Institute for Advanced Simulation

# Strategies for Implementing Scientific Applications on Parallel Computers

Bernd Körfgen and Inge Gutheil

published in

*Multiscale Simulation Methods in Molecular Sciences*,
J. Grotendorst, N. Attig, S. Blügel, D. Marx (Eds.),
Institute for Advanced Simulation, Forschungszentrum Jülich,
NIC Series, Vol. **42**, ISBN 978-3-9810843-8-2, pp. 551-569, 2009.



http://www.fz-juelich.de/nic-series/volume42

# Strategies for Implementing Scientific Applications on Parallel Computers

**Bernd Körfgen and Inge Gutheil**

Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)
Forschungszentrum Jülich
52425 Jülich, Germany
*E-mail: {B.Koerfgen, I.Gutheil}@fz-juelich.de*

This contribution presents two examples for the numerical treatment of partial differential equations using parallel algorithms / computers. The first example solves the Poisson equation in two dimensions; the second partial differential equation describes the physical process of vibration of a membrane. Both problems are tackled with different strategies: The Poisson equation is solved by means of the simple Jacobi algorithm and a suitable parallelization scheme is discussed; in the second case the parallel calculation is performed with the help of the ScaLAPACK library and the issue of data distribution is addressed. Finally benchmarks of linear algebra libraries on the IBM Blue Gene/P architecture and for the PowerXCell processor are shown.

## 1 Motivation

The growth of processing power of computer systems observed through the last decades allowed scientists and engineers to perform more and more complex and realistic simulations. The driving force for this development was the exponentially increasing density of integrated circuits, e.g. processors, which can be empirically described by Moore's law. The key point of Moore's law is the observation that the circuit density of electronic devices doubles approximately every two years. The growing number of components together with the accompanying higher clock frequencies permitted to perform more intricate computations in shorter times.

Today this growth of circuit density and clock frequencies becomes more and more difficult to achieve. Furthermore the demand for compute power grows even faster (improved spatial / time resolution of models or the introduction of additional interactions / effects are required) and forces the developers of supercomputers to find new strategies to increase the available compute power. One approach which has become quite popular over the last two decades is the utilization of parallel computers. Many of nowadays parallel architectures use multi-core processors as building blocks in order to obtain the necessary compute power. Adding up ever more of these components generates the problem of excessive power consumption; not only to supply the electrical power but likewise to handle the produced heat are real challenges for the design of supercomputers.

Possible strategies to overcome the so-called 'power wall' are the reduction of clock frequencies and thus power consumption and the usage of special high performance processors which do more computations per clock cycle. The first concept has been realized with IBM's highly scalable **Blue Gene**[1] architecture. The latter strategy has been implemented by means of the **Cell processor**[2] which is together with Opteron processors the workhorse of the IBM Roadrunner[3] - the first supercomputer to reach **one Petaflop/s** ($10^{15}$

floating point operations per second). The price one has to pay for this development is the growing complexity of these heterogeneous systems.

The development of scientific applications on parallel computers, especially on highly scalable heterogeneous platforms poses a challenge to every scientist and engineer. The difficulties arising are to some extent problem specific and have to be resolved as the case arises assisted where possible by **parallel performance tools** like Scalasca[4]. Other tasks are generic and the application developer can resort to well established algorithms or even ready to use software solutions. Examples for these generic methods are **graph partitioning** algorithms for the decomposition of the problem (parallel load balancing) or **linear algebra** algorithms.

We will focus in the following on linear algebra because these algorithms are the core of many simulation codes and thus determine the efficiency and scalability of the whole application.

## 2 Linear Algebra

Numerical linear algebra is an active field of research which provided over the years many methods / algorithms for the treatment of standard problems like the solution of systems of linear equations, the factorization of matrices, the calculation of eigenvalues / eigenvectors etc.[5]. The most suitable algorithm for a given linear algebra problem, e.g. arising in a scientific application, has to be determined depending on the properties of the system / matrix (see for instance Ref. 6) like:

- **symmetry**
- **definiteness** (positive, negative, . . . )
- **non-zero structure** (dense, sparse, banded)
- **real** or **complex** coefficients

and so on. Furthermore the scientist has to decide whether to use a **direct** solver, leading to a transformation of the original matrix and thus (for large problems) generating a need for huge **main memory**, or to use an **iterative** solver which works with the original matrix.

The same rationale holds for the more specialized field of **parallel linear algebra** methods. There the additional aspects originating from the parallel **computer architecture** have to be taken into account in order to choose a suitable algorithm. Several topics influencing the choice and even more the consequent implementation of these algorithms are[7,8]:

- memory architecture (**shared-memory** vs. **distributed memory**)
- amount of **memory per process/processor**
- implemented **cache** structures

It is far beyond the scope of this contribution to give an overview of the available algorithms. Instead we refer to review articles like Refs. 9–11.

From a practical point of view another important decision is whether the user implements the linear algebra algorithm **himself** or relies on **available software / libraries**. A variety of well-known, robust packages providing high computational performance are on the market, which can be used as building blocks for an application software. Some **freely-available** libraries are:

- Basic Linear Algebra Subprograms **(BLAS)**[12]
- Linear Algebra Package **(LAPACK)**[13]
- Scalable LAPACK **(ScaLAPACK)**[14]
- **(P)ARPACK** - a (parallel) package for the solution of large eigenvalue problems[15]
- Portable, Extensible Toolkit for Scientific computation **(PETSc)**[16]

Some of them like BLAS or LAPACK are **serial** software, which help to gain good single processor performance, but leave the task of **parallelization** of the high-level linear algebra computations, e.g. solution of the coupled linear equations, to the user; others, e.g. ScaLAPACK or PARPACK, contain implementations of **parallel solvers**. Thus these packages relieve the user of the parallelization, but still they rely on special data distribution schemes[17] which require a specific organization of the application program. As a consequence the user has to handle the corresponding data distribution on his own, i.e. he has to parallelize his program at least partly. Nevertheless this might be a lot easier than to implement the full parallel linear algebra algorithm.

Since both strategies are preferable under certain circumstances, we will present in the following two simple physical problems where the **parallel numerical solution** will be demonstrated paradigmatically along the two different approaches:

In Section 3 the Poisson equation will be treated using a **parallel Jacobi solver** for the evolving system of linear equations.

