# Parallel Computing

X. Cai[1,2], E. Acklam[3], H. P. Langtangen[1,2], and A. Tveito[1,2]

[1] Simula Research Laboratory
[2] Department of Informatics, University of Oslo
[3] Numerical Objects AS

**Abstract.** Large-scale parallel computations are more common than ever, due to the increasing availability of multi-processor systems. However, writing parallel software is often a complicated and error-prone task. To relieve Diffpack users of the tedious and low-level technical details of parallel programming, we have designed a set of new software modules, tools, and programming rules, which will be the topic of the present chapter.

## 1 Introduction to Parallel Computing

Parallel computing, also known as concurrent computing, refers to a group of independent processors working collaboratively to solve a large computational problem. This is motivated by the need to reduce the execution time and to utilize larger memory/storage resources. The essence of parallel computing is to partition and distribute the entire computational work among the involved processors. However, the hardware architecture of any multi-processor computer is quite different from that of a single-processor computer, thus requiring specially adapted parallel software. Although the message passing programming model, especially in terms of the MPI standard [3], promotes a standardized approach to writing parallel programs, it can still be a complicated and error-prone task. To reduce the difficulties, we apply object-oriented programming techniques in creating new software modules, tools, and programming rules, which may greatly decrease the user effort needed in developing parallel code. These issues will be discussed in this chapter within the framework of Diffpack.

The contents of this chapter are organized as follows. The present section contains a brief introduction to some important concepts related to parallel computing in general. Section 2 proposes a performance model that has advantages in analyzing and predicting CPU consumptions by complex parallel computations. Then, Section 3 is devoted to an introduction to the basic functionality of the MPI standard. Later on, Section 4 presents a high-level Diffpack interface to the MPI routines, whereas Section 5 concerns parallelizing Diffpack simulators that use explicit finite difference schemes. Finally, Section 6 covers the topic of parallelizing implicit finite element computations on unstructured grids.

## 1.1 Different Hardware Architectures

The hardware architectures of different multiple-processor systems span a wide spectrum. We may categorize multiple-processor systems from the angle of memory configuration. At one extreme, there are the so-called *shared memory* parallel computers. The most important feature of such parallel computers is that all the processors share a single global memory space, which is realized either at the hardware level or at the software level. At the other extreme of the hardware architecture spectrum, there are the so-called *distributed memory* parallel computers, where each processor has its own individual memory unit. All the processors are then connected through an interconnection network, by which communication between processors takes place. Using the network speed as a main criterion, we can further divide the distributed memory systems