Chapter 6

Schur complement method

6.1 Introduction

Schur complement (or Dual Schur Decomposition) [190, 191, 192] is a direct parallel method, based on the use of non-overlapping subdomains with implicit treatment of interface conditions. Its main feature is that each processor has to solve only twice its own subdomain plus an interface problem to obtain the exact solution of the subdomain, with just one global communication operation. Compared with multigrid or Krylov subspace solvers, it has received less attention. However, it seems a very promising technique. A description of the algorithm is given and our implementation is benchmarked.

In the approach used for our implementation, the two main considerations are: *(i)* The role of Schur complement method is not to be used as a solver for the global domain but as an auxiliary procedure for a smaller level of a MG algorithm and *(ii)* Many right-hand-side terms are to be solved with the same matrix A (as for instance, for the pressure-correction equations of constant physical properties CFD problems, as discussed in section 1.2, so it is worth investing effort in a pre-processing stage of the algorithm, that depends only on A. Even more, the data generated in the preprocessing stage can be saved to be reused in other executions with the same mesh. These ideas are fully exploited in our implementation.

6.2 Algorithm

6.2.1 Reordering

The linear equation system to be solved is denoted as:

$$Ax = b \tag{6.1}$$

The unknowns in vector x are partitioned into a family of P subsets, one per processor, plus one *interface*, labeled s. The interface s is defined so that for any pair of subsets p_1, p_2, no unknown of p_1 is directly coupled with any unknown of p_2, but only to its own unknowns and to s:

$$\text{if } i \in x_{p_1} \text{ and } j \in x_{p_2} \text{ then } A_{i,j} = 0 \tag{6.2}$$

According to this decomposition, unknowns in vector x are reordered as :

$$x = [x_0, x_1, \cdots, x_{P-1}, x_s]^t \tag{6.3}$$

Inside each subset, the usual ordering of the unknowns (from bottom left to right) is preserved. Unlike in Jacobi or MG algorithms, here each subset x_p contains *only* the inner unknowns of its subdomain, but not the interface unknowns, contained in x_s. Let $\hat{N}_p$ be the number of *inner*

unknowns of subset p and N_s the number of unknowns of the interface s. The symbol $\hat{N}_p$ is used instead of N_p, that is reserved for the *total* number of unknowns (inner plus interface) of processor p.

According to the new ordination, the system can be expressed in terms of block matrices as:

$$\begin{bmatrix} A_{0,0} & 0 & & \cdots & A_{0,s} \\ 0 & A_{1,1} & & \cdots & A_{1,s} \\ \vdots & & & & \vdots \\ 0 & \cdots & & A_{P-1,P-1} & A_{P-1,s} \\ A_{s,0} & A_{s,1} & & \cdots & A_{s,s} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{P-1} \\ x_s \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{P-1} \\ b_s \end{bmatrix} \tag{6.4}$$

The sizes and structures of the submatrices in equation 6.4 are:

1. $A_{0,0} \cdots A_{P-1,P-1}$ are sparse matrices (penta or heptadiagonal if a structured mesh is used) with $\hat{N}_p$ rows and columns, that express the coupling of the inner unknowns of processor p.

2. $A_{0,s} \cdots A_{P-1,s}$ are sparse matrices with $\hat{N}_p$ rows and N_s columns that express the coupling of inner unknowns of processor p with the interface unknowns.

3. $A_{s,0} \cdots A_{S,P-1}$ are sparse matrices with N_s rows and $\hat{N}_p$ columns that express the coupling of interface unknowns with inner unknowns of processor p.

4. $A_{s,s}$ is a sparse matrix with N_s rows and columns that expresses the coupling of the interface unknowns.

In a parallel implementation, a conventional domain decomposition is done and then the interface nodes are treated separately. Each processor has an inner domain plus a part of the interface.

