Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

MAS

Modelling, Analysis and Simulation

Modelling, Analysis and Simulation

An Unstructured Parallel Least-Squares Spectral Element Solver for Incompressible Flow Problems

Margreet Nool, Michael M.J. Proot

REPORT MAS-R0312 OCTOBER 22, 2003

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-3703

An Unstructured Parallel Least-Squares Spectral Element Solver for Incompressible Flow Problems

Margreet Nool[1] and Michael M.J. Proot[2]

[1] *CWI*

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

[2] *Delft University of Technology, Faculty of Aerospace Engineering,*

P.O. Box 5058, 2600 GB Delft, The Netherlands

Currently working at Shell Global Solutions International B.V., The Netherlands

ABSTRACT

The parallelization of the least-squares spectral element formulation of the Stokes problem has recently been discussed for incompressible flow problems on structured grids. In the present work, the extension to unstructured grids is discussed. It will be shown that, to obtain an efficient and scalable method, two different kinds of distribution of data are required involving a rather complicated parallel conversion between the data. Once the data conversion has been performed, a large symmetric positive definite algebraic system has to be solved iteratively. It is well known that the Conjugate Gradient method is a good choice to solve such systems. To improve the convergence rate of the Conjugate Gradient process, both Jacobi and Additive Schwarz preconditioners are applied. The Additive Schwarz preconditioner is based on domain decomposition and can be implemented such that a preconditioning step corresponds to a parallel matrix-by-vector product. The new results reveal that the Additive Schwarz preconditioner is very suitable for the p-refinement version of the least-squares spectral element method. To obtain good portable programs which may run on distributed-memory multiprocessors, networks of workstations as well as shared-memory machines we use `MPI` (Message Passing Interface). Numerical simulations have been performed to validate the scalability of the different parts of the proposed method. The experiments entailed simulating several large scale incompressible flows on a Cray T3E and on an SGI Origin 3800 with the number of processors varying from one to more than one hundred. The results indicate that the present method has very good parallel scaling properties making it a powerful method for numerical simulations of incompressible flows.

2000 Mathematics Subject Classification: 60K35,65F10,65N35,65Y05,68M14,68M20

1998 ACM Computing Classification System: G.1.6,G.1.8,G.4,J.2

Keywords and Phrases: Parallelization, Spectral/hp Elements method, Least-Squares method, Conjugate Gradient algorithm, Additive Schwarz Preconditioning, Unstructured Grids

Note: Funding for this work was provided by the National Computing Facilities Foundation (NCF), under project numbers NRG-2000.07 and MP-068. Computing time was also provided by HPαC, Centre for High Performance Applied Computing at the Delft University of Technology.

1. Introduction

For the accurate simulation of turbulent flows, one needs to be capable of resolving the main flow with great precision. This in turn necessitates the use of high order methods such as spectral element methods. These methods were, until recently, only applicable to generic problems with very simple geometries. As a consequence, high order methods were not very suitable for *real life* incompressible flow problems, because of their inability to simulate flows in complex domains and the enormous time needed to perform the *real life* simulations since the resulting algebraic systems are generally not symmetric and difficult to solve efficiently. This is due to the fact that the (algebraic) systems are of saddle point type and must be solved with numerical methods having poor convergence properties.

The conforming and nonconforming Galerkin spectral element methods for incompressible flows [8, 4, 2, 7, 5] allow complex geometries, but are still fairly time-consuming to solve real life problems due to the saddle-point structure of the algebraic system.

The least-squares spectral element method is based on two important and successful numerical methods: the spectral/*hp* element method and the least-squares finite element method. Least-squares spectral element methods, denoted by `LSQSEM`, combine the generality of finite element methods with the accuracy of the spectral methods and also the theoretical and computational advantages in the algorithmic design and implementation of the least-squares methods. These methods allows the use of equal order interpolation polynomials for all the variables. The choice of the spectral elements for the discretization of the least-squares formulation results in superior accuracy due to high-order basis functions. The accuracy of a least-squares spectral element discretization of the Stokes problem (cast in the velocity-vorticity-pressure form) has been reported in [12, 13] for different boundary conditions. In particular, spectral element formulations benefit from an exponential rate of convergence in smooth regions. Since the extension of the least-squares spectral element method for the Stokes equations to the Navier-Stokes equations is straightforward, only the parallelization of the Stokes problem is treated in the present paper.

In [9] the parallelization of the least-squares spectral element solver has been discussed for structured grids. In the present paper, the extension to unstructured discretizations is made. The parallelization is done with `MPI`. The domain is discretized with a mesh of non-overlapping conforming quadrilateral spectral elements. In the structured case, in a non-boundary node, always four spectral elements coincide and an interface edge will be either a *horizontal* or a *vertical* connection between spectral elements. Data transfer between adjacent spectral elements (e.g., situated at different parallel processors) can be performed from North to South and from East to West. After two sweeps the communication process will be accomplished. For the unstructured case, such data transfers are less straightforward: there is no matter of horizontal/vertical connections. It may happen that the east side of a spectral element is situated at the North side of its neighboring spectral element. Several definitions are needed to obtain an unambiguous approach on how data streams have to take place.

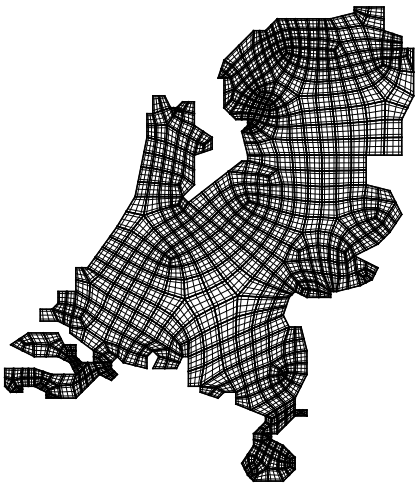

Figure 1: Map of *The Netherlands* as an example of an unstructured grid

As an example of an unstructured grid we choose the map of *The Netherlands*, with its intricate coastline, see Fig. 1. Although it is not representative for computational fluid dynamics grids, most

characteristics of 2D grids are covered by this example. The complexity can be extended by adding some holes in the domain. To restrict the amount of memory at the initialization phase, in which grids are built, each spectral element only has information about itself and its adjacent neighbor elements. Appendix A describes test cases used for the numerical experiments and their dimensions. In Appendix B, the data structure used for spectral elements is presented. For the solution part of the method, the same data distribution can be applied as for the structured grids and also the implementation of the conversion step agrees with the structured case.

Large scale problems demand for robust and parallelizable iterative solvers. Numerical methods based on the *least-squares principle* generate algebraic systems which are symmetric positive definite. Currently, the most popular iterative method for such problems is the preconditioned Conjugate Gradient method. To achieve a high convergence rate, we concentrate on well parallelizable preconditioners. Diagonal or Jacobi preconditioning is an example of an easy-to-parallelize method, but its convergence results are poor. The Additive Schwarz preconditioner appears to be very suitable for the high-order version of the `LSQSEM` solver.

The rest of the paper is organized in the following way. The least-squares spectral element formulation of the Stokes problem will be briefly discussed in Sect. 2. For an extended discussion involving the actual least-squares formulation, the treatment of the different boundary conditions, the a priori estimates and the local-global mapping operator, we refer to [11]. The investigation of the grid, the distribution of the spectral elements along the processors, the computation of the local systems, and data conversion to one very large global system and finally the implementation of the solution process is treated in Sect. 3. Sect. 4 describes the parallel platforms and the implementation tools. Four different kinds of test cases, outlined in Sect. 5, have been used in the numerical experiments. Their results, presented in Sect. 6, show that the implementation suited for distributed-memory systems is well scalable. We show, also in that section, that the advantage of the Additive Schwarz preconditioning increases with the refinement of the spectral element when compared to Jacobi preconditioning. In the last section, conclusions are given.

2. The least-squares spectral element method

We restrict ourselves to a short outline of the method in order to describe the parallelization steps. As stated above, only the Stokes equations will be treated in the present paper. The extension of the least-squares method to the Navier-Stokes equations is straightforward and can be found in [11].

2.1 The formulation of the Stokes problem

To obtain a *bona fide* least-squares formulation, the Stokes problem is *first* transformed into a system of first order partial differential equations by introducing the vorticity as an auxiliary variable. By using the identity

$$\nabla \times \nabla \times \mathbf{u} = -\Delta \mathbf{u} + \nabla(\nabla \cdot \mathbf{u})$$

and the incompressibility constraint $\nabla \cdot \mathbf{u} = 0$, the governing equations, in $\mathbb{R}^2$, subsequently read

$$\nabla p + \nu \nabla \times \omega = \mathbf{f} \quad \text{in } \Omega, \tag{2.1}$$

$$\omega - \nabla \times \mathbf{u} = \mathbf{0} \quad \text{in } \Omega, \tag{2.2}$$

$$\nabla \cdot \mathbf{u} = 0 \quad \text{in } \Omega, \tag{2.3}$$

where $\mathbf{u} = [u_1, u_2]^{\mathrm{T}}$ represents the velocity vector, p the pressure, ω the vorticity, $\mathbf{f} = [f_1, f_2]^{\mathrm{T}}$ the forcing term (if applicable) and ν the kinematic viscosity. In two dimensions, system (2.1)-(2.3) consists of four equations and four unknowns and is uniformly elliptic of order four.