In Section 4 the eigenvalue problem arising from the calculation of the vibration of a membrane is solved using a **ScaLAPACK routine**.

Of course, one would not use these solutions in real applications. Neither is the Jacobi algorithm a state-of-the-art method for the solution of a system of linear equations, nor is the eigensolver from ScaLAPACK the optimal choice for the given problem. Both examples result in a sparse matrix as will be shown in the following. ScaLAPACK contains solvers for full and banded systems, whereas (P)ARACK is a library based on the Arnoldi method which is very suitable for the calculation of a **few** eigenvalues for large **sparse** systems; thus (P)ARPACK would be the natural choice for this kind of problem.

Nevertheless due to the importance of ScaLAPACK for many application fields, e.g. **multiscale simulations**, and the simplicity of the Jacobi algorithm we present them as illustrative examples.

## 3 The Poisson Problem

In this section we discuss the numerical solution of the Poisson equation as an example for the approximate treatment of partial differential equations. We give a short outline of

the steps necessary to obtain a serial and later on parallel implementation of the numerical solver. Similar but more elaborate material on this topic can be found in the Refs. 18–20.

In a first step we discuss the discretization of the Poisson equation and introduce one simple solver for the evolving system of linear equations. Afterwards we focus on the parallelization of this algorithm.

### 3.1 Discretization of the Poisson Equation

The **Poisson equation** in two dimensions is given by

$$\Delta u \;=\; \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \;=\; f(x,y) \quad , \quad (x,y) \in \Omega \subset \mathbb{R}^2 \tag{1}$$

where $\Omega$ is a domain in $\mathbb{R}^2$. For simplicity $u(x,y)$ shall be given on the boundary $\partial\Omega$ by a **Dirichlet boundary condition**

$$u(x,y) = g(x,y) \quad , \quad (x,y) \in \partial\Omega \tag{2}$$

The functions $f(x,y)$ and $g(x,y)$ are given and $u(x,y)$ is to be calculated.

Since the analytic solution of such a partial differential equation might not be feasible depending on the shape of the domain, the functions $f, g$ etc., one often has to resort to the numerical solution of such a differential equation. In the following we will develop a simple scheme how to calculate $u$ approximately. For this we assume that the domain has a simple form: $\Omega$ is a rectangle (Figure 1). In order to determine the approximate solution

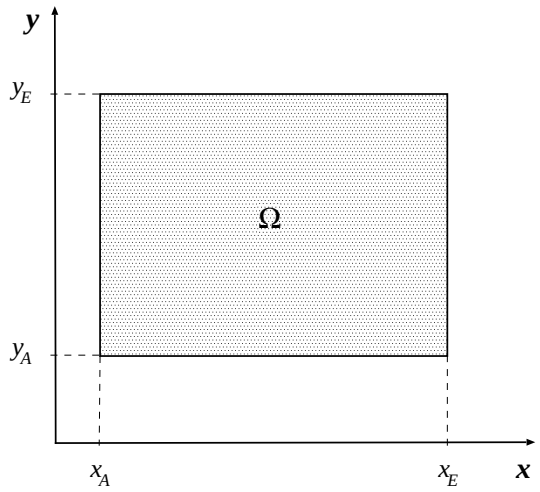


Figure 1. Rectangular domain in $\mathbb{R}^2$

of the Poisson equation, $u$ is calculated at certain points of the rectangle. We impose $\Omega = (x_A, x_E) \times (y_A, y_E)$ with an equidistant mesh (Figure 2), where $(x_A, x_E)$ is divided

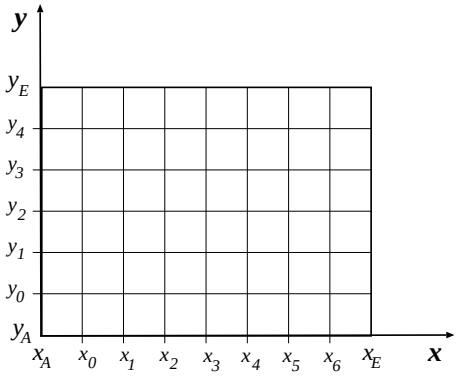


Figure 2. Mesh for $NI = 7$ and $NJ = 5$

into $(NI + 1)$ sub-intervals and $(y_A, y_E)$ into $(NJ + 1)$ sub-intervals, $(NI, NJ \in \mathbb{N})$. The mesh width $h$ is then given by

$$h \;=\; \frac{(x_E - x_A)}{(NI+1)} \;=\; \frac{(y_E - y_A)}{(NJ+1)} \tag{3}$$

With this choice for the mesh the approximate solution will be calculated at the $NI \cdot NJ$ inner points of the domain (The outer points don't have to be calculated, because they are given by the Dirichlet boundary condition!).

As a next step the second derivatives are replaced by finite differences. For this purpose we use the following Taylor expansions of $u$ at a point $(x, y)$:

$$u(x+h,y) = u(x,y) + hu_x(x,y) + \frac{h^2}{2!}u_{xx}(x,y) + \frac{h^3}{3!}u_{xxx}(x,y) \pm \dots \tag{4}$$

$$u(x-h,y) = u(x,y) - hu_x(x,y) + \frac{h^2}{2!}u_{xx}(x,y) - \frac{h^3}{3!}u_{xxx}(x,y) \pm \dots \tag{5}$$

Addition of both equations and division by $h^2$ gives

$$\frac{u(x-h,y) - 2u(x,y) + u(x+h,y)}{h^2} = u_{xx}(x,y) + O(h^2) \tag{6}$$

The result of the analogous procedure for the $y$-direction is

$$\frac{u(x,y-h) - 2u(x,y) + u(x,y+h)}{h^2} = u_{yy}(x,y) + O(h^2) \tag{7}$$

Using these finite differences the Poisson equation for the $NI \cdot NJ$ inner mesh points of the domain $\Omega$ is given by

$$\begin{gathered} u_{xx}(x_i,y_j) + u_{yy}(x_i,y_j) = f(x_i,y_j) \\ (i = 0,\dots,NI-1\;;\; j = 0,\dots,NJ-1) \end{gathered} \tag{8}$$

By neglecting the discretization error $O(h^2)$ Eqs. (8) can be written as:

$$u_{i,j-1} + u_{i-1,j} - 4u_{i,j} + u_{i,j+1} + u_{i+1,j} = h^2 f_{i,j} \tag{9}$$

for $i = 0, \ldots, NI-1\ ;\ j = 0, \ldots, NJ-1$ . The unknowns

$$u_{i,j} := u(x_i, y_j) \tag{10}$$

have to be calculated from the $NI \cdot NJ$ coupled linear equations (9).