Example. To clarify the decomposition, consider the following situation. It is a domain with 6 nodes in x axis and 4 nodes in y axis, to be solved with $P = 3$ processors. A standard ordering of the unknowns in x vector can be as indicated in Fig. 6.1. The areas of the processors are filled in different colors.

	p=0		p=1		p=2		
4	19	20	21	22	23	24	
3	13	13	15	16	17	18	
2	7	8	9	10	11	12	
1	1	2	3	4	5	6	
j	1	2	3	4	5	6	i

Figure 6.1: Schur decomposition. Domain before reordering.

	p=0		p=1		p=2		
4	4	***23***	8	***24***	15	16	
3	3	***21***	7	***22***	13	14	
2	2	***19***	6	***20***	11	12	
1	1	***17***	5	***18***	9	10	
j	1	2	3	4	9	10	i

Figure 6.2: Schur decomposition. Domain after reordering.

In the example, we use a one-dimensional decomposition for x axis. Columns $x = 1, 2$ are assigned to processor $p = 0$, $x = 3, 4$ to $p = 1$ and columns $x = 5, 6$ to $p = 2$. The domain after the reordering has been represented in Fig. 6.2. The interface s is formed by two columns of nodes, indicated with italic boldface. Interface unknowns can be arbitrarily located at the east or at the west of the area of each processor. In our implementation, they are at the west (and at the north for two-dimensional domains). Nodes are now ordered first by subset and then, inside each subset, from bottom left to top right. The interface unknowns are numbered globally after the last subdomain $p = P-1$, from bottom left to top right. For instance, for $p = 1$ inner nodes are $5 \cdots 8$ and interface nodes 18,20,22,24. In the example, a diffusion equation with a equally spaced mesh is considered. Dirichlet boundary conditions are used, with a trivial equation $x_n = b_n$ for the boundary nodes. Under these conditions, the submatrices of A needed by processor $p = 1$ after the reordering are as follows:

$$A_{1,1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 \\ 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.5}$$

$$A_{1,s} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.6}$$

$$A_{s,1} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.7}$$

$$A_{s,s} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 4 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 4 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 4 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 4 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.8}$$

To avoid confusions, it is important to note that this example is an exception. In general, $A_{p_1,p_2} \neq (A_{p_2,p_1})^t$.

6.2.2 Solution of the reordered system

Gaussian elimination, treating each matrix as a scalar (this is, block Gaussian elimination), is used to transform the last block equation of system (6.4) from its original form,

$$[A_{s,0}, A_{s,1}, \cdots, A_{s,s}] \cdot [x_0, x_1, \cdots, x_{P-1}, x_s]^t = b_s \tag{6.9}$$

to:

$$\left[0, 0, \cdots, 0, \tilde{A}_{s,s}\right] \cdot [x_0, x_1, \cdots, x_{P-1}, x_s]^t = \tilde{b}_s \tag{6.10}$$

this is,

$$\tilde{A}_{s,s} x_s = \tilde{b}_s \tag{6.11}$$

where

$$\tilde{A}_{s,s} = A_{s,s} - \sum_{p=0}^{P-1} A_{s,p} A_{p,p}^{-1} A_{p,s} \tag{6.12}$$

and

$$\tilde{b}_s = b_s - \sum_{p=0}^{P-1} A_{s,p} A_{p,p}^{-1} b_p \tag{6.13}$$

Thus, equation (6.11) can be used to solve for the interface unknowns x_s *before* the interior values. Once x_s is known, each of the x_p can be determinated solving its original equation, extracted from equation (6.4):

$$A_{p,p} x_p = b_s - A_{p,s} x_s \tag{6.14}$$

After the difficulties associated with Block Jacobi algorithms, section 4.5.1, to be able to solve the interface before the inner nodes, seems magical ! ... but its just reordering of the unknowns and linear algebra. However, as we will see, the implementation of the technique is not that easy.

6.2.3 Evaluation of $\tilde{A}_{s,s}$ and $\tilde{b}_s$

The first problem to implement the algorithm is how to evaluate $\tilde{b}_s$ and $\tilde{A}_{s,s}$, as both involve $A_{p,p}^{-1}$ and its explicit computation is not possible except for very small problems. Assume that matrices $A_{s,p}$, $A_{p,s}$ and $A_{s,s}$ are made available to processor p. If each of the processors only has the coefficients of its part of the problem, communications are needed to reach this stage.