2.2 Discretization
The domain is discretized with a mesh of $\mathcal{K}$ non-overlapping conforming quadrilateral spectral elements of the same order N. Each quadrilateral spectral element is mapped on the parent spectral element Ω^e by using an iso-parametric mapping to the bi-unitsquare $[-1,1]\times[-1,1]$ with local coordinates ξ_1 and ξ_2. In the parent element all variables, located at the Gauss-Legendre-Lobatto collocation (GLL) points, can be approximated by the same Lagrangian interpolant. Each spectral element gives rise to a local system of the form:

$$A_i\, z_i = f_i, \qquad \text{with } i = 1, \cdots, \mathcal{K}, \tag{2.4}$$

where the matrices A_i and right-hand side vectors f_i are given by

$$A_i = \int\limits_{\Omega_e} \left[\mathcal{L}\left(\psi_{0,0}\right), \cdots, \mathcal{L}\left(\psi_{N,N}\right)\right]^T . \left[\mathcal{L}\left(\psi_{0,0}\right), \cdots, \mathcal{L}\left(\psi_{N,N}\right)\right]\ d\Omega, \tag{2.5}$$

and

$$f_i = \int\limits_{\Omega_e} \left[\mathcal{L}\left(\psi_{0,0}\right), \cdots, \mathcal{L}\left(\psi_{N,N}\right)\right]^T\ \mathcal{F}\ d\Omega, \tag{2.6}$$

respectively and where $\psi_{i,j} = h_i\left(\xi_1\right) h_j\left(\xi_2\right)$.

Due to the continuity requirements between the C^0-spectral elements, some of the variables z_i corresponding to an internal edge will belong to more than one local system which necessitates the introduction of a global numbering.

2.3 The global system
The global assembly of the $\mathcal{K}$ local systems (2.4) can be obtained with:

$$KU = F \Leftrightarrow \left[\sum_{i=1}^{\mathcal{K}} \mathcal{G}_i^T A_i \mathcal{G}_i\right] U = \sum_{i=1}^{\mathcal{K}} \mathcal{G}_i^T f_i, \tag{2.7}$$

where $\mathcal{G}_i$ is a sparse *gathering* or Boolean matrix. The matrix K represents the symmetrical, globally gathered matrix of full bandwidth and the vectors U and F represent the global nodes (e.g., the unknown variables and *knowns*) and the global right-hand side function, respectively.

If the known components are numbered last, one can subdivide the vector U into an unknown component U_1 and a known component U_2. Consequently, the matrix K can be factored into submatrices $K_{1,1}$, $K_{1,2}$, $K_{1,2}^T$ and $K_{2,2}$ as well as the right-hand side vector F into the vectors F_1 and F_2. Hence, system (2.7) has the following structure

$$\begin{bmatrix} K_{1,1} & K_{1,2} \\ K_{1,2}^T & K_{2,2} \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix}, \tag{2.8}$$

which readily allows "*static condensation*" of the knowns, leading to the following sparse symmetric and positive definite system

$$K_{1,1}U_1 = F_1 - K_{1,2}U_2\ , \tag{2.9}$$

which can be solved with a (preconditioned) conjugate gradient method. The results in [10] revealed that it is necessary to construct the global system (2.9) in parallel to obtain a good scalable solver.

3. Parallel implementation aspects

We distinguish six steps in the `LSQSEM` method.

3.1 Investigation of the grid

First the coordinates of the numbered nodes are read from file. Each spectral element receives a unique number; a spectral element is determined by its four corners (also read from file). Furthermore, the boundary conditions of a spectral element on the physical boundary are given.

We start by determining, on a single processor called `ROOT`, the neighbors of a spectral element. For structured grids, it is simple to determine the North, East, South and West neighbors, as described in [9]. For unstructured grids, however, the North edge of a spectral element will not always border on the South edge of an adjacent spectral element, especially if the number of elements having a common vertex does not equal to four. The following definition is used to determine whether two spectral elements are neighbors:

Definition 1 *Two spectral elements are neighbors when they have two successive vertices in common.*

The edges are numbered and the number of the adjacent spectral element will be stored for each edge. A spectral element with less than four neighboring spectral elements must be a boundary element.

In order to treat special unstructured cases around internal vertices and along the boundaries, several `LOGICAL` arrays (of length 4) are a part of the `TYPE` structure of a spectral element. The information stored per spectral element is only restricted to the spectral element itself and its neighbors, according to Def. 1. Information of neighbors of neighbors is not available to limit the amount of data, see Appendix B.

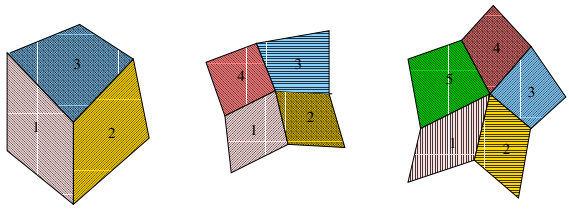

Figure 2: Interior node cases

Let a spectral element, also called a cell, be the smallest computational unit. In the initialization phase, the collocation points get their velocity-vorticity-pressure values, a local number, but also a unique global number. Especially for unstructured grids, this global numbering takes more effort in a distributed-memory environment. It is necessary to indicate for which spectral element the initialization of a vertex takes place to preclude that a component of a collection point awards more than one global number. Secondly, it must be determined in advance that if values are received from a neighboring cell whether they have to be copied through or not. Consider, e.g., the triple interior node illustrated in Fig. 2. Suppose the internal node components have been initialized in S_1 at p_1, then its values have to be sent to the left (S_3 at p_3) and to the right (S_2 at p_2). After this send operation, processor p_1 is ready to perform another task. At the same time processors p_2 and p_3 receive their data from p_1 and the initialization is finished. However in case of a four- or five-fold interior node, the data p_2 receives, has to be sent to its right neighbor S_3 on processor p_3 (see Fig. 2, *center* and *right* pictures). In other words, somehow the situation which is valid on a processor must be known and this information is stored in the `LOGICAL` arrays mentioned above.

Similar situations can occur for boundary nodes. Although S_2 in Fig. 3 is not really a boundary spectral element, one of its edges finds itself on the boundary and therefore its values are set according to the imposed boundary conditions. Moreover, S_1 and S_4 (*right* picture) are no neighbors according to Def. 1, but they have the same boundary node in common. The use of unstructured grids allowing the special cases as shown in Fig. 2 and 3 enables to improve the numerical solution.

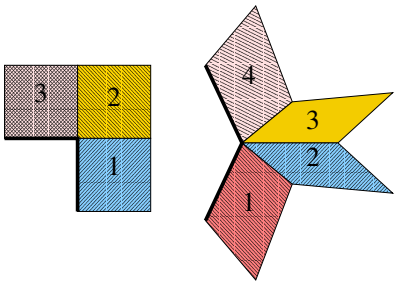

Figure 3: Special boundary cases

3.2 Spectral Element distribution
When the composition of the grid is known, the spectral elements can be distributed across $\texttt{MIN}(\mathcal{K}, \mathcal{P})$ processors, where $\mathcal{P}$ denotes the number of processors. One of the simplest partitioning schemes, the so-called *linear* scheme, is to assign each spectral element according to its number: the first $\mathcal{K}/\mathcal{P}$ spectral elements are assigned to processor 0, the next $\mathcal{K}/\mathcal{P}$ to 1, etc. Another simple partitioning is to scatter the spectral elements: the first one to processor 0, the second one to 1 etc., in such a way that the balance is preserved. We remark that in most cases the number of spectral elements will not be a multiple of the number of processors. We have tried to improve the mentioned partitionings by using Chaco [6], a software package designed to partition graphs. Some reduction in wall clock time was obtained in the `Geometry` phase (see Sect. 3.3), but disappears in the conversion phase, which appears to profit from a random partitioning. At the end, we choose for the linear scheme.

