided into strips of rows as in the following:

$$\begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix} x = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix}. \tag{8.2}$$

Let $P_{\mathcal{R}(A_i^T)}$ be the projector onto the range of A_i^T and ${A_i}^+$ be the Moore-Penrose pseudo-inverse of the partition A_i. The block Cimmino algorithm then computes a solution iteratively from an initial estimate $x^{(0)}$ according to:

$$u_i = A_i^+ \left(b_i - A_i x^{(k)} \right) \quad i = 1, ..., p \tag{8.3}$$

$$x^{(k+1)} = x^{(k)} + \omega \sum_{i=1}^{p} u_i, \tag{8.4}$$

where ω is a real parameter whose value we will shortly show to be immaterial. We note the independence of the set of p equations, which is why the method is so attractive in a parallel environment. The block Cimmino method is described in more detail by Ruiz [22].

Although the matrix in equation (8.1) can be rectangular and the Cimmino method can work on such systems [16], for our main discussion we will assume that A is square and of order n.

With the above notations, the iteration equations are thus:

$$\begin{aligned} x^{(k+1)} &= x^{(k)} + \omega \sum_{i=1}^{p} A_i^+ \left(b_i - A_i x^{(k)} \right) \\ &= \left(I - \omega \sum_{i=1}^{p} A_i^+ A_i \right) x^{(k)} + \omega \sum_{i=1}^{p} A_i^+ b_i \\ &= Q x^{(k)} + \omega \sum_{i=1}^{p} A_i^+ b_i. \end{aligned}$$

The iteration matrix for block Cimmino $H = I - Q$ is then a sum of projectors $H = \omega \sum_{i=1}^{p} \mathcal{P}_{\mathcal{R}(A_i^T)}$. It is thus symmetric and positive definite and so we can solve

$$Hx = \xi, \tag{8.5}$$

where $\xi = \omega \sum_{i=1}^{p} A_i^+ b_i$, using conjugate gradient or block conjugate gradient methods. As ω appears on both sides of equation (8.5), we can set it to one.

The starting point for our work on this is the thesis and code of Mohammed Zenadi [24]. We have been porting his code that uses MPI and multi-threading to our machines and have been resolving a few issues. We have been studying the resulting code and have obtained good parallelism for some problems. We show the performance of the code on matrix `cage12` from the SuiteSparse test set in Figure 8.1. The machine used for these runs is a local heterogeneous machine called scarf. It has a number of Intel nodes (including some E5-2650, E2660, X5675, X5530, and E5530 nodes)[3]. The actual configuration is determined by the batch scheduler at runtime.

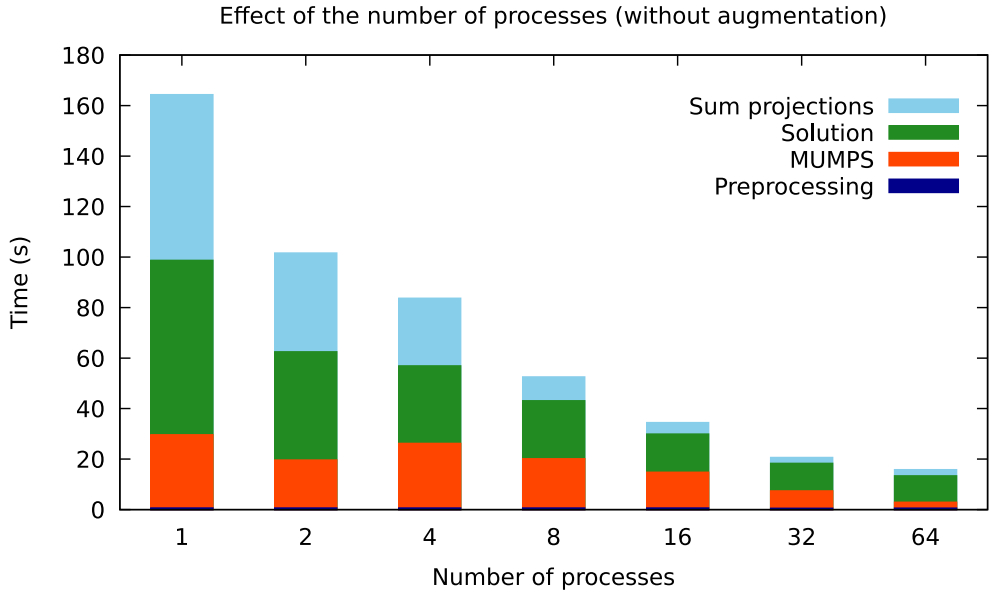

Figure 8.1: The speedup of block Cimmino for `cage12` on the scarf machine at RAL.