First, we will consider the evaluation of $\tilde{A}_{s,s}$. We rewrite equation (6.12) as

$$\tilde{A}_{s,s} = A_{s,s} - \sum_{p=0}^{P-1} \tilde{A}_{s,s}^p \tag{6.15}$$

here each of the terms:

$$\tilde{A}_{s,s}^p = A_{s,p} A_{p,p}^{-1} A_{p,s} \tag{6.16}$$

is the contribution from processor p to $\tilde{A}_{s,s}$. It can be evaluated column by column, without explicitly needing $A_{p,p}^{-1}$, proceeding as follows:

For column c from 1 to N_s, solve for auxiliary vector t in equation:

$$A_{p,p} t = [A_{p,s}]_c \tag{6.17}$$

where $[\cdot]_c$ is column c of a matrix. A lot of effort can be saved as many of the $[A_{p,s}]_c$ are null. Then,

$$\left[\tilde{A}^p_{s,s}\right]_c = A_{s,p} t \tag{6.18}$$

When the contribution of each processor is available, they are added and the result subtracted from $A_{s,s}$, to obtain $\tilde{A}_{s,s}$. In our implementation, all the processors need the matrix $\tilde{A}_{s,s}$, so MPI_Allreduce [155] is used to evaluate and propagate it. As $\tilde{A}_{s,s}$ depends only on A, it can be reused for different b vectors, like a LU decomposition. Note that although $A_{s,s}$ is a sparse matrix, this is not the case of $\tilde{A}_{s,s}$.

The same approach is used for $\tilde{b}_s$, but here only one system has to be solved. Equation (6.13) is expressed as:

$$\tilde{b}_s = b_s - \sum_{p=0}^{P-1} \tilde{b}^p_s \tag{6.19}$$

where

$$\tilde{b}^p_s = A_{s,p} A^{-1}_{p,p} b_p \tag{6.20}$$

Each processor evaluates its contribution $\tilde{b}^p_s$ as:

$$\begin{aligned} A_{p,p} t &= b_p \\ \tilde{b}^p_s &= A_{s,p} t \end{aligned} \tag{6.21}$$

6.2.4 Implementation. First approach

The algorithm is divided into two stages: *(i)* **Preprocessing** (Alg. 6.1) to be used just once per matrix and *(ii)* **Solution** (Alg. 6.2) of x for a given b, to be used once per b vector. Exact band LU decompositions has been used for $A_{p,p}$ and $\tilde{A}_{s,s}$. The conventional approach, however, is to use iterative solvers to do so (i.e., GMRES is used in [192]). The symbol $[\cdot]^r$ is used to denote the row r of a matrix.

Step P.3 involves communication as processor p does not have the coefficients that relate the inner nodes of neighbouring processors with its interface nodes. In our example, this is the case, for instance, of nodes $9, 11, 13, 15$, owned by $p = 2$ but coupled with interface nodes $18, 20, 22, 24$, owned by $p = 1$.

In the solution stage, each processor has to solve two local problems (steps S1 and S3) plus the interface problem S2, using the respective LU decompositions. There is only one communication operation per right hand side term to be solved.

In the implementation proposed in Alg. 6.2, all the processors solve simultaneously for the same interface values (step S.2). Doing so, communications are minimized but an important fraction of sequential work is introduced and a lot of memory is wasted due to data replication. Other alternatives are: *(i)* Solve the interface with a single processor and *(ii)* use a parallel solver for dense matrices. Both techniques were compared in [191] on a SP2, concluding that although the parallel interface solver is communication dominant, it is more efficient than the serial interface solver for large size problems. A better option (for the class of problems considered in this work) is presented in section 6.2.5.