3.3 Geometry: grid initialization and numbering
This phase denotes the initialization of the grid, including the mapping from local to global numbering, which is one of the most complicated steps of the parallelization process. To acquire a unique global numbering, it is required to define which collocation points at the edges and in the vertices belong to a particular spectral element:

Definition 2 *Collocation points in a vertex belong to the spectral element numbered with the smallest value. In case the vertex is situated on the boundary, the collocation points belong to the* boundary *spectral element numbered with the smallest value.*

Definition 3 *Collocation points situated on an interface edge belong to the spectral element numbered with the smallest value.*

Definition 4 *Collocation points situated on a boundary edge belong to that particular spectral element.*

As soon as the collocation points of the spectral elements according to definitions (2)-(4) have been initialized and all of its components have been locally numbered, a unique global number can be attributed to its components, as described in [9]. The global numbers associated with internal edges and vertices will be copied to neighboring spectral elements. Fig. 4 illustrates how the global numbering **sweeps** through the grid: capital **T** denotes that components have already received their global number, capital **F** denotes that components are waiting for their values. The upper figures reflect the initialization phase and the arrows denote that the values on that edge are ready to be copied. The values associated with an edge of a spectral element numbered k, are ready to be copied to an edge of a neighboring spectral element numbered l, when the components of the vertices of spectral element numbered k, have already been initialized, denoted by a **T** in both end nodes, and $k < l$. The lower figures of Fig. 4 show the result of the first sweep and again the arrows indicate the next sweep. Notice that it takes four sweeps to complete the initialization in the left figure and two for the right

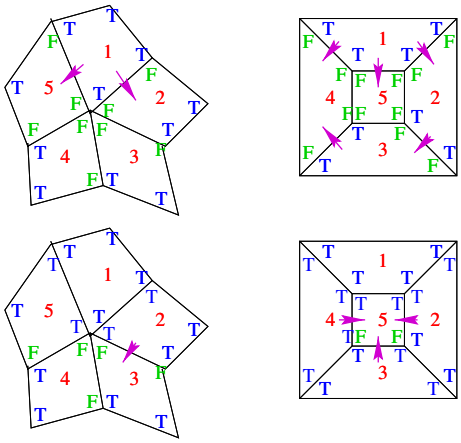

Figure 4: Example of the way the components get their global number

figure. Although there is very intensive data traffic between the processors this process requires less than a second of wall clock time (see e.g., Tables 2 and 4).

3.4 The computation of the local systems

The third phase computes the local systems and the right-hand side values. This part is the most time-consuming part of the calculations on spectral elements and can be perfectly parallelized (cf. Table 4, third column)

3.5 Data conversion

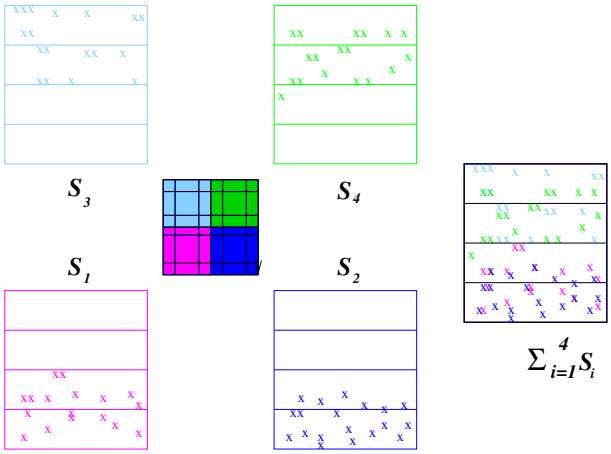

Figure 5: Schematic drawing of the computation of the global matrix K, being the sum over the contributions of $\mathcal{K} = 4$ spectral elements and K is subdivided into $\mathcal{P} = 4$ *strips*

After the local systems have been computed, the matrix K, or rather its parts $K_{1,1}$ and $K_{1,2}$ (cf. eq. (2.9)) can be constructed. In the preparation step, called `SUM`, the contribution of the local systems to matrix K are gathered. In [9] the way of communication applied in this step is treated in detail.

Of each spectral element, its components in the collocation points contribute to the large, global

matrix K, as illustrated in Fig. 5. These contributions to K are very sparse and in most cases the nonzeros are situated '*close*' together, based on their global numbering. The sparsity of K asks for a Compressed Sparse Row(CSR)-formatted storage approach. For basic operations on matrices, stored in CSR-format, subroutines from SPARSKIT [16] were used. A CSR-matrix addition $C = A + B$ with different sparsity patterns for A and B, is rather time-consuming. In general, the sparsity pattern of the resulting matrix C will deviate from those of the input matrices. In case the matrix K is constructed on a single process, $\mathcal{K} - 1$ CSR-additions must be performed. However, in case of a block-rowwise distribution of K across the processors, many of these additions will not be performed because the contribution of a single spectral element to such a *strip* is empty, due to the clustering of the nonzeros. Wall clock times for the CSR-matrix addition(s), being a part of the construction of a strip of the global matrix K, are listed in Table 1. In general, this process is not well-balanced, therefore the maximum measured time is given. We observe that in case of 8 or more PEs, the speed-up becomes larger than the number of PEs involved. In other words, super-linear speed-up is achieved in the matrix addition phase of the conversion step and is typical for the current parallel formulation.

Table 1: The maximum wall clock time on SGI 3800(TERAS) for the `CSR`-formatted matrix addition $C = A + B$. $\mathcal{P}$ denotes the number of processors used. (`Airfoil I`, $N = 6$)

$\mathcal{P}$	1	4	8	16	32	48	64	80	96	128
time	203	74.8	19.2	6.19	1.23	.479	.286	.288	.131	.105
speed-up	1.00	2.71	10.6	32.7	165	423	708	704	1547	1930

3.6 The preconditioned Conjugate Gradient process

Since the least-squares spectral element method yields symmetric positive definite algebraic matrices, a natural choice for the iterative solver is the preconditioned Conjugate Gradient (CG) method. In particular, since it can be proved that this method converges in a finite number of iteration steps.

It is well-known that an increase of the polynomial order improves the accuracy. However, the condition number of the stiffness matrix K grows rapidly with the polynomial degree N. As a consequence, the number of CG iterations increases and starts to dominate the solution method. To improve the convergence rate of the Conjugate Gradient process, Jacobi and Additive Schwarz preconditioners are applied. Both preconditioners can be parallelized, a necessary condition for our parallel code.

To start, the parallel Jacobi or diagonal preconditioner was used, but its convergence rate was low as shown in Sect. 6. A possibility to improve the convergence is to use overlapping or non-overlapping Additive Schwarz methods [15, 3] as preconditioning. For `LSQSEM`, based on non-overlapping subdomains, a preconditioner based on domain decomposition will be most plausible. A domain should coincide with a single spectral element or with a couple of neighboring spectral elements. Ainsworth et al.[1] proves that the growth of the condition number of the Additive Schwarz preconditioned system is bounded by $(1 + \log N)^2$ for an open surface and $(1 + \log N)$ for a closed surface. Moreover, this preconditioner attempts to be well parallelizable. The basic Additive Schwarz iteration is as follows:

1. *For* $i = 1, \ldots, \mathcal{K}$ *do*
2. *Compute* $\delta_i = \mathcal{G}_i^T A_i^{-1} \mathcal{G}_i (b - Ax)$
3. *End do*
4. $x_{new} = x + \sum_{i=1}^{\mathcal{K}} \delta_i$.

An obvious characteristic of this approach is that the components in each subdomain are not updated until a whole cycle of updates through all domains are completed. This is a very opportune

quality for parallel processing.

Let us consider the fixed-point iteration method of the form

$$x_{new} = Gx + f. \tag{3.1}$$

When preconditioning is applied, the system to be solved is commonly written as

$$M^{-1}Ax = M^{-1}b, \tag{3.2}$$

where the matrix M is called the *preconditioning* matrix. By applying the formulation of (3.1), we obtain the matrix

$$G = I - M^{-1}A. \tag{3.3}$$

Following Saad [15], we define the matrix

$$T_i = \mathcal{G}_i^T A_i^{-1} \mathcal{G}_i \quad \text{and} \quad P_i = \mathcal{G}_i^T A_i^{-1} \mathcal{G}_i A \tag{3.4}$$

to obtain

$$x_{new} = x + \sum_{i=1}^{\mathcal{K}} T_i(b - Ax) = (I - \sum_{i=1}^{\mathcal{K}} P_i)x + \sum_{i=1}^{\mathcal{K}} T_i b. \tag{3.5}$$

Observe that here

$$G = (I - \sum_{i=1}^{\mathcal{K}} P_i) \quad \text{and} \quad f = \sum_{i=1}^{\mathcal{K}} T_i b. \tag{3.6}$$

The result is that

$$M^{-1}A = \sum_{i=1}^{\mathcal{K}} P_i \quad \text{and} \quad M^{-1} = \sum_{i=1}^{\mathcal{K}} P_i A^{-1} = \sum_{i=1}^{\mathcal{K}} T_i. \tag{3.7}$$

According to Saad [15], the Additive Schwarz Preconditioner can be written as

---

1. *Input* v : *Output* : $z = M^{-1}v$
2. *For* $i = 1, \ldots, \mathcal{K}$ *do*
3. *Compute* $z_i = T_i v$
4. *End do*
5. *Compute* $z := z_1 + z_2 + \cdots + z_{\mathcal{K}}$

In Sect. 3.5, we mentioned that it is to prefer to construct the global system in order to solve eq. (2.9). Analogously, the preconditioning matrix M^{-1} will be assembled such that a preconditioning step corresponds to a (parallel) matrix-by-vector product. Therefore we have to interpret the operation $T_i = \mathcal{G}_i^T A_i^{-1} G_i$ carefully, especially its role on the interface nodes. Two possibilities to construct the preconditioning matrix arise:

1. The first one corresponds to the way the global system has been composed: collocation points on vertices and interface edges achieve contributions of neighboring spectral elements. Once the updated matrices T_i have been constructed they can be inverted and converted to a large global matrix.