The default partitioning method used in the code is PaToH [8]. In the NLAFET project we are now testing various partitioning approaches since the number of block Cimmino iterations is strongly related to the number of columns in the border of the bordered block diagonal form that we illustrated in Figure 4.1. For the matrix `cage12`, we see in Figure 8.2(left) that the number of iterations is not affected by the number of processes and that it does not vary significantly when the number of partitions increases (see Figure 8.2(right)) although, as expected, the number of iterations increases with the number of partitions.

[3]See `http://www.scarf.rl.ac.uk/hardware`

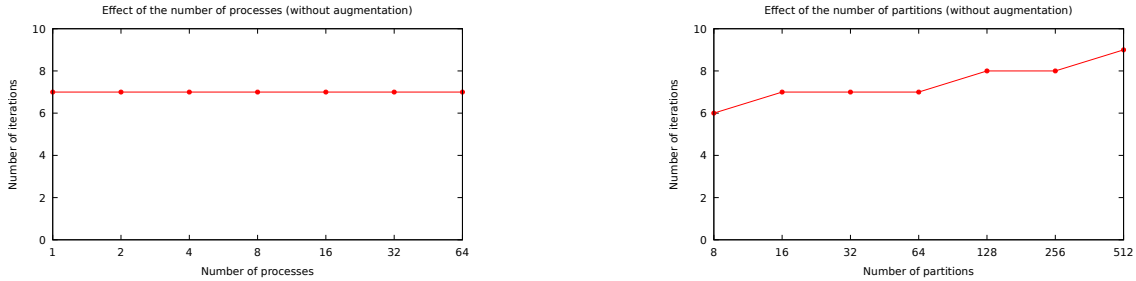

Figure 8.2: The effect of the number of processes (left) and number of partitions (right) on the convergence of block Cimmino on a system with matrix `cage12`.

We note that in the code of Zenadi and in the earlier research [11] we have developed an algorithm ABCD (Augmented Block Cimmino Distributed) that totally avoids the interaction between blocks by augmenting them so that they are mutually orthogonal. This yields a pseudo-direct method that should converge in one iteration. We will also be developing this approach further in the NLAFET project.

9 Concluding remarks

In conclusion, we emphasize that this work is very much still ongoing and we are only just past the half-way point in the project.

What we can say already is that there is lots of parallelism in sparse direct solvers but programming this, while interesting and fun, is extremely tough.

Much of the work that we have described has been done to provide a solid platform for future work in the NLAFET project. We are currently examining in detail the two main approaches for using assembly trees in sparse factorization, namely the supernodal and the multifrontal method. While the data handling is much simpler in multifrontal schemes and exploitation of parallelism should thus be easier, the storage requirements are often higher. Our future work will quantify these issues and give recommendations concerning these two approaches.

One of the reasons why we have used runtime systems is so that our codes can be more easily ported to different architectures and we plan to illustrate this by porting our codes to GPUs and to a heterogeneous system with multicore nodes and GPUs. The authors of the runtime systems are developing versions for distributed memory environments that we look forward to testing.

One approach for solving systems with highly unsymmetric matrices is to permute them to make them less unsymmetric and this will be used with the *LU* codes that we will develop as an extension to the tree-based methods for symmetric systems. This approach will then be compared with our threshold Markowitz approach. Finally, the current block Cimmino code uses MUMPS for the direct solution of subproblems. The MUMPS code is arguably one of the best parallel direct solvers, but it is based on MPI. In the context of our hierarchical approach to exploiting parallelism in this hybrid method, we will be targeting a multicore environment at the direct solver level for which our new codes being developed in Task 3.2 might be better suited.