6.2.5 Implementation. Distributed evaluation and storage of $\tilde{A}^{-1}_{s,s}$

Initial benchmarks of previous algorithm showed that, even using a full-matrix LU decomposition to solve for x_s in step S.2, as $\tilde{A}_{s,s}$ is a dense matrix, the cost of this operation can be high compared

```
Preprocessing of A {
    P.1-form A_{s,s} matrix :
        r = 1 → N_s {
            if owner(r) = me
                form [A_{s,s}]^r
        }
        send my rows of A_{s,s} to the other processors
        q = 1 → P − 1 {
            if q ≠ me
                receive rows of A_{s,s} owned by q
        }
    P.2-form A_{p,s} matrix
    P.3-form A_{s,p} matrix
    P.4-form A_{p,p} matrix and evaluate its LU decomposition
    P.5-evaluate Ã_{s,s}
        c = 1 → N_s {
            if [A_{p,s}]_c ≠ 0 {
                evaluate [Ã^p_{s,s}]_c using (6.17, 6.18)
            }
        evaluate T ← Σ_{p=0}^{P−1} Ã^p_{s,s} (all-reduce operation)
        Ã_{s,s} ← A_{s,s} − T
    P.6-evaluate and store the LU decomposition of Ã_{s,s}
}
```

Algorithm 6.1: Schur complement algorithm. Preprocessing (first approach).

```
Solution of Ax = b {
    S.1-evaluate b̃_s = b_s − Σ_{p=0}^{P−1} b̃^p_s
        S.1.1-solve A_{p,p} t = b_p
            using the pre-evaluated LU dec. of A_{p,p} (step P.4)
        S.1.2-evaluate b̃^p_s ← A_{s,p} t
        S.1.3-evaluate b̃_s ← Σ_{p=0}^{P−1} b̃^p_s (MPI_Allreduce)
        S.1.4-evaluate b̃_s ← b_s − b̃_s
    S.2-solve the interface nodes x_s from Ã_{s,s} x_s = b̃_s
        using the pre-evaluated LU dec. of A_{s,s} (step P.6)
    S.3-solve the inner nodes x_p from A_{p,p} x_p = b_s − A_{p,s} x_s
        S.3.1-evaluate t = b_s − A_{p,s} x_s
        S.3.2-solve A_{p,p} x_p = t
            using the pre-evaluated LU dec. of A_{p,p} (step P.4)
}
```

Algorithm 6.2: Schur complement algorithm. Solution (first approach).

with the band LU solution of the local problems. This leads to relatively poor performance of the previous algorithm.

On the other hand, processor p does not need all the x_s values to solve for its x_p subset. However, the LU algorithm (step S2) requires the evaluation of all the values.

The previous considerations suggested the following enhanced (for our application) version of the algorithm. The first idea was to evaluate and store $\tilde{A}_{s,s}^{-1}$. However, it is too large and we do not need *all* the matrix. If it was available we would use it to do the following matrix vector product:

$$x_s = \tilde{A}_{s,s}^{-1}\tilde{b}_s \tag{6.22}$$

However, we do not need all the components of x_s, but only the values of x_p directly coupled with our inner area. For instance, in our example $p = 1$ only needs x_s in nodes $17, 19, 21, 23$ (in $p = 0$ area) and $18, 20, 22, 24$ (in its own area). Thus, we only need the rows of $\tilde{A}_{s,s}^{-1}$ associated with the required values of x_s. To evaluate them we use the expression:

$$\left[Q^t\right]^{-1} = \left[Q^{-1}\right]^t \tag{6.23}$$

where Q is any non-singular square matrix. The rows of $\tilde{A}_{s,s}$ are evaluated as columns of its transpose. The modified algorithms are Algs. 6.3 (only the modified steps are shown) and 6.4.

Preprocessing of A {
...
P.6-evaluate and store the LU decomposition of $\tilde{A}_{s,s}^t$
P.7-evaluate the required subset of the rows of $\tilde{A}_{s,s}^{-1}$
    $\tilde{A}_{s,s}^{-1} \leftarrow 0$
    for $c = 1 \rightarrow N_s$ {
        if I need component c of x_s {
            solve for t in $\tilde{A}_{s,s}^t t = e_c$, using $\tilde{A}_{s,s}^t$
            $\left[\tilde{A}_{s,s}^{-1}\right]_c \leftarrow t$ (store t as the c row of $\tilde{A}_{s,s}^{-1}$)
        }
    }

Algorithm 6.3: Schur complement algorithm. Preprocessing (second approach).

After step P.7, LU decomposition of $\tilde{A}_{s,s}^t$ is not needed anymore, so it can be deallocated. As actually only the rows of $\tilde{A}_{s,s}^{-1}$ in use are stored by each processor, while the full LU decomposition was needed, so the modification of the algorithm not only saves CPU time but also memory.