2. The second approach is to start with the global matrix

$$K = \sum_{i=1}^{\mathcal{K}} \mathcal{G}_i^T A_i \mathcal{G}_i.$$

The elements of this matrix already have the correct values, being precisely the sum of the contributions of the spectral elements also those corresponding to interface collocation points on edges and vertices. Obviously, only values related to the *unknowns* will be of interest for the preconditioning matrix. This yields that not K but its submatrix $K_{1,1}$ contains exactly the values we are looking for.

A reverse operation from the global matrix $K_{1,1}$ to the local, updated matrices $U_i, i = 1, \cdots, \mathcal{K}$ forms the basis for the preconditioning. The distribution of the matrices U_i is chosen equal to the A_i distribution.

The inverse of U_i can be easily computed by the `LAPACK` routines `_POTRF` and `_POTRI`. We remark that the sparsity of U_i^{-1} differs from that of A_i and in practice the number of zeros in U_i^{-1} appears to be less. The computation of the preconditioning matrix

$$M^{-1} = \sum_{i=1}^{\mathcal{K}} \mathcal{G}_i^T U_i^{-1} \mathcal{G}_i \tag{3.8}$$

can be performed analogously to the computation of $K_{1,1}$.

Remark 1 *Numerical experiments with an alternative preconditioner which uses the non-updated local matrix A_i^{-1} instead of U_i^{-1}, demonstrate that the number of iterations does not decrease. Another attempt with a simply weighted update of A_i with weighted factors corresponding to the multiplicity of the values on edges and vertices does not improve the convergence either.*

Remark 2 *The Additive Schwarz preconditioning requires substantially more memory than the Jacobi diagonal preconditioning. Both the* large *global matrices $K_{1,1}$ and M^{-1} must be available simultaneously, of course distributed over the processors involved and stored in* `CSR` *format. At the time of construction of the preconditioner also memory for the* updated *matrices U_i must be allocated. The original matrices A_i embedded in $K_{1,1}$ and $K_{1,2}$ have already become superfluous at that time.*

4. Parallel platforms and implementation tools

The calculations have been performed on

- Cray T3E system Vermeer (named after the Dutch painter) at HPαC, Centre for High Performance applied Computing of the Delft University of Technology, with 128 user processing elements (PEs) interconnected by the fast 3D torus network with a peak performance of 76.8 Gigaflop/s. The T3E uses the DEC Alpha 21164 for its computational tasks. Each PE is configured with 128 Mbytes of local memory, providing more than 16 Gbytes of globally addressable distributed memory.
- The SGI Origin 3800 TERAS with 1024 500 MHz RI 14000 processors, subdivided into six partitions, two (interactive) 32-CPU partitions and four batch partitions of 64, 128, 256 and 512 CPU's, respectively. The theoretical peak performance is 1 Teraflop/s. A message passing model is allowed on the Origin using optimized SGI versions of e.g., `MPI`.

To obtain good portable programs which may run on distributed-memory multiprocessors, networks of workstations as well as shared-memory machines we use `MPI`. Besides the standard communication mode, it appears to be necessary to use the buffered-mode send operation `MPI_BSEND` (see e.g., [14]). This operation can be started independent of whether a matching receive has been posted. Its completion does not depend on the occurrence of a matching receive. In this way deadlock situations, in which all processes are blocked, are prevented.

5. The test cases

5.1 The Netherlands test case

The performance of the '*The Netherlands*' grid (see Fig. 1) is demonstrated by means of a smooth model problem. The velocity boundary condition is used in the numerical simulations; the pressure level is set at one node. More details, like the number of spectral elements, can be found in Appendix A.

5.2 Cylinder test cases

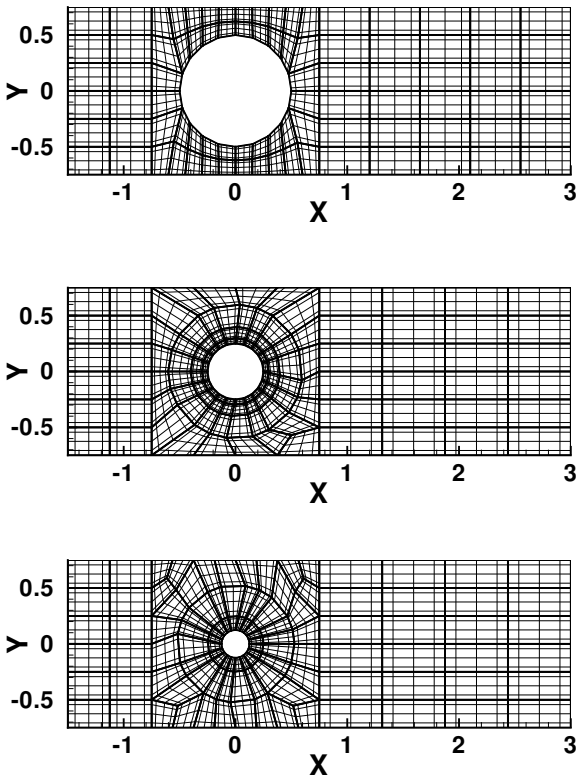

Figure 6: The spectral element grids (of order $N = 4$) used to simulate the Stokes flow of the cylinder test cases: large (`cyl01`), medium (`cyl02`) and small(`cyl03`)

In [11], three test cases of a Stokes flow around a cylinder have been considered. In this paper, the same simulations are examined, except that here the emphasis is on the parallel performance of the solution method. For more details on the overall quality of the least-squares spectral element method concerning the cylinder test cases compared to the Galerkin method, we refer to [11]. For these simulations, a cylinder with diameter d moves with speed 1 along the centerline of a narrow channel of height $h = 1.5$ filled with a fluid with density $\rho = 1$. The three different cylinders have been considered while keeping the geometry of the channel wall fixed. The cylinder diameters d are 1.0, 0.5 and 0.25, respectively. We refer to these test cases as the *large*, the *medium* and *small* cylinder test case. The level of difficulty for the three test cases is comparable with respect to parallelization. The grid of the large cylinder test can be considered as a structured grid with a *rectangle* hole inside. The middle and small test cases are examples of unstructured grids. For all cases the same technique, developed for unstructured grids has been used to build up the global system.

More details, like the number of spectral elements, can be found in Appendix A. The grids are shown in Fig. 6. The boundary conditions are defined in the following way: the velocity components ($\mathbf{u} = [1, 0]$) are prescribed on the channel boundary. A no-slip velocity boundary condition ($\mathbf{u} = 0$) has been prescribed on the boundary of the cylinder. The pressure constant has been set to zero at point $(x, y) = (-1.5, 0.75)$.

5.3 NACA-4412 airfoil test cases

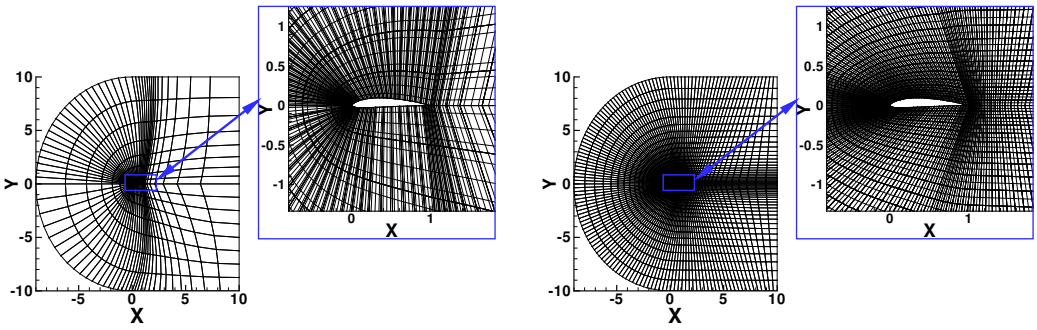

Figure 7: Two boundary-fitted grids around the NACA-4412 airfoil: `Airfoil I` (*left* graph), and `Airfoil II`, (*right* graph)

Another class of Stokes problems considered in this paper corresponds to flow around a NACA-4412 airfoil (Fig. 7). More details, like the number of spectral elements, can be found in Appendix A.