10 Acknowledgements

This work is supported by the NLAFET Project funded by the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement 671633. We would like to thank Philippe Gambron for doing the experiments reported in Section 8, Jonathan Hogg for his earlier work in the project developing the first versions of some of the kernels used in Sections 5 and 6, and Tim Davis (Texas A&M) for discussions

on parallel Markowitz. We also thank Jennifer Scott and Bo Kågström (Umeå) for their comments on a draft of this manuscript.

References

[1] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, *Multifrontal QR factorization for multicore architectures over runtime systems*, in Proceedings of Euro-Par 2013 Parallel Processing, Springer-Verlag, 2013, pp. 521–532.

[2] ——, *Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems*, ACM Trans. Math. Softw., 43 (2016), pp. Article 13, 17 pages.

[3] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent, *Multifrontal parallel distributed symmetric and unsymmetric solvers*, Comput. Methods in Appl. Mech. Eng., 184 (2000), pp. 501–520.

[4] J. R. Bunch, L. Kaufman, and B. N. Parlett, *Decomposition of a symmetric matrix*, Numerische Mathematik, 27 (1976), pp. 95–110.

[5] J. R. Bunch and B. N. Parlett, *Direct methods for solving symmetric indefinite systems of linear equations*, SIAM Journal on Numerical Analysis, 8 (1971), pp. 639–655.

[6] A. Buttari, *Fine-grained multithreading for the multifrontal QR factorization of sparse matrices*, SIAM Journal on Scientific Computing, 35 (2013), pp. C323–C345.

[7] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Parallel Computing, 35 (2009), pp. 38–53.

[8] Ü. V. Çatalyürek and C. Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Transactions on Parallel and Distributed Systems, 10 (1999), pp. 673–693.

[9] T. A. Davis and Y. Hu, *The university of Florida sparse matrix collection*, ACM Trans. Math. Softw., 38 (2011), pp. 1:1–1:25.

[10] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices. Second Edition.*, Oxford University Press, Oxford, England, 2016.

[11] I. S. Duff, R. Guivarch, D. Ruiz, and M. Zenadi, *The augmented block Cimmino distributed method*, SIAM Journal on Scientific Computing, 37 (2015), pp. A1248–A1269.

[12] I. S. Duff, J. Hogg, and F. Lopez, *Experiments with sparse Cholesky using a sequential task-flow implementation*, Tech. Rep. RAL-TR-2016-016, Rutherford Appleton Laboratory, Oxfordshire, England, 2016. NLAFET Working Note 7. Submitted to NACO.

[13] I. S. Duff and F. Lopez, *Experiments with sparse Cholesky using a parametrized task graph implementation*, Tech. Rep. RAL-TR-2017-006, Rutherford Appleton Laboratory, Oxfordshire, England, 2017. NLAFET Working Note 14. Accepted for presentation at PPAM 2017.

[14] I. S. Duff and J. K. Reid, *The design of MA48, a code for the direct solution of sparse unsymmetric linear systems of equations*, ACM Trans. Math. Softw., 22 (1996), pp. 187–226.

[15] ——, *Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems*, ACM Trans. Math. Softw., 22 (1996), pp. 227–257.

[16] T. Elfving, *Block-iterative methods for consistent and inconsistent linear equations*, Numerische Mathematik, 35 (1980), pp. 1–12.

[17] A. M. Erisman, R. G. Grimes, J. G. Lewis, W. G. Poole Jr., and H. D. Simon, *Evaluation of orderings for unsymmetric sparse matrices*, SIAM J. Scientific and Statistical Computing, 7 (1987), pp. 600–624.

[18] J. Hogg, *A new sparse LDLT solver using a posteriori threshold pivoting*, Technical Report RAL-TR-2016-017, Rutherford Appleton Laboratory, Oxfordshire, England, 2017. NLAFET Working Note 6.

[19] J. Hogg, J. Reid, and J. Scott, *Design of a multicore sparse Cholesky factorization using DAGs*, SIAM Journal on Scientific Computing, 32 (2010), pp. 3627–3649.

[20] J. Hogg and J. Scott, *HSL_MA97 : a bit-compatible multifrontal code for sparse symmetric systems*, Technical Report RAL-TR-2011-024, Rutherford Appleton Laboratory, Oxfordshire, England, 2011.

[21] ——, *A study of pivoting strategies for tough sparse indefinite systems*, ACM Trans. Math. Softw., 40 (2013), pp. Article 4, 19 pages.