The approximation used here for $u_{xx} + u_{yy}$ is called **5-point stencil** (Figure 3). The

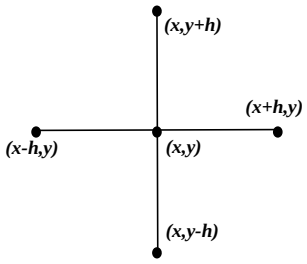


Figure 3. 5-point stencil

name describes the numerical dependency between the points of the mesh. The lexicographical numbering (Figure 4) of the mesh points

$$l = j \cdot NI + i + 1 \quad ; i = 0, \ldots, NI-1\ ;\ j = 0, \ldots, NJ-1 \tag{11}$$

and

$$u_l \ := \ u_{i,j} \tag{12}$$

allows a compact representation of the system of linear equations by means of a matrix.

The coefficient matrix $A$ is a block tridiagonal matrix:

$$A = \begin{pmatrix} A_1 & I & & & \\ I & A_2 & I & & \\ & \ddots & \ddots & \ddots & \\ & & I & A_{NJ-1} & I \\ & & & I & A_{NJ} \end{pmatrix} \in \mathbb{R}^{(NI \cdot NJ) \times (NI \cdot NJ)} \tag{13}$$

with $A_i, I \in \mathbb{R}^{NI \times NI}$ ; here $I$ is the unit matrix and

$$A_i = \begin{pmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -4 & 1 \\ & & & 1 & -4 \end{pmatrix} \quad i = 1, \ldots, NJ \tag{14}$$

This means the task to solve the Poisson equation numerically leads us to the problem to find the solution of a system of linear equations:

$$A\,\vec{u} \ = \ \vec{b} \tag{15}$$

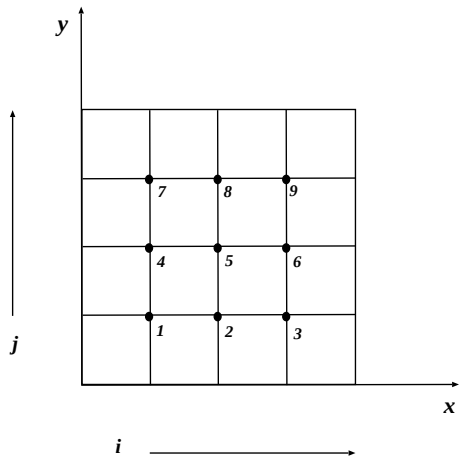


Figure 4. Lexicographical numbering of a $5 \times 5$ mesh (with $3 \times 3$ inner points)

with

$$A \in \mathbb{R}^{(NI \cdot NJ) \times (NI \cdot NJ)} \quad \text{and} \quad \vec{u}, \vec{b} \in \mathbb{R}^{(NI \cdot NJ)} \tag{16}$$

The right hand side $\vec{b}$ contains the $f_{i,j}$ of the differential equations as well as the Dirichlet boundary condition.

For the solution of these coupled linear equations many well-known numerical algorithms are available. We will focus here on the classic but very simple Jacobi algorithm.

### 3.2 The Jacobi Algorithm for Systems of Linear Equations

Suppose

$$A \;=\; D - L - U \tag{17}$$

is a decomposition of the matrix $A$, where $D$ is the diagonal sub-matrix, $-L$ is the strict lower triangular part and $-U$ the strict upper triangular part. Then for the system of linear equations holds

$$A\,\vec{u} = \vec{b} \quad \Leftrightarrow \quad (D - L - U)\,\vec{u} = \vec{b} \quad \Leftrightarrow \quad D\,\vec{u} = (L + U)\,\vec{u} + \vec{b} \quad \Leftrightarrow \tag{18}$$

$$\vec{u} \;=\; D^{-1}(L + U)\,\vec{u} + D^{-1}\,\vec{b} \quad \text{if } D^{-1} \text{ exists .} \tag{19}$$

From Eq. (19) follows the iteration rule (for $D$ non-singular)

$$\vec{u}^{\,(k)} = D^{-1}(L + U)\,\vec{u}^{\,(k-1)} + D^{-1}\,\vec{b} \quad \text{with } k = 1, 2, \ldots \tag{20}$$

This iterative procedure is known as **Jacobi** or **total-step method**. The second name is motivated by the fact that the next iteration is calculated **only** from the values of the unknowns of the last iteration. There are other schemes, e.g. Gauß-Seidel algorithm, which depend on old **and** the current iteration of the unknowns!
The corresponding pseudo code for the serial Jacobi algorithm is given here:

**Jacobi algorithm**

*Choose an initial vector* $\vec{u}^{\,(0)} \in \mathbb{R}^n$

*For* $k = 1, 2, \ldots$

*For* $i = 1, 2, \ldots, n$

$$u_i^{(k)} = \frac{1}{a_{ii}} \left( b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij}\, u_j^{(k-1)} \right)$$

The Poisson equation (9) discretized with the **5-point stencil** results in the following iteration procedure

$$\begin{pmatrix} u_1 \\ \vdots \\ \vdots \\ \vdots \\ u_N \end{pmatrix}^{(k)} = -\frac{1}{4} \begin{pmatrix} A'_1 & -I & & & \\ -I & A'_2 & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & A'_{NJ-1} & -I \\ & & & -I & A'_{NJ} \end{pmatrix} \begin{pmatrix} u_1 \\ \vdots \\ \vdots \\ \vdots \\ u_N \end{pmatrix}^{(k-1)} - \frac{1}{4} \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ \vdots \\ b_N \end{pmatrix} \tag{21}$$

with $N = NI \cdot NJ$ and

$$A'_i = \begin{pmatrix} 0 & -1 & & & \\ -1 & 0 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 0 & -1 \\ & & & -1 & 0 \end{pmatrix} \quad i = 1, \ldots, NJ \tag{22}$$