5.4 Stokes cylinder test case

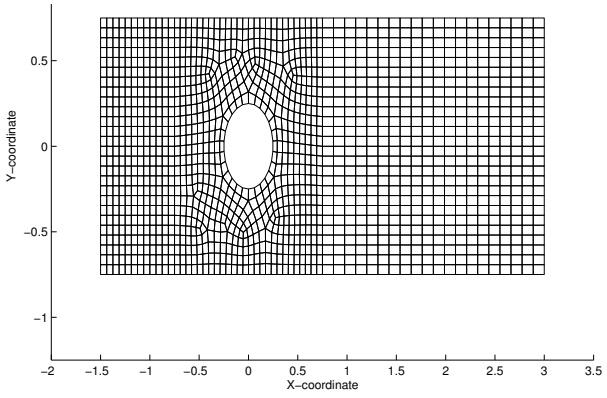

Figure 8: The Stokes Cylinder test case

Along the cylinder wall of the Stokes cylinder test grid (see Fig. 8) a no-slip condition has been prescribed. The velocity components on this boundary are: $\mathbf{u} = (1, 0)$. The pressure has been set to zero on the left boundary. More details, like the number of spectral elements, can be found in Appendix A.

6. Numerical experiments

In this section several aspects of the parallelization of the least-squares spectral element are discussed. The emphasis is on the scalability of the method on the platforms discussed in Sect. 4 using a large number of processors. Although on both machines the same `MPI`-implementation is executed, the scalability varies, because of a different ratio between the computational speed and communication time of the Cray T3E(Vermeer) and the SGI 3800(TERAS). Moreover, the Vermeer has been used in dedicated mode, which implies that a request for $\mathcal{P}$ PEs will always result in allocation of exactly $\mathcal{P}$ processors. This contrasts the behavior of the TERAS on which several processes will run on the same processor and the number of processors involved may differ from $\mathcal{P}$. As a consequence, the timing results achieved for the Vermeer are more reliable than those measured for the TERAS.

The execution speed of the TERAS compared to those of the Vermeer is much higher, since:

- the peak performance per processor is nearly twice as high,
- the communication between processors is faster,
- more memory per processor.

For small numbers of processors, the processors may run out of memory. In those cases no results are shown (see e.g., Fig. 9 (Vermeer)) and the speed-ups are based on extrapolated values.

The importance of testing the execution on many parallel processors (about one hundred) is that the negative effect on the parallel speed-up of non-scalable operations like broadcasting becomes visible. In Sect. 6.4, the broadcast operation dominates the performance of the solution process for large $\mathcal{P}$, and, therefore an alternative approach has been implemented. Secondly, Sect. 6.5 urges effective preconditioning since large problems mostly suffer from large condition numbers. Thirdly, the use of memory deserves particular notice. Some aspects of the geometry of the spectral element discretization will be determined on a single processor: e.g., which elements are neighbors and what will be the multiplicity of both the internal and boundary nodes. Clearly, the required amount of information on the `ROOT` processor depends on the problem size and may not cause memory overflow. To prevent overflow on `ROOT`, the data must be restricted per spectral element. After the data have been distributed/broadcasted to the processors involved, it may be extended per spectral element.

6.1 Memory use

In Appendix B the data structure of a spectral element/cell is shown. The memory for the allocatable arrays will be allocated after the distribution of the spectral elements. Unfortunately, due to allocation and deallocation of memory during execution of the different phases and the conversion of one distribution into another one, a good reliable formula for the amount of data required on a single processor is not available. We emphasize that everywhere in the code the allocation of memory has been minimized as much as possible. Especially, the data conversion part has been coded very flexible such that the allocation of data is not bounded by an estimate, but based on the really counted number. As an example, consider the construction of the global matrix K. Theoretically, each processor may receive data from all spectral elements distributed across all available processors. In practice however, an increase of $\mathcal{P}$ will lead to less communication between the spectral elements and a part of the global matrix. It appears that only a restricted number of spectral elements contributes to a particular matrix *strip*.(Recall Sect. 3.5.) We remark that an allocation statement may never depend linearly on $\mathcal{P}$, because then an increase of $\mathcal{P}$ will lead to an increase of data allocation per processor instead of a decrease.

6.2 Wall clock times

To give an impression of the execution times achieved for `LSQSEM`, we present the results in different ways: i) for both machines the wall clock times, depending on $\mathcal{P}$, including the timings of parts of the implementation, ii) for several test cases the wall clock times as a function of the number of processors, iii) for a fixed number of processors the wall clock times as a function of the system size,

and iv) for a single machine the wall clock times, depending on $\mathcal{P}$, including the timings of parts of the implementation for different polynomial orders.

Table 2: Wall clock times (in seconds) obtained for *The Netherlands* grid with $N = 4$

SGI 3800, TERAS						
$\mathcal{P}$	Spectral elements		Conversion		CG	Complete
	`Geometry`	Stokes	`SUM`	`CSR`	time	time
1	0.54	16.80	3.08	235.05	2478.2	2734.4
2	0.24	8.40	1.55	74.94	1267.2	1352.7
4	0.18	4.17	0.80	24.10	740.9	770.4
8	0.08	2.02	0.42	8.06	330.3	341.2
16	0.10	1.10	0.50	3.02	210.3	215.8
24	0.10	0.72	0.24	1.61	152.0	154.9
Cray T3E, Vermeer						
$\mathcal{P}$	Spectral elements		Conversion		CG	Complete
	`Geometry`	Stokes	`SUM`	`CSR`	time	time
8	0.36	40.37	2.58	70.36	1361.4	1476.0
16	0.22	20.44	1.33	25.17	740.2	788.6
32	0.16	10.44	0.71	8.22	646.4	667.2
64	0.14	5.23	0.44	3.33	237.0	248.5

The global system of the *the Netherlands* grid is solved by about 3400 CG iterations (Jacobi preconditioning) with as stopping criterion:

$$||residual||_2 \leq tol_{rel} \times ||residual_{init}||_2 + tol_{abs}, \tag{6.1}$$

where $tol_{rel} = 10^{-10}$ and $tol_{abs} = 10^{-16}$. Table 2 lists the wall clock times achieved on both the TERAS and the Vermeer. Obviously, most time is spent by the CG solution process (more than 90 %). The conversion part is expensive in case a few processors are involved. In Sect. 6.3 we will elaborate upon this point. The initialization part of the spectral elements (called Stokes) takes a few seconds and is perfectly scalable.

To display comparable wall clock times for the cylinder test cases, the polynomial approximation N of the spectral elements has been set to 6 for the computations on the Vermeer and to 8 for the computations on the TERAS for all cylinder diameter problems. In case $N = 6$, Additive Schwarz preconditioning results in 356, 357 and 365 CG iterations, respectively, independently of the number of processors used. The number of CG iterations required in case of $N = 8$ is 581, 566 and 567, respectively. In general, varying the number of processors results into a constant number of iterations. Fig. 9 (*left*) represents the wall clock times (in seconds) for the cylinder test cases (cf. Sect. 5.2) on both the Vermeer and the TERAS. Along the horizontal axis the number of processors $\mathcal{P}$ is shown. The results achieved for the T3E satisfy our expectations: the wall clock times decrease for growing $\mathcal{P}$. We like to comment on the deviant behavior of the results achieved at the TERAS for large $\mathcal{P}$. We analyzed that this behavior arises from the solution process of the global system. In Sect. 6.4, we will give an explanation about this deviation.

Wall clock times for the NACA-4412 airfoil test cases are given in Fig. 9 (*right*). For a *small* value $\mathcal{P}$ the speed-up for the complete process is very good (more than 13 achieved on 16 PEs of the Vermeer (`Airfoil I`, $N = 2$, `T3E`)). Again, we observe that an extension of the number of processors does not always reduce the wall clock time for experiments on the TERAS (dotted lines) with $\mathcal{P} > 48$. Results achieved on the Vermeer do not show a large increase in wall clock time. In Sect. 6.4, we will return to this.

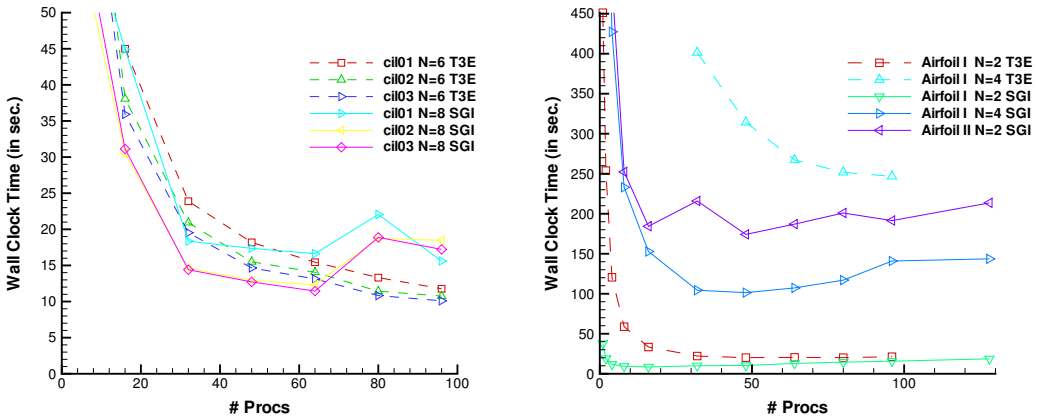

Figure 9: The total wall clock time for the cylinder test cases (*left*) and the NACA-4412 airfoil test cases (*right*) executed on both the Cray T3E(Vermeer) and the SGI 3800(TERAS). Additive Schwarz has been used as a preconditioner

Table 3: The NACA-4412 grids and their wall clock times on 32 processors

	N	System size	# iterations	Complete time $\mathcal{P} = 32$
`Airfoil I`	2	9334	1430	9.2
`Airfoil I`	4	37376	4032	102.7
`Airfoil II`	2	59072	4575	211.8
`Airfoil I`	6	84096	8159	857.4
`Airfoil I`	8	149504	14120	2863.2
`Airfoil II`	4	236288	14648	3324.6

Table 3 shows the system size, the number of iterations and the wall clock time (on the SGI 3800(TERAS)) achieved on 32 processors. In accordance to an increase of the system size, the number of iterations grows. We do not give judgment on the best choice for grid and polynomial order.