Each interface node is needed by processors at both sides of the interface. Currently both evaluate the corresponding row of the matrix. An additional enhancement can be to parallelize step P.7, saving CPU time in the preprocessing stage. However, depending on the ratio F/B of the computer (section 4.2.1), this may not be convenient. Additionally, depending on N_s, it might be better to solve for t in step P.7 using another method rather than LU decomposition of $\tilde{A}_{s,s}$. However, as our main interest is not the preprocessing stage, these possible enhancements have not been implemented.

When the algorithm is used with a single processor, $P = 1$, a conventional (sequential) LU decomposition is used (section 2.7.2). Thus, each processor has to solve just once all the domain.

6.2.6 Sparse matrix storage

A lot of memory and CPU time can be saved storing matrices $A_{s,p}$, $A_{p,s}$ and $\tilde{A}_{s,s}^{-1}$ as sparse. It is worth doing it even to solve small problems. The sparse matrix storage technique used in our implementation has been choosen considering that in the solution stage, the matrices are only needed to evaluate matrix-vector products. For each row, the non-null columns and its indices are stored using a double and a integer vector. For instance, in our example, $A_{1,s}$ is stored as:

Solution of $Ax = b$ {
S.1-evaluate $\tilde{b}_s = b_s - \sum_{p=0}^{P-1} \tilde{b}_s^p$
S.1.1-solve $A_{p,p}t = b_p$
using the pre-evaluated LU dec. of $A_{p,p}$ (step P.4)
S.1.2-evaluate $\tilde{b}_s^p \leftarrow A_{s,p}t$
S.1.3-evaluate $\tilde{b}_s \leftarrow \sum_{p=0}^{P-1} \tilde{b}_s^p$ (MPI_Allreduce)
S.1.4-evaluate $\tilde{b}_s \leftarrow b_s - \tilde{b}_s$
S.2-solve the interface nodes x_s from $\tilde{A}_{s,s}x_s = \tilde{b}_s$
S.2.1-evaluate $x_s = \tilde{A}_{s,s}^{-1}\tilde{b}_s$ where needed
using the required subset of rows (pre-evaluated in P.7)
S.3-solve the inner nodes x_p from $A_{p,p}x_p = b_s - A_{p,s}x_s$
S.3.1-evaluate $t = b_s - A_{p,s}x_s$
S.3.2-solve $A_{p,p}x_p = t$
using the pre-evaluated LU dec. of $A_{p,p}$ (step P.4)
}

Algorithm 6.4: Schur complement algorithm. Solution (second approach).

```
A1s={
  1->NULL
  2->[2,3,4] , [-1.,-1.]
  3->[2,5,6] , [-1.,-1.]
  4->NULL
}
```

Where the symbol NULL is used to denote an empty row. For non-null rows, the first vector contains the number of non-null elements and its position and the second, the floating point numbers. Incidentally, this solution is similar to the procedure adopted in the code Matlab [193], except that Matlab uses an additional integer vector to indicate the position of the non-null rows, instead of a NULL pointer. As for our application the time spent in step S2 is already low, the performance of both systems would probably be similar.

An object-oriented approach has been used to implement the sparse-matrix subroutines. The data structures are opaque to the external user, who can access the matrices through a set of functions such as add-row or matrix-vector. The same approach has been used to implement the Schur complement solver and the DDACM solver (section 7).

6.3 Benchmark

The problem model (section 2.7.1) was solved for different values of P and N in the JFF cluster. Solution times, summarized in Table 6.1, are excellent. Times are low and speed-ups, represented in Fig. 6.3, are very good[1]. The biggest problem that our version of the algorithm can solve with 256Mytes RAM per node (without swapping) is about $N = 350^2 = 122.500$. If swapping is allowed only during the preprocessing stage, larger problems can be solved. The majority of CPU time is spent solving band LU decompositions. This is done with the function proposed in [105] (as in section 2.7.2).