This can be seen easily by application of the Jacobi matrix decomposition on the coefficient matrix given by Eqs. (13) and (14). The pseudo code for this special case is shown here

**Jacobi algorithm for the Poisson equation**

*Choose initial vector* $\vec{u}^{\,(0)} \in \mathbb{R}^N$

*For* $k = 1, 2, \ldots$

*For* $j = 0, 1, \ldots, NJ - 1$

*For* $i = 0, 1, \ldots, NI - 1$

$$u_{i,j}^{(k)} = \frac{1}{4} \left( u_{i,j-1}^{(k-1)} + u_{i-1,j}^{(k-1)} + u_{i,j+1}^{(k-1)} + u_{i+1,j}^{(k-1)} - h^2 f_{i,j} \right)$$

### 3.3 Parallelization of the Jacobi Algorithm

The numerical treatment of the Poisson equation led us to the task to solve a system of linear equations. We introduced the Jacobi algorithm as a simple method to calculate this solution and presented the corresponding serial pseudo code. Now the next step is to discuss strategies **how to implement the Jacobi algorithm on a parallel computer**.

The important point about the Jacobi (total-step) algorithm one has to remember is that the calculation of the new iteration only depends on the values of the unknowns from the **last iteration** as can be seen for instance from Eq. (21). As a consequence the processors of a parallel computer can calculate the new iteration of the unknowns simultaneously, supposed each unknown is assigned to its own processor. This makes the parallelization of the Jacobi algorithm quite easy compared to other methods with more complicated dependencies between different iterations.

Usually the number of unknowns is much larger than the number of available processors. Thus some / many unknowns have to be assigned to one processor, i.e. for our example: the inner points of $\Omega$ (Figure 1) are distributed to the available processors. With other words the **domain** $\Omega$ is **decomposed** according to a suitable strategy.
The criteria for a "suitable" strategy are

- **load balance**, i.e. same / similar number of unknowns for each processor
- minimization of the **communication** between the processors, i.e. the dependency on unknowns stored on other processors (within one iteration step!) is reduced

For our example, the Poisson equation in two dimensions, a reasonable domain decomposition is shown in Figure 5: Each processor "owns" a domain of the same size, i.e. each

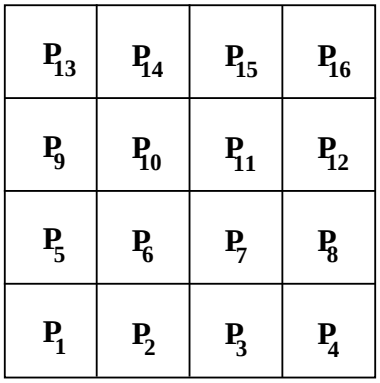


Figure 5. Domain decomposition of a square $\Omega$ with 16 processors

$P_i$ "owns" the same number of points. Furthermore the ratio area to edges of each square and consequently the ratio between the number of inner points (no dependency on points "owned" by other processors) to the number of points near the boundary is rather good. This point can been seen even better from Figure6.

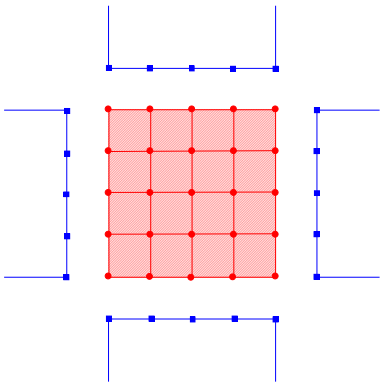

Figure 6. Dependency of the unknowns of processor $P_i$ (red) on values stored on the neighbors (blue)

In Fig. 6 the points / corresponding unknowns of processor $P_i$ are represented by red circles, whereas the blue squares depict the **ghost points**, i.e. points stored on other processors which are required for the calculation of the next iteration of the unknowns on processor $P_i$.

The dependencies / ghost points shown in Fig. 6 are a result of the 5-point stencil (see Fig. 3) originating from the Laplace operator in Eq. (1). Thus the domain decomposition of choice might differ for other differential equations or other discretization schemes, e.g. finite elements.

Due to the dependencies between the unknowns "owned" by different processors it is clear that the parallelization of the Jacobi algorithm has to introduce statements which will take care of the communication between the processors. One portable way to handle the communication is the widely used **Message Passing Interface (MPI)**[21] library.
The pseudo code of the parallel Jacobi algorithm is given here:

**Parallel Jacobi algorithm**

*Choose initial values for the* **own mesh points** *and the* **ghost points**

*Choose initial Precision* (*e.g. Precision* $= 10^{10}$)

*While Precision* $> \varepsilon$ (*e.g.* $\varepsilon = 10^{-5}$)

1. *Calculate next iteration for the* **own domain**
2. *Send the new iteration* **on boundary of domain** *to* **neighboring processors**
3. *Receive the new iteration for the* **ghost points**
4. *Calculate Precision* $= \|A\vec{u}^{(k)} - \vec{b}\|$

*End While*

The steps 2 and 3 show the extension of the serial Jacobi algorithm by *Send* and *Receive* statements. This is of course only one very simple way to implement such a communication with the four neighboring processors. In real applications one will look for more efficient communication patterns.

Step 4 requires implicitly **global communication**, because the vector $\vec{u}^{(k)}$ holding the approximate solution of the system of linear equations is distributed over all processors. As soon as the required precision of the solution is achieved the iteration stops.

## 4 Vibration of a Membrane

The vibration of a homogeneous membrane is governed by the time-dependent partial differential equation[22]

$$\frac{\partial^2 v}{\partial t^2} \;=\; \Delta\, v \tag{23}$$

In order to solve this equation we make a separation ansatz for the time and spatial variables:

$$v(x,y,t) \;=\; u(x,y)\, g(t) \tag{24}$$

By insertion of Eq. (24) into Eq. (23) one immediately obtains

$$g(t)\, \Delta\, u(x,y) \;=\; u(x,y)\, g''(t) \quad \Leftrightarrow \tag{25}$$

$$\frac{\Delta\, u(x,y)}{u(x,y)} \;=\; \frac{g''(t)}{g(t)} \tag{26}$$

The left side of Eq. (26) is independent of $t$, the right side of $x, y$. Therefore both sides must be equal to a constant $-\lambda$

$$\frac{\Delta\, u(x,y)}{u(x,y)} \;=\; \frac{g''(t)}{g(t)} \;=\; -\lambda \quad \Leftrightarrow \tag{27}$$

$$\Delta\, u(x,y) \;=\; -\lambda\, u(x,y) \quad \text{and} \quad g''(t) \;=\; -\lambda\, g(t) \tag{28}$$

The differential equation for $g(t)$ can be solved easily with the usual ansatz (a linear combination of trigonometric functions).