Table 4 gives the wall clock times for the Stokes cylinder problem for the Cray T3E(Vermeer). `Geometry` does not depend on N, whereas the third column, called Stokes, shows scalable results, Conversion is expensive for small $\mathcal{P}$, just like the computation of the Additive Schwarz preconditioner. In case $\mathcal{P} = 96$, it takes about 1.5 % of the execution time. The percentage of the solution time by CG to the complete time increases from 93 % for $N = 2$ to 97 % for $N = 4$.

6.3 The performance of process steps

Fig. 10 illustrates the speed-up for several parts of the `LSQSEM` code: 1) the computation of the local systems (`make local systems`) 2) the construction of the Additive Schwarz preconditioning matrix (`make preconditioner`) and 3) data conversion phase (`convert data`). Especially, the speed-up lines achieved for the Cray T3E illustrate the good parallel performance of the initialization steps of the `LSQSEM` code. Again, like in the data conversion step discussed in Sect. 3.5 super-linear speed-up is obtained, now for `make preconditioner`. We remark that the main parts of the implementation of `make preconditioner` are also used within the data conversion step, like the `CSR`-formatted matrix

Table 4: Wall clock times (in sec.) measured on the Cray T3E(Vermeer) for the *Stokes cylinder* grid

Stokes cylinder, $N = 2$							
$\mathcal{P}$	Spectral elements		Conversion		Make	CG	Complete
	Geometry	Stokes	SUM	CSR	preconditioner	time	time
8	0.19	0.35	0.35	14.71	32.70	106.94	156.15
16	0.15	0.18	0.23	6.29	11.50	63.01	82.22
32	0.13	0.09	0.18	2.24	3.55	41.29	48.35
48	0.12	0.06	0.16	1.34	1.83	37.69	42.02
64	0.15	0.05	0.16	1.08	1.29	33.11	37.29
80	0.13	0.05	0.15	0.94	0.97	35.56	39.46
96	0.13	0.04	0.16	0.78	0.73	34.05	36.78
Stokes cylinder, $N = 4$							
64	0.18	0.46	0.30	7.40	12.54	523.04	544.81
80	0.16	0.36	0.28	5.32	8.84	491.90	507.73
96	0.16	0.33	0.26	3.92	6.69	473.59	485.82

addition $C = A + B$, see also Table 1. The speed-up achieved on the TERAS is a bit disappointing. Both the `make preconditioner` and `convert data` phases require communication, which is *slow* compared to the computing capacity of the TERAS.

6.4 CG Solution process

Fig. 11 shows the wall clock times of the solution process of the NACA-4412 `Airfoil I` grid ($N = 4$) achieved for the Cray T3E(Vermeer,*left*) and SGI 3800(TERAS,*right*). Obviously, the amount of parallelism decreases; for the Cray T3E we obtain $T_{32} : T_{64} = 1.62$ and $T_{48} : T_{96} = 1.50$. For the TERAS the results are even worse: $T_{32} : T_{64} = 0.98$ and $T_{48} : T_{96} = 0.72$. An increase in the number of used processors results in a growth of the wall clock time!

Let us concentrate on the main parts of the solution process: each CG iteration involves three `SAXPY` operations, two inner products and one matrix-by-vector product. Application of the Additive Schwarz preconditioning adds one more matrix-by-vector product per iteration step. The `SAXPY` operations can be performed perfectly in parallel, and are also well-scalable. The computation of the inner products requires communication, of course. However, from numerical experiments it appears that their contribution to the total execution time is small and can therefore be neglected. Hence, the decrease in speed-up must arise from the matrix-by-vector product(s), something we did not expect in advance.

What makes the matrix-by-vector so expensive for a large number of CPUs? The Compressed Sparse Row storage format of the global matrix is one of the most general schemes for storing sparse matrices. To perform the matrix-by-vector product $y = Ax$ in parallel, using this format results in a vector y of which each component can be computed independently as the dot product of the i-th row of the matrix with the vector x. Each processor will handle a few rows that are assigned to it. If the matrix has been partitioned well-balanced the speed-up will be close to linear, in case each processing element contains a copy of the complete vector x. It appears that the origin of the growth in execution time is caused by the *gathering* of the x-vector on behalf of the matrix-by-vector product. Such a matrix-by-vector product results in a vector distributed across the processors corresponding to the distribution of the matrix over the PEs. However, to perform the next multiplication the resulting vector y will play the role of x and its parts must therefore be assembled on `ROOT` and be broadcasted to all processors. In case of the Additive Schwarz preconditioning it occurs twice per step. Fig. 11 presents, in addition, the main components of the solution process performed by CG. The line (`gather RHS`) denotes half the time for the complete gathering process; it corresponds to

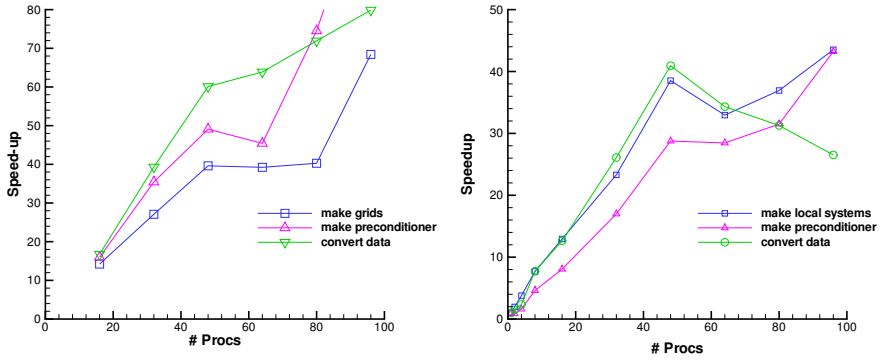

Figure 10: The speed-up for several parts of the `LSQSEM` code, achieved for the large cylinder test case executed on the Cray T3E(Vermeer,*left*) and the SGI 3800(TERAS,*right*)

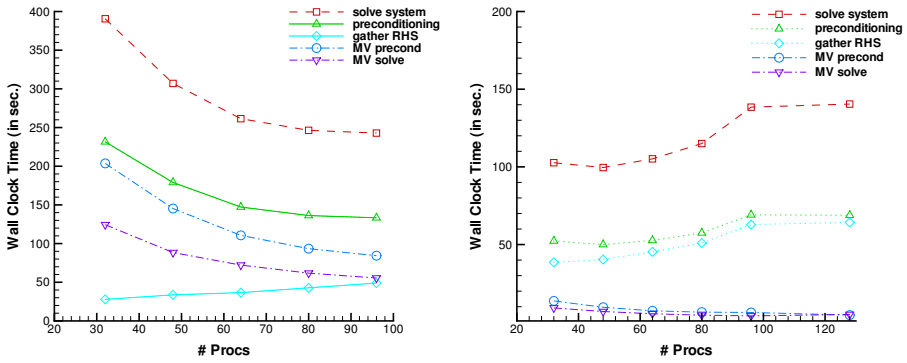

Figure 11: The Vermeer(Cray T3E,*left*) and TERAS(SGI 3800,*right*) wall clock times for several parts of the solution process of the NACA-4412 `Airfoil I` grid with $N = 4$. After 4032 CG iteration steps with Additive Schwarz preconditioning the residual has become small enough

the wall clock time of the gathering on behalf of (`MV precond`) or the wall clock time on behalf of (`MV solve`). The matrix-by-vector product (`MV solve`), that does not include the gathering of the input vector, denotes the time needed to multiply by the matrix $K_{1,1}$, the main contribution in case no preconditioning is applied. Additive Schwarz `preconditioning` takes the sum of the matrix-by-vector product (`MV precond`) and the gathering of the right-hand side vector. The matrix-by-vector multiplication with the preconditioning matrix M^{-1} requires more time than those on behalf of the solution process because this matrix is less sparse than the original global matrix $K_{1,1}$.