For $P = 2$, each processor has to solve twice a domain with $N/2$ equations. With a distributed evaluation and storage of $\tilde{A}_{s,s}^{-1}$ approach, the time to solve the interface equation is low. Thus, for large problems, as expected, the CPU times with $P = 1$ and $P = 2$ are similar. For more processors, one would expect, at best, to have $S = P/2$. Surprisingly, for large meshes, the results are much better: even a super-linear speed up is obtained in some cases. For $P = 16$ we have $S \approx 21$, while

[1] Note that these are *solution* times and not *pre-processing* times. Pre-processing times are much higher so this approach can not be directly applied to non-constant matrices such as momentum or energy discrete equations.

N	$P=1$	$P=2$	$P=4$	$P=9$	$P=16$
400	4.66×10^{-4}	1.40×10^{-3}	2.26×10^{-3}	2.81×10^{-3}	3.80×10^{-3}
1600	6.87×10^{-3}	7.25×10^{-3}	3.23×10^{-3}	3.66×10^{-3}	5.20×10^{-3}
3600	2.30×10^{-2}	2.40×10^{-2}	7.90×10^{-3}	5.51×10^{-3}	6.23×10^{-3}
6400	5.42×10^{-2}	5.48×10^{-2}	1.68×10^{-2}	8.83×10^{-3}	8.22×10^{-3}
10000	1.05×10^{-1}	1.05×10^{-1}	3.02×10^{-2}	1.40×10^{-2}	1.10×10^{-2}
22500	3.53×10^{-1}	3.49×10^{-1}	9.36×10^{-2}	3.83×10^{-2}	2.37×10^{-2}
40000	8.35×10^{-1}	8.26×10^{-1}	2.15×10^{-1}	7.52×10^{-2}	5.28×10^{-2}
62500	1.62×10^{0}	1.60×10^{0}	4.15×10^{-1}	1.39×10^{-1}	8.48×10^{-2}
90000	2.81×10^{0}	2.80×10^{0}	7.11×10^{-1}	2.39×10^{-1}	1.29×10^{-1}

Table 6.1: Execution times of Dual Schur solver.

the "theoretical" optimal would be $S=8$. This is remarkable, specially on a loosely coupled parallel computer.

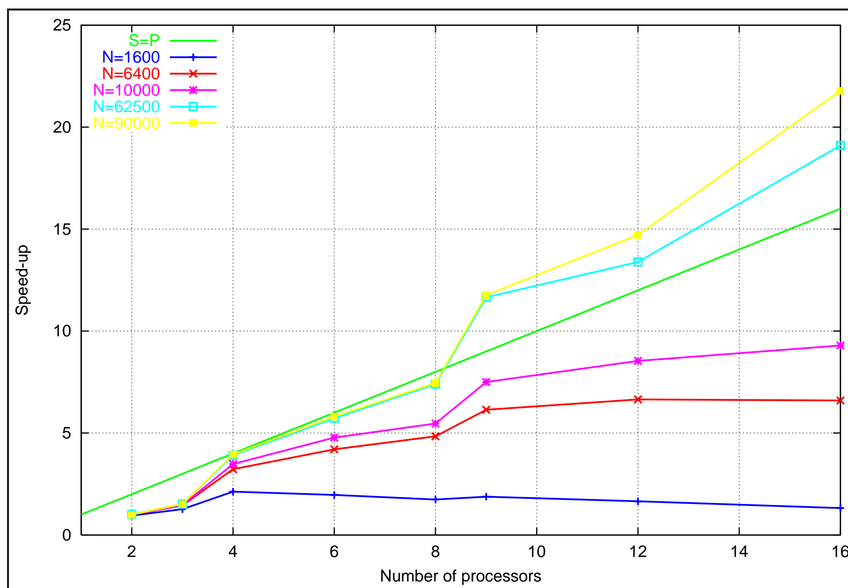

Figure 6.3: Speed-ups of the Schur algorithm in the JFF cluster.

To see the reason of this good behavior, we analyse the fraction of time spent in each of the solution steps. More than 95% of the time is spent in the following operations:

- Solution of the two band LU systems, steps S.1.1 and S.3.2.
- MPI_Allreduce operation, step S.1.3,
- Solution of the interface problem, step S.2.1.

For $P=16$, these times have been represented versus the problem size in Fig. 6.4. As a reference, the CPU time obtained with $P=1$ has also been represented.