[22] D. F. Ruiz, *Solution of large sparse unsymmetric linear systems with a block iterative method in a multiprocessor environment*, PhD Thesis, Institut National Polytechnique de Toulouse, 1992. CERFACS Technical Report, TH/PA/92/06.

[23] O. Schenk and K. Gärtner, *On fast factorization pivoting methods for sparse symmetric indefinite systems*, Electronic Transactions on Numerical Analysis, 23 (2006), pp. 158–179.

[24] M. Zenadi, *The solution of large sparse linear systems on parallel computers using a hybrid implementation of the block Cimmino method.*, Thése de Doctorat, Institut National Polytechnique de Toulouse, Toulouse, France, dcembre 2013.

11 Appendix. Test problems

#	Problem	n (10^3)	$nz(A)$ (10^6)	$nz(L)$ (10^6)	*Flops* (10^9)	Application/Description
1	Schmid/thermal2	1228	4.9	51.6	14.6	Unstructured thermal FEM
2	Rothberg/gearbox	154	4.6	37.1	20.6	Aircraft flap actuator
3	DNVS/m_t1	97.6	4.9	34.2	21.9	Tubular joint
4	Boeing/pwtk	218	5.9	48.6	22.4	Pressurised wind tunnel
5	Chen/pkustk13	94.9	3.4	30.4	25.9	Machine element
6	GHS_psdef/crankseg_1	52.8	5.3	33.4	32.3	Linear static analysis
7	Rothberg/cfd2	123	1.6	38.3	32.7	CFD pressure matrix
8	DNVS/thread	29.7	2.2	24.1	34.9	Threaded connector
9	DNVS/shipsec8	115	3.4	35.9	38.1	Ship section
10	DNVS/shipsec1	141	4.0	39.4	38.1	Ship section
11	GHS_psdef/crankseg_2	63.8	7.1	43.8	46.7	Linear static analysis
12	DNVS/fcondp2	202	5.7	52.0	48.2	Oil production platform
13	Schenk_AFE/af_shell3	505	9.0	93.6	52.2	Sheet metal forming
14	DNVS/troll	214	6.1	64.2	55.9	Structural analysis
15	AMD/G3_circuit	1586	4.6	97.8	57.0	Circuit simulation
16	GHS_psdef/bmwcra_1	149	5.4	69.8	60.8	Automotive crankshaft
17	DNVS/halfb	225	6.3	65.9	70.4	Half-breadth barge
18	Um/2cubes_sphere	102	0.9	45.0	74.9	Electromagnetics
19	GHS_psdef/ldoor	952	23.7	144.6	78.3	Large door
20	DNVS/ship_003	122	4.1	60.2	81.0	Ship structure
21	DNVS/fullb	199	6.0	74.5	100.2	Full-breadth barge
22	GHS_psdef/inline_1	504	18.7	172.9	144.4	Inline skater
23	Chen/pkustk14	152	7.5	106.8	146.4	Tall building
24	GHS_psdef/apache2	715	2.8	134.7	174.3	3D structural problem
25	Koutsovasilis/F1	344	13.6	173.7	218.8	AUDI engine crankshaft
26	Oberwolfach/boneS10	915	28.2	278.0	281.6	Bone micro-FEM
27	ND/nd12k	36.0	7.1	116.5	505.0	3D mesh problem
28	ND/nd24k	72.0	14.4	321.6	2054.4	3D mesh problem
29	Janna/Flan_1565	1565	59.5	1477.9	3859.8	3D mechanical problem
30	Oberwolfach/bone010	987	36.3	1076.4	3876.2	Bone micro-FEM
31	Janna/StocF-1465	1465	11.2	1126.1	4386.6	Underground aquifer
32	GHS_psdef/audikw_1	944	39.3	1242.3	5804.1	Automotive crankshaft
33	Janna/Fault_639	639	14.6	1144.7	8283.9	Gas reservoir
34	Janna/Hook_1498	1498	31.2	1532.9	8891.3	Steel hook
35	Janna/Emilia_923	923	21.0	1729.9	13661.1	Gas reservoir
36	Janna/Geo_1438	1438	32.3	2467.4	18058.1	Underground deformation
37	Janna/Serena	1391	33.0	2761.7	30048.9	Gas reservoir

Table A.1: Test matrices and their characteristics. n is the matrix order, $nz(A)$ the number entries in the matrix A, $nz(L)$ the number of entries in the factor L, and *Flops* corresponds to the operation count for the matrix factorization.