How can the wall clock time be reduced for the gathering process of the vector to be multiplied? Let us again focus on the *sparse* matrix-by-vector product $y = A.X$ on $\mathcal{P}$ parallel processors, where A is a block-rowwise distributed matrix and X denotes the entire vector, consisting of the parts of the distributed input vector x_i stored on processor i, $i = 0, \cdots, \mathcal{P}-1$. Each row of A requires only a few elements of X, i.e., the number of nonzeros of that particular row of A. The entire matrix A, however, will need the entire vector X. For increasing number $\mathcal{P}$, the number of rows per processor is decreasing and consequently, the dimension of the input vector x_i, $i = 0, \cdots, \mathcal{P}-1$. There will be

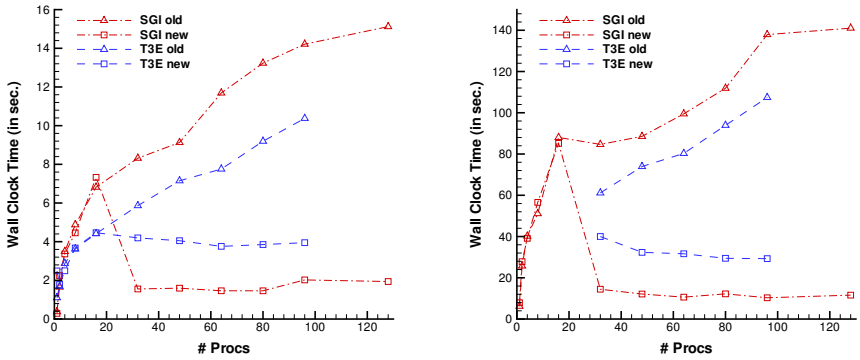

Figure 12: Wall clock times for the gathering part of the solution process of the NACA-4412 `Airfoil I` grid with $N = 2$ (*left*), and $N = 4$ (*right*). After 1562 and 4424 CG iteration steps, respectively, with Additive Schwarz preconditioning convergence has been achieved

a $\mathcal{P}$ such that the transfer of vector x_i to processor $\mathcal{P}_j$ on which a particular part of A is stored is superfluous: column numbers on $\mathcal{P}_j$ do not correspond to the elements of x_i on $\mathcal{P}_i$. In such case, the operation to create the entire vector X on each processor by gathering and broadcasting might be too expensive. An alternative solution is to create per processor a vector $\widehat{X}$ of the same dimension as X of which those parts x_i agree with X, which have at least one element that corresponds to a column number of A on the particular processor. Notice that, in case of large $\mathcal{P}$, the vectors $\widehat{X}$ differ per processor.

Fortunately, the iteration matrix A does not change per step, which enables to determine which parts x_i must be transferred only once. Hence, the wall clock time for that determination can be neglected. Fig. 12 illustrates that the wall clock time of the combination of `MPI_GATHER_VECTOR` and `MPI_BROADCAST`, denoted by `OLD`, increases for increasing $\mathcal{P}$. Note that `MPI_GATHER_VECTOR` creates the same vector X, independent of $\mathcal{P}$. In the best case, the wall clock time for this operation will be independent of $\mathcal{P}$. It is even more likely that the time increases with the number of processors. The vector X, that has to be broadcasted, has the same dimension for each value of $\mathcal{P}$, but the number of times it will be broadcasted equals $\mathcal{P}$. We assume that the entire time for $\mathcal{P}$ broadcasts will depend linearly on $\mathcal{P}$. The lines indicated by `NEW` are the results from the alternative approach, in which $\widehat{X}$ is computed instead of X. For $\mathcal{P} \leq 16$ the results agree with the *old* approach since much better results were obtained than by computing $\widehat{X}$, which will not or hardly differ from X. For larger values of $\mathcal{P}$ the gain is obvious, especially for the SGI 3800(TERAS).

6.5 Convergence behavior

Fig. 13 shows the number of matrix-by-vector products of both preconditioners for the cylinder test cases where `AS` denotes Additive Schwarz preconditioning and `Diag` denotes diagonal scaling. The results have been achieved for 16 processors. Since the initial residual error is larger in case of Additive Schwarz, we applied as stopping criterion:

$$||residual||_2 \leq tol_{abs},$$

where $tol_{abs} = 10^{-10}$, instead of the stopping criterion mentioned in (6.1). The reduction factor in the matrix-by-vector multiplications by applying Additive Schwarz compared to Jacobi preconditioning is more obvious for larger values of the polynomial degree N: it grows from 1.14 ($N = 4$) to 1.58 ($N = 10$), respectively. Only rounding errors sometimes cause some extra iterations. Despite the increase of memory allocation we prefer the Additive Schwarz approach.

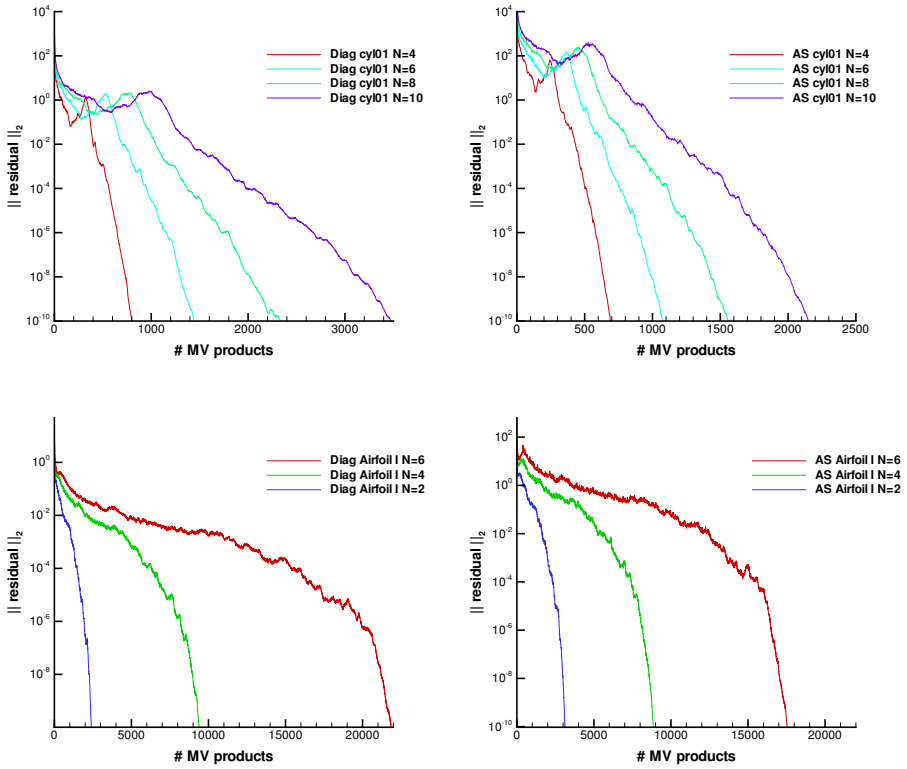

Figure 13: The convergence behavior of the Conjugate Gradient process for the large cylinder test case (`cyl01`, $N = 4, 6, 8, 10$)(*above*) achieved with Additive Schwarz preconditioning(`AS`) and Diagonal scaling(`Diag`), and, of the `Airfoil I` test case ($N = 4, 6, 8$)(*below*). A Jacobi CG iteration step takes one matrix-by-vector product, an Additive Schwarz iteration two matrix-by-vector products

The reduction in matrix-by-vector multiplications does not imply a reduction in wall clock time automatically, as can be concluded from Table 5. On the Vermeer (Cray T3E), a slight gain is obtained for $N = 8$. In all other cases the diagonal preconditioning takes less time. On the TERAS(SGI 3800) Additive Schwarz preconditioning is more expensive.

Table 5: The wall clock time in seconds for the solution process of the large cylinder test case. Timings were achieved for 32 processors on the Vermeer(Cray T3E) and 16 processors on the TERAS(SGI 3800). `Diag cyl01` ($N = 10$) ran on 64 processors on the Vermeer, `AS cyl01` ($N = 10$) was too large for the Vermeer

	Vermeer				TERAS			
	$N = 4$	$N = 6$	$N = 8$	$N = 10$	$N = 4$	$N = 6$	$N = 8$	$N = 10$
`AS cyl01`	5.01	29.88	118.33	too big	1.32	6.09	34.07	132.74
`Diag cyl01`	4.75	29.28	119.69	237.83	1.36	5.83	20.21	114.08

7. Conclusions

Least-squares spectral element methods result in symmetric and positive definite systems of linear equations which can be solved in parallel by CG. The parallelization of this kind of problems requires two different strategies. The numerical results confirm the good parallel properties of the element-by-element parallelization strategy. The combination of this strategy with the parallel CG solver results in a good parallelizable code to solve incompressible flow problems.

In [11], a direct substructuring method is presented, which seems very efficient for medium-scale two-dimensional problems. This strategy is not very useful for large-scale problems. For these kinds of problems iterative methods are to be preferred. However, the Conjugate Gradient method with Jacobi preconditioning is not efficient to solve the algebraic systems resulting from least-squares spectral element discretizations. By introducing the Additive Schwarz method, the convergence rate improves, especially for high values of the Lagrangian interpolants ($N >= 4$). By this, the choice between p-refinement and h-refinement becomes more obvious. Although the level of parallelism for the h-refinement version is higher, at least in the initial phase, due to a larger number of spectral elements, we expect that also on distributed memory machines p-refinement is advantageous: higher accuracy results can be obtained for the same CPU costs. The reason for that is that the largest part of the execution time is spent by the solution process. In that phase, the distribution along the available processors is always optimal, and, independent of the number of spectral elements.

A relatively high price for the gathering and broadcasting of the iteration vector on a large number of processors, especially measured at the TERAS, has been replaced by a more flexible approach. Not the entire input vector for the matrix-by-vector multiplication is created and broadcasted but only those parts are gathered which are actually needed on that particular processor. The main disadvantage of the previous implementation, an absolutely non-scalable part in the expensive solution process, in which an increase of the number of processors used causes a larger wall clock time, has disappeared.

References

1. M. Ainsworth and B. Guo. An Additive Schwarz preconditioner for p-version boundary element approximation of the hypersingular operator in three dimensions. *Numer. Math.* 85:343–366, 2000.
2. W. Couzy. Spectral Element Discretization of the Unsteady Navier-Stokes Equations and its Iterative Solution on Parallel Computers. *PhD Thesis*, Ecole Polytechnique Fédérale de Lausanne, 1995.
3. P.F. Fischer. An overlapping Schwarz method for spectral element solution of the incompressible Navier-Stokes equations. *J. Comput. Phys.*, 133(1):84–101, 1997.
4. P.F. Fischer and A.T. Patera. Parallel simulation of viscous incompressible flows. *Ann. Rev. Fl. Mech.*, 26:483–527, 1994.
5. R.D. Henderson and G.E. Karniadakis. Unstructured spectral element methods for simulation of turbulent flows. *J. Comput. Phys.*, 122(2):191–217, 1995.
6. B. Hendrickson and R. Leland. The Chaco user's guide. *Tech. Rep. SAND942692,* SANDIA National Laboratories, Albuquerque, NM, 1993.
7. G.E. Karniadakis and S.J. Sherwin. *Spectral/hp Element Methods for CFD.* Oxford University Press, 1999.
8. Y. Maday and A.T. Patera. Spectral element methods for the incompressible Navier-Stokes equations. In A.K. Noor and J.T. Oden (eds.), *State-of-the-Art Surveys in Computational Mechanics*, pages 71–143, 1989.
9. M. Nool and M.M.J. Proot. A parallel, state-of-the-art, least-squares spectral element solver for incompressible flow problems. In J.M.L.M. Palma, J.J. Dongarra, V. Hernández, A.A. de Sousa (eds.), *High Performance Computing for Computational Science – VECPAR 2002, Selected Papers and Invited Talks*, Vol. 2565 of *Lecture Notes in Computer Science*, pages 39–52, Springer-Verlag, Berlin, 2003.
10. M. Nool and M.M.J. Proot. Parallel implementation of a least-squares spectral element solver for incompressible flow problems. Submitted to *J. Supercomput.*, 2002.
11. M.M.J. Proot. The Least-Squares Spectral Element Method. Theory, Implementation and Application to Incompressible Flows. *PhD Thesis*, Delft, University of Technology, 2003.
12. M.M.J. Proot and M.I. Gerritsma. A least-squares spectral element formulation for the Stokes problem. *J. Sci. Comput.*, 17:311–322, 2002.
13. M.M.J. Proot and M.I. Gerritsma. Least-squares spectral elements applied to the Stokes problem. *J. Comput. Phys.*, 181:454–477, 2002.
14. M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, and J.J. Dongarra. *MPI: The Complete Reference.* MIT Press, Cambridge, Massachusetts, 1996.
15. Yousef Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, Boston(MA), 1996.
16. Yousef Saad. SPARSKIT: a basic tool-kit for sparse matrix computations (Version 2). `http://www.cs.umn.edu/research/arpa/SPARSKIT/sparskit.html`

Appendix A
List of test cases with their dimensions

	N	# SE	Interfaces	Interiors	*Knowns*
`The Netherlands`	2	676	8117	2704	1059
`The Netherlands`	4	676	18933	24336	2115
`Large cylinder`	4	86	2383	3096	553
`Large cylinder`	6	86	3747	8600	829
`Large cylinder`	8	86	5111	16856	1105
`Large cylinder`	10	86	6475	27864	1381
`Medium cylinder`	6	74	3219	7400	781
`Medium cylinder`	8	74	4391	14504	1041
`Small cylinder`	6	68	2955	6800	757
`Small cylinder`	8	68	4031	13328	1009
`Airfoil I`	2	584	7008	2336	568
`Airfoil I`	4	584	16352	21024	1136
`Airfoil I`	6	584	25696	58400	1704
`Airfoil II`	2	3692	44304	14768	1056
`Airfoil II`	4	3692	103376	132912	2112
`Stokes cylinder`	2	1432	17131	5728	829
`Stokes cylinder`	4	1432	39991	51552	1657

Appendix B
The data structure used for a spectral element

```
! define type NODE
   TYPE,PUBLIC :: node
      REAL(DOUBLE)                          :: X,Y
   END TYPE node
```

The `TYPE node` declaration statement is used to declare the x- and y-coordinates of a node in a 2D domain. It is also used within the `TYPE node` declaration statement:

```
! define type cell
   TYPE,PUBLIC :: cell
      INTEGER                               :: NUMBER
      INTEGER                               :: N1,N2
      TYPE(NODE),DIMENSION(0:3)             :: CORNERS
      TYPE(NODE),DIMENSION(:,:),POINTER     :: XYS
      INTEGER,DIMENSION(0:3)                :: SIDE
      REAL(DOUBLE)                          :: AREA
      REAL(DOUBLE),DIMENSION(:,:),POINTER :: DETERMINANT,DXIDX1,DXIDX2,&
      &                                      DETADX1,DETADX2
      LOGICAL,DIMENSION(0:3)                :: KNOWN_CORNER,READY_TO_COPY,&
      &                                      IC,IC_INIT,IC_COPY,&
      &                                      BC0,BC1,BC2,BC3,BC_END
      LOGICAL                               :: BOUNDARY
      INTEGER                               :: CELLFACE
      CHARACTER(LEN=32)                     :: CELLTYPE
   END TYPE cell
```

in which `NUMBER` denotes the spectral element number, $1 \leq$ `NUMBER` $\leq \mathcal{K}$. The `INTEGER`s `N1` and `N2` denote the polynomial order of the spectral element in x- and y-direction, respectively. Note that for unstructured grids those values must agree. The array `CORNERS` stands for the coordinates of the four corners/vertices of a spectral element, whereas `XYS` denotes the coordinates of the Gauss-Legendre points of that spectral element, including the interface points on edges and in vertices.

The entities `SIDE(0)`, `SIDE(1)`, `SIDE(2)` and `SIDE(3)` correspond to the spectral element number of the adjacent *below*, *right*, *above*, and *left* spectral element. Although we can not speak about *left* and *right* for unstructured grids, a certain order must be defined. If `SIDE(I)` $== 0$, for $0 \leq$ `I` ≤ 3, then `SIDE(I)` will be a boundary edge.

The entities `AREA`, `DETERMINANT`, `DXIDX1`, `DXIDX2`, `DETADX1` and `DETADX2` are used for the computation of the local systems. The `LOGICAL` arrays are used for the global numbering of the collocation points, such that a global number of a component in a collocation point agrees with the number of all spectral elements of which it is a part of. If `KNOWN_CORNER(I)` then the collocation points in the vertex between the edges `SIDE(I-1)` and `SIDE(I)` have received their global number (cf. capital **T** in Sect. 3.3), otherwise the points are not yet initialized and are still waiting for their correct value (cf. capital **F** in Sect. 3.3). If `READY_TO_COPY(I)` then all data in the collocation points on edge `I` are ready to copy to spectral element numbered `SIDE(I)` if that spectral element is at the same processor, otherwise they are ready to be sent by calling `MPI_BSEND`. The receiving processor will call `MPI_IPROBE` to check whether the data have arrived. By means of a call to `MPI_RECV` the data will be actually received and stored in the right location. The receiving processor *knows* exactly how much messages it will receive. For those people who are interested in `MPI` implementations, we remark that since a processor may be both on the buffered send side and on the receiving side, the communication

is not finished when all messages have been sent, but at the moment all messages have been received. In some cases an extra loop was required to empty the buffer between two processors.

The `LOGICAL` arrays `IC`, `IC_INIT` and `IC_COPY` are used in the special internal node cases illustrated in Fig. 2 and the arrays `BC0`, `BC1`, `BC2`, `BC3` and `BC_END` for the special boundary cases shown in Fig. 3. A global number will be given to a component of `CORNER(I)` if `IC_INIT(I)` and then it will be copied or sent to spectral element `SIDE(I)`, $0 \leq \mathtt{I} \leq 3$. Finally the values `BOUNDARY`, `CELLFACE` and `CELLTYPE` are used by the geometry to prescribe the boundary conditions on boundary spectral elements.