Except for small problems, time due to communications and solution of interface problem is relatively low and grows less than LU solution time.

However, the reason for the super-linear speed up is the non-linear cost of the LU operations. For instance, for the case with $N=90.000$, each processor only has to solve two problems with $N=5.625$. As can be seen in Fig. 2.3, where CPU time of sequential LU solution is represented versus N, this is a huge difference.

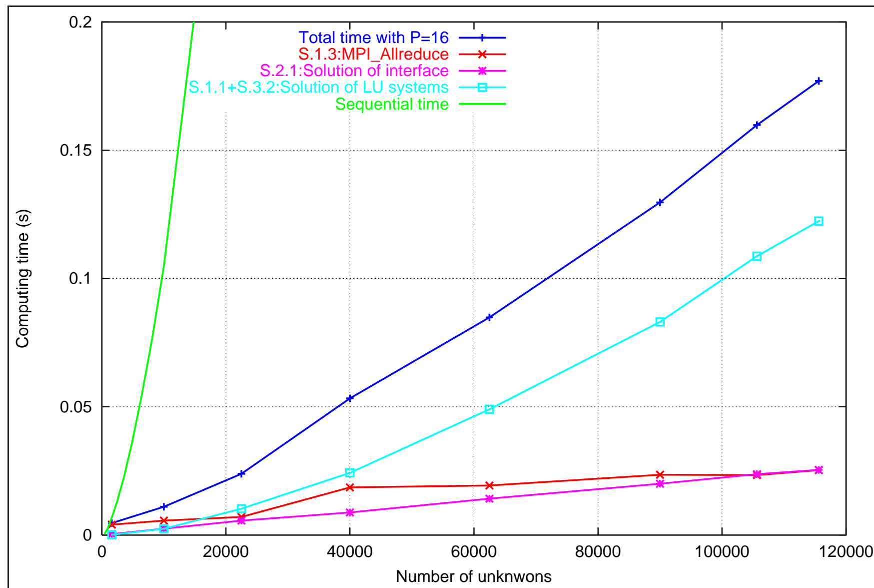

Figure 6.4: Breakdown of the processing time of the Shur solver, with $P = 16$ for different problem sizes.

It can be argued that using a better solver for each subset of x (Steps S.1.1 and S.3.2), the time for the sequential LU solution would be lower and thus the speed-up smaller. This claim is totally correct. However note that the first local problem (Step S.1.1) has to be solved accurately to avoid loosing precision. It should be investigated which could be the best option to do so. A nested Schur complement decomposition seems an interesting option. However, as our goal here are not the sequential algorithms, only used to measure of the speed-up, this has not been done here.

As the communications cost is low, the reduction of the processing time due to the use of better networks would be relatively small. For instance, assuming a zero cost of the MPI_Allreduce operation, total processing time with $N = 90000$ and $P = 16$ would be about 1.06×10^{-1} and $S \approx 26.4$.

6.4 Final remarks

In its area of application (situations where a relatively small and constant matrix has to be used to solve for many right-hand-side vectors) the Shur complement method is a remarkably efficient algorithm, if a fast method is used to solve the interface problem.

This holds specially for low cost parallel computers. As the data to be exchanged grows with the interface size, it is tolerant to low bandwidth networks. It is also tolerant to high latency as there is a single communication episode in all the solution process. In Chapter 7 it is used as an auxiliary component of the DDACM algorithm.

6.5 Nomenclature

A	matrix
b	right hand side
$A_{p,p}$	coupling matrix of inner unknowns of processor p
$A_{p,s}$	coupling matrix of inner unknowns of processor p with the interface
$A_{s,p}$	coupling matrix of the interface with the inner unknowns of processor p
$A_{s,s}$	coupling matrix of interface unknowns
$\tilde{A}_{s,s}$	matrix of interface equation
$\tilde{b}$	right hand side of interface equation
N	total number of unknwons
$\hat{N}_p$	number of inner unknowns of processor p
N_p	number of unknowns of processor p
N_s	number of unknowns of the interface
p	processor
P	number of processors
t	generic vector
T	generic matrix
s	interface
S	speed-up
x	unknown vector

Subindices

p	referent to processor p
s	referent to the interface