Problem	n $\times 10^3$	$nz(A)$ $\times 10^6$	$nz(L)$ $\times 10^6$	$flops$ $\times 10^9$
Oberwolfach/t2dal	4.26	0.02	0.28	0.02
GHS_indef/dixmaanl	60.00	0.18	1.58	0.05
Oberwolfach/rail_79841	79.84	0.32	4.43	0.33
GHS_indef/dawson5	51.54	0.53	5.69	0.90
Boeing/bcsstk39	46.77	1.07	9.61	2.66
Boeing/pct20stif	52.33	1.38	12.60	5.63
GHS_indef/copter2	55.48	0.41	12.70	6.10
GHS_indef/helm2d03	392.26	1.57	33.00	6.16
Boeing/crystk03	24.70	0.89	10.90	6.26
Oberwolfach/filter3D	106.44	1.41	23.80	8.71
Koutsovasilis/F2	71.50	2.68	23.70	11.30
McRae/ecology1	1000.00	3.00	72.30	18.20
Cunningham/qa8fk	66.13	0.86	26.70	22.10
Oberwolfach/gas_sensor	66.92	0.89	27.00	22.10
Oberwolfach/t3dh	79.17	2.22	50.60	70.10
Lin/Lin	256.00	1.01	126.00	285.00
GHS_indef/sparsine	50.00	0.80	207.00	1390.00
PaRSEC/Ge99H100	112.98	4.28	669.00	7070.00
PaRSEC/Ga10As10H30	113.08	3.11	690.00	7280.00
PaRSEC/Ga19As19H42	133.12	4.51	823.00	9100.00

Table A.2: Easy Indefinite. Statistics as reported by the analyse phase of SSIDS with default settings, assuming no delayed pivots.

Problem	n $\times 10^3$	$nz(A)$ $\times 10^6$	$nz(L)$ $\times 10^6$	$flops$ $\times 10^9$
TSOPF/TSOPF_FS_b39_c7	28.22	0.37	2.61	0.26
TSOPF/TSOPF_FS_b162_c1	10.80	0.31	1.89	0.36
QY/case39	40.22	0.53	3.87	0.40
TSOPF/TSOPF_FS_b39_c19	76.22	1.00	7.28	0.75
TSOPF/TSOPF_FS_b39_c30	120.22	1.58	11.10	1.10
GHS_indef/cont-201	80.59	0.24	7.12	1.11
GHS_indef/stokes128	49.67	0.30	6.35	1.16
TSOPF/TSOPF_FS_b162_c3	30.80	0.90	6.37	1.41
TSOPF/TSOPF_FS_b162_c4	40.80	1.20	7.32	1.43
GHS_indef/ncvxqp1	12.11	0.04	3.56	2.52
GHS_indef/darcy003	389.87	1.17	23.20	3.01
GHS_indef/cont-300	180.90	0.54	17.20	3.58
GHS_indef/bratu3d	27.79	0.09	7.49	4.72
GHS_indef/cvxqp3	17.50	0.07	6.33	5.27
TSOPF/TSOPF_FS_b300	29.21	2.20	13.40	6.92
TSOPF/TSOPF_FS_b300_c1	29.21	2.20	13.50	7.01
GHS_indef/d_pretok	182.73	0.89	24.80	7.42
GHS_indef/turon_m	189.92	0.91	24.70	7.60
TSOPF/TSOPF_FS_b300_c2	56.81	4.39	27.00	14.10
TSOPF/TSOPF_FS_b300_c3	84.41	6.58	40.50	21.40
GHS_indef/ncvxqp5	62.50	0.24	22.90	24.30
GHS_indef/ncvxqp3	75.00	0.27	39.30	63.70
GHS_indef/ncvxqp7	87.50	0.31	51.00	101.00

Table A.3: Hard Indefinite. Statistics as reported by the analyse phase of SSIDS with default settings, using matching-based ordering, assuming no delayed pivots.