In the following we want to solve the spatial partial differential equation

$$\Delta\, u(x,y) \;=\; -\lambda\, u(x,y) \tag{29}$$

numerically. In section 3.1 we presented the discretization of the Poisson equation in two dimensions. In order to allow a re-use of the results derived there, we will calculate the solution of Eq. (29) for a **rectangular** membrane / domain.

Furthermore we choose for simplicity the Dirichlet boundary condition

$$u(x,y) \;=\; 0 \quad \text{for } (x,y) \in \partial\Omega \tag{30}$$

Using the same discretization for the Laplace operator and lexicographical numbering of the mesh points / unknowns as in section 3.1 one can see easily that Eq. (29) leads to the **eigenvalue problem**

$$A\, \vec{u} \;=\; -\lambda\, \vec{u} \tag{31}$$

where the matrix $A$ is given by Eqs. (13) and (14).

In section 3 we presented a simple algorithm for the solution of the system of linear equations and discussed the parallelization by hand. For the eigenvalue problem we choose a different strategy: We make use of a widely used parallel library, namely the **ScaLAPACK** library.

### 4.1 Parallel Solution Using the ScaLAPACK Library

The largest and most flexible public domain library with linear algebra routines for distributed memory parallel systems up to now is ScaLAPACK[14]. Within the ScaLAPACK project many LAPACK routines were ported to distributed memory computers using MPI.

The basic routines of ScaLAPACK are the **PBLAS** (Parallel Basic Linear Algebra Subroutines)[23]. They contain parallel versions of the BLAS which are parallelized using **BLACS** (Basic Linear Algebra Communication Subprograms)[24] for communication and sequential BLAS for computation. Thus the PBLAS deliver very good performance on most parallel computers.

ScaLAPACK contains direct parallel solvers for dense linear systems (LU and Cholesky decomposition), linear systems with band matrices as well as parallel routines for the solution of linear least squares problems and for singular value decomposition.

Furthermore there are several different routines for the solution of the full symmetric eigenproblem. We will focus in the following on a **simple driver routine** using the QR-algorithm, which computes all eigenvalues and optionally all eigenvectors of the matrix.

Besides this there are other eigensolvers available which are implementations of other algorithms, e.g. a divide-and-conquer routine; an additional expert driver allows to choose a range of eigenvalues and optionally eigenvectors to be computed.

For performance and load balancing reasons ScaLAPACK uses a **two-dimensional block cyclic distribution** for full matrices (see ScaLAPACK Users' Guide)[17]:
First the matrix is divided into blocks of size MB $\times$ NB, where MB and NB are the number of rows and columns per block, respectively. These blocks are then uniformly distributed across the MP $\times$ NP **rectangular processor grid** in a cyclic manner. As a result, each process owns a collection of blocks. Figure 7 shows the distribution of a $(9 \times 9)$ matrix subdivided into blocks of size $(3 \times 2)$ distributed across a $(2 \times 2)$ processor grid.

| | 0 | | 1 | | 0 | | 1 | | 0 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | $a_{11}$ | $a_{12}$ | $a_{13}$ | $a_{14}$ | $a_{15}$ | $a_{16}$ | $a_{17}$ | $a_{18}$ | $a_{19}$ |
| | $a_{21}$ | $a_{22}$ | $a_{23}$ | $a_{24}$ | $a_{25}$ | $a_{26}$ | $a_{27}$ | $a_{28}$ | $a_{29}$ |
| | $a_{31}$ | $a_{32}$ | $a_{33}$ | $a_{34}$ | $a_{35}$ | $a_{36}$ | $a_{37}$ | $a_{38}$ | $a_{39}$ |
| 1 | $a_{41}$ | $a_{42}$ | $a_{43}$ | $a_{44}$ | $a_{45}$ | $a_{46}$ | $a_{47}$ | $a_{48}$ | $a_{49}$ |
| | $a_{51}$ | $a_{52}$ | $a_{53}$ | $a_{54}$ | $a_{55}$ | $a_{56}$ | $a_{57}$ | $a_{58}$ | $a_{59}$ |
| | $a_{61}$ | $a_{62}$ | $a_{63}$ | $a_{64}$ | $a_{65}$ | $a_{66}$ | $a_{67}$ | $a_{68}$ | $a_{69}$ |
| 0 | $a_{71}$ | $a_{72}$ | $a_{73}$ | $a_{74}$ | $a_{75}$ | $a_{76}$ | $a_{77}$ | $a_{78}$ | $a_{79}$ |
| | $a_{81}$ | $a_{82}$ | $a_{83}$ | $a_{84}$ | $a_{85}$ | $a_{86}$ | $a_{87}$ | $a_{88}$ | $a_{89}$ |
| | $a_{91}$ | $a_{92}$ | $a_{93}$ | $a_{94}$ | $a_{95}$ | $a_{96}$ | $a_{97}$ | $a_{98}$ | $a_{99}$ |

Figure 7. Block cyclic 2D distribution of a $(9 \times 9)$ matrix subdivided into $(3 \times 2)$ blocks on a $(2 \times 2)$ processor grid. The numbers outside the matrix indicate processor row and column indices, respectively.

ScaLAPACK as a parallel successor of LAPACK attempts to leave the calling sequence of the subroutines unchanged as much as possible in comparison to the corresponding sequential subroutine from LAPACK. The user should have to change only a few parameters in the calling sequence to use ScaLAPACK routines instead of LAPACK routines.

Therefore ScaLAPACK uses so-called **descriptors**, which are integer arrays containing all necessary information about the **distribution of the matrix**. This descriptor appears in the calling sequence of the parallel routine instead of the leading dimension of the matrix in the sequential one.

For example the sequential simple driver **DSYEV** from LAPACK for the computation of **all eigenvalues** and (optionally) eigenvectors of a **real symmetric** ($N \times N$) matrix A has the following calling sequence[25]:

```
...
CALL DSYEV(JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, INFO)
...
```

where JOBZ and UPLO are characters indicating whether to compute eigenvectors, and whether the lower or the upper triangular part of the matrix A is provided. LDA is the leading dimension of A and W is the array of eigenvalues of A. The other variables are used as workspace and for error report.

The corresponding ScaLAPACK routine **PDSYEV** is called as follows[17]:

```
...
      CALL PDSYEV ( JOBZ, UPLO, N, A, IA, JA, DESCA,
     $              W, Z, IZ, JZ, DESCZ, WORK, LWORK, INFO )
...
```

As one can see the leading dimension LDA of the LAPACK call is substituted by the indices IA and JA and the descriptor DESCA. IA and JA indicate the start position of the **global matrix** (usually IA, JA = 1, but in cases where the global matrix is a sub-matrix of a larger matrix IA, JA $\neq$ 1 might occur) , whereas DESCA contains all information regarding the distribution of the global matrix. The parameters IZ, JZ, and DESCZ provide the same information for Z, the matrix of the eigenvectors calculated by PDSYEV.

In order to use the ScaLAPACK routine the user has to distribute his system matrix in the way required by ScaLAPACK. Thus the user has to setup the **processor grid** by initializing MP, the number of processor rows, and NP, the number of processor columns. Furthermore one has to choose a suitable blocking of the matrix, i.e. MB and NB. For many routines, especially for the eigenvalue solvers and the Cholesky decomposition, MB=NB is mandatory. (Since MB and NB are crucial for the performance of the solver, one has to use these parameters with care.[26]) Further details on the two-dimensional block cyclic distribution of the matrix A given by Eqs. (13) and (14) can be found in the appendix.

Once the matrix has been distributed to the processors, the calculation of the eigenvalues and corresponding eigenvectors for the vibration of the rectangular membrane (Eq. (31)) can be calculated easily by one call of the routine **PDSYEV**. Please note that the matrix of the eigenvectors Z, is distributed to the processors; thus if necessary, e.g. for output, it is again the task of the user to collect the different local data and to generate the global matrix.

## 5 Performance of Parallel Linear Algebra Libraries

The performance, i.e. scalability, of parallel libraries and the availability of optimized implementations for specific hardware platforms are major criteria for the selection of (linear algebra) functions to be used within a scientific application. Especially the availability of optimized software differs largely for different architectures: for distributed memory systems using MPI many (non-)numerical libraries are freely-available or are provided by the vendors. For new and non-standard hardware like the **PowerXCell 8i processor**[27] the situation is not that comfortable. This can reduce the usability largely particularly for the Cell processors which have to be programmed in **assembler code style**.

At present a **BLAS** library, which can be used just like its serial counterpart and hides the complexity of the parallel hardware from the application developer[28], exists for the PowerXCell architecture whereas other linear algebra libraries are still under development. Figure 8 shows the processing power of a PowerXCell 8i processor in Gflops (= $10^9$ Floating point operations per second) for a **double precision** matrix-matrix multiplication (BLAS routine **DGEMM**) as function of the matrix size. The results are shown for calculations where 2, 4 and 8 Synergistic Processing Elements **(SPE)**[27] are used; the Cell processor contains 8 SPEs which have a theoretical compute power of 12.8 Gflops each (3.2 GHz, 4 flops per cycle).

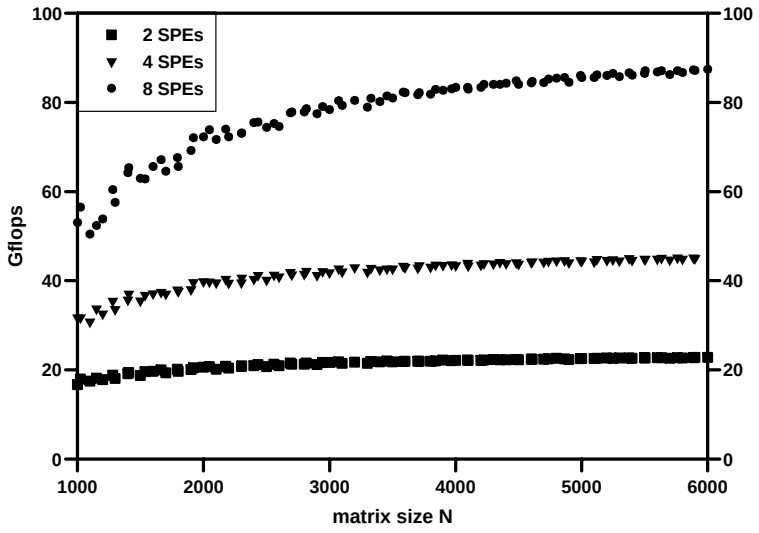


Figure 8. Processing power [Gflops] of a double precision matrix-matrix multiplication as function of the matrix size for a **PowerXCell processor**. Results are shown for different numbers of SPEs: 2, 4 and 8 SPEs.

From Figure 8 one learns that the sustained performance of a PowerXCell processor for the double precision matrix-matrix multiplication is approximately 90 Gflops and thus more than 85% of the theoretical peak performance (102.4 Gflops). Furthermore one sees that the number of computations per second scales well with the number of SPEs and is quite independent of the problem size. The recent multi-core processor **Intel Core i7**

(quad-core) has a theoretical compute power of 51.2 Gflops; the PowerXCell shows twice the performance in a real computation but for a substantially lower price and electrical power consumption.

A complementary strategy to build supercomputers is to add huge numbers of low-cost, regarding price as well as power consumption, processors to gain high compute power. This concept has been implemented for instance with IBM's Blue Gene/P where each processor core supplies 'only' 3.4 Gflops (850 MHz, 4 flops per cycle) and has a local memory of 512 MByte. But the complete Blue Gene/P system **JUGENE**[29] at JSC with 65536 cores accomplishes more the 220.000 Gflops and has a main memory of 128 TByte (= 131.072 GBytes).

Figure 9 gives the performance of JUGENE for the parallel double precision matrix-matrix multiplication **PDGEMM** from the ScaLAPACK library using 256, 1024 and 4096 of its processors. Obviously the real compute power as well as its scaling with increasing

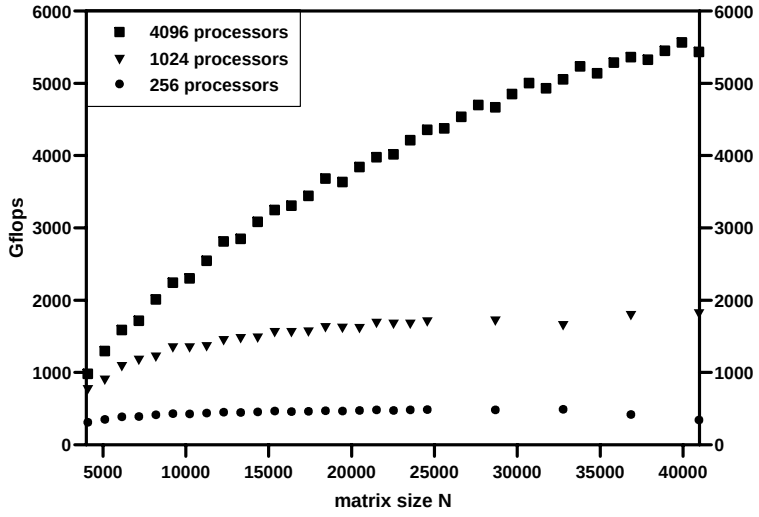


Figure 9. Processing power [Gflops] of a double precision matrix-matrix multiplication as function of the matrix size on JSC's **Blue Gene/P** system. Results are shown for different processor numbers: 256, 1024 and 4096 processors.

processor number depends largely on the problem size. This is a general observation on massively parallel computer architectures: Most algorithms show a much better parallel performance if the problem sizes increase / scale together with the number of processors employed - a behaviour known as **weak scaling** and foreseen by Gustafson[30] in 1988.

A computation rate of approximately 5.500 Gflops is shown in Figure 9 for a matrix size of 40.000 and 4096 processors. This result is only about $40\%$ of the theoretical peak performance of the 4096 processors and the parallel speedup for increasing processor numbers is far from optimal, nevertheless it illustrates the potential of supercomputers like the IBM Blue Gene/P with several ten thousand to hundreds of thousands of processors for scientific applications. With these results in mind it is no surprise that the combination of

a massively parallel system and special high performance processors allowed to enter the era of **Petaflop computing** with the **IBM Roadrunner**[3].

## 6 Conclusion

In this contribution we presented two examples for the numerical treatment of partial differential equations using parallel algorithms / computers. Both problems were tackled with different strategies: The first example has been solved by means of the simple Jacobi algorithm and a suitable parallelization scheme was discussed. In the second case the parallel calculation has been performed with the help of the ScaLAPACK library.

The pros and cons of the different strategies are obvious. If a suitable parallel library is available and a reorganization of the application software according to the complex data distribution schemes of the libraries is possible, the parallel routines from the library will provide a robust numerical solution with fair or even good performance. Otherwise the user has to choose a parallelization scheme which best fits his specific application problem and he has to implement the necessary algorithms himself; in order to improve the single processor performance it is still recommendable to use serial library routines, e.g. from BLAS or LAPACK, wherever possible!

Benchmarks of a parallel linear algebra routine were shown for the IBM Blue Gene/P architecture and for the PowerXCell processor. The results demonstrate the compute power of special purpose processors as well as the potential of massively parallel computers.

## Appendix

In section 4.1 some information on the **two-dimensional block cyclic distribution** of the data used by **ScaLAPACK** has been given. In this appendix we will discuss this issue in greater detail.

In Fig. 10 a code fragment is shown which distributes the N $\times$ N matrix given by Eqs. (13) and (14) according to the ScaLAPACK data scheme with block sizes NB=MB to an MP $\times$ NP processor grid. Inclusion of this fragment into a parallel program allows the calculation of the eigenvalues and eigenvectors using the routine **PDSYEV**:

```
...
      CALL PDSYEV ( JOBZ, UPLO, N, A, IA, JA, DESCA,
     $              W, Z, IZ, JZ, DESCZ, WORK, LWORK, INFO )
...
```

Notice that in the sequential as well as in the parallel routine the matrix A is destroyed. The difference is that in the sequential case if the eigenvectors are requested A is overwritten by the eigenvectors whereas in the parallel case the eigenvectors are stored to a separate matrix Z.

The matrix Z has to be allocated with the same local sizes as A and DESCZ is filled with the same values as DESCA. The size LWORK of the local workspace WORK can be found in the ScaLAPACK Users' Guide.

```
! Create the MP * NP processor grid
CALL BLACS_GRIDINIT(ICTXT,'Row-major',MP,NP)
! Find my processor coordinates MYROW and MYCOL
! NPROW returns the same value as MP,
! NPCOL returns the same value as NP
CALL BLACS_GRIDINFO(ICTXT,NPROW,NPCOL,MYROW,MYCOL)
! Compute local dimensions with routine NUMROC from TOOLS
! N is dimension of the matrix
! NB is block size
MYNUMROWS = NUMROC(N,NB,MYROW,0,NPROW)
MYNUMCOLS = NUMROC(N,NB,MYCOL,0,NPCOL)
! Local leading dimension of A,
! Number of local rows of A
MXLLDA = MYNUMROWS
! Allocate only the local part of A
ALLOCATE(A(MXLLDA,MYNUMCOLS))
! Fill the descriptors, P0 and Q0 are processor coordinates
! of the processor holding global element A(1,1)
CALL DESCINIT(DESCA,N,N,NB,NB,P0,Q0,ICTXT,MXLLDA,INFO)
! Fill the local part of the matrix with data
do j = 1, MYNUMCOLS, NB        ! Fill the local column blocks
  do jj=1,min(NB,MYNUMCOLS-j+1)   ! all columns of one block
    jloc = j-1 + jj
! local column index
    jglob = (j-1)*NPCOL + MYCOL*NB +jj ! global column index
    do i = 1, MYNUMROWS, NB      ! local row blocks in column
      do ii=1,min(NB,MYNUMROWS-i+1)
! rows in row block
        iloc = i-1 + ii
! local row index
        iglob = (i-1)*NPROW + MYROW*NB+ii ! global row index
        A(iloc,jloc) = 0
        If (iglob==jglob) A(iloc,jloc)=-4
        If (iglob==jglob+1.and.mod(jglob,NI)/=0) &
                                          A(iloc,jloc)=1
        If (jglob==iglob+1.and.mod(iglob,NI)/=0) &
                                          A(iloc,jloc)=1
        If (iglob==jglob+NI) A(iloc,jloc)=1
        If (jglob==iglob+NI) A(iloc,jloc)=1
      enddo
    enddo
  enddo
enddo
```

Figure 10. Code fragment which distributes the matrix given by Eqs. (13) and (14) according to ScaLAPACK (It is assumed that MB=NB=NI and N=NI·NJ).

The four nested loops in Fig. 10 show how local and global indices can be computed from block sizes, the number of rows and columns in the processor grid and the processor coordinates. The conversion of global to local indices and vice versa is supported by some auxiliary routines in the **TOOLS** sub-library of ScaLAPACK. Here the routine NUMROC is used to calculate the number of rows / columns stored on the corresponding processor.

There is also a sub-library **REDIST** of ScaLAPACK which allows the redistribution of any two-dimensional block cyclically distributed matrix to any other block cyclic two-dimensional distribution. Thus if A was column cyclically distributed or if the eigenvectors have to be column cyclically distributed for further computations they can be redistributed by such a routine, as a column cyclic distribution is nothing else but a block cyclic two-dimensional distribution to a $1 \times$ NPR (with NPR = number of processors) grid with block size 1.

## References


1. IBM System Blue Gene/P
   `http://www-03.ibm.com/systems/deepcomputing/bluegene/`
2. IBM Cell Broadband Engine technology
   `http://www-03.ibm.com/technology/cell/`
3. Los Alamos National Lab's Supercomputer Roadrunner
   `http://www.lanl.gov/roadrunner/`
4. Scalable Performance Analysis of Large-Scale Applications (Scalasca)
   `http://www.fz-juelich.de/jsc/scalasca/`
5. J. J. Dongarra, I. S. Duff, D. .C. Sorensen, and H. A. van der Vorst, *Numerical Linear Algebra for High-Performance Computers*, SIAM, Philadelphia (1998).
6. `http://gams.nist.gov/Classes.html` and especially
   `http://gams.nist.gov/serve.cgi/Class/D/`
7. B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications using networked Workstations and Parallel Computers*, Pearson, Upper Saddle River (2005).
8. A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, Pearson, Harlow (2003).
9. M. Bücker, *Iteratively Solving Large Sparse Linear Systems on Parallel Computers* in Quantum Simulation of Complex Many-Body Systems: From Theory to Algorithms, J. Grotendorst et al. (Ed.), John von Neumann Institute for Computing, NIC Series Vol. **10**, 521-548 (2002)
   `http://www.fz-juelich.de/nic-series/volume10/buecker.pdf`
10. B. Lang, *Direct Solvers for Symmetric Eigenvalue Problems* in Modern Methods and Algorithms of Quantum Chemistry, J. Grotendorst (Ed.), John von Neumann Institute for Computing, NIC Series Vol. **3**, 231-259 (2000).
    `http://www.fz-juelich.de/nic-series/Volume3/lang.pdf`
11. B. Steffen, *Subspace Methods for Sparse Eigenvalue Problems* in Modern Methods and Algorithms of Quantum Chemistry, J. Grotendorst (Ed.), John von Neumann Institute for Computing, NIC Series Vol. **3**, 307-314 (2000).
    `http://www.fz-juelich.de/nic-series/Volume3/steffen.pdf`

12. Basic Linear Algebra Subprograms (BLAS)
http://www.netlib.org/blas/
13. Linear Algebra Package (LAPACK)
http://www.netlib.org/lapack/
14. Scalable Linear Algebra Package (ScaLAPACK)
http://www.netlib.org/scalapack/
15. http://www.caam.rice.edu/software/ARPACK/
16. Portable, Extensible Toolkit for Scientific computation (PETSc)
http://www-unix.mcs.anl.gov/petsc/petsc-as/
17. L. S. Blackford, J. Choi, A. Cleary et al., *ScaLAPACK Users' Guide*, SIAM, Philadelphia (1997).
18. G. Ahlefeld, I. Lenhardt, and H. Obermaier, *Parallele numerische Algorithmen*, Springer, Berlin (2002).
19. J. M. Ortega, *Introduction to parallel and vector solution of linear systems*, Plenum Press, New York (1988).
20. J. Zhu, *Solving partial differential equations on parallel computers*, World Scientific, Singapore (1994).
21. W. Gropp, E. Lus,k and A. Skjellum, *Using MPI : Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge (1994).
22. R. Courant and D. Hilbert, *Methods of Mathematical Physics*, Volume I, Interscience Publishers, New York (1953).
23. Parallel Basic Linear Algebra Subprograms (PBLAS)
http://www.netlib.org/scalapack/pblas_qref.html
24. J. J. Dongarra and R. C. Whaley, *A User's Guide to the BLACS*, LAPACK Working Note **94** (1997).
http://www.netlib.org/lapack/lawns/lawn94.ps
25. E. Anderson, Z. Bai, C. Bischof et al., *LAPACK Users' Guide, Second Edition*, SIAM, Philadelphia (1995).
26. I. Gutheil, *Basic Numerical Libraries for Parallel Systems* in Modern Methods and Algorithms of Quantum Chemistry, J. Grotendorst (Ed.), John von Neumann Institute for Computing, NIC Series Vol. **3**, 47-65 (2000).
http://www.fz-juelich.de/nic-series/Volume3/gutheil.pdf
27. IBM PowerXCell 8i processor datasheet
http://www-03.ibm.com/technology/resources/
28. IBM Cell Broadband Engine - Software Development Kit (SDK) 3.1
http://www-128.ibm.com/developerworks/power/cell/index.html
29. Blue Gene/P at JSC
http://www.fz-juelich.de/jsc/jugene/
30. John L. Gustafson, *Reevaluating Amdahl's law*, Communications of the ACM **31**, 532–533, 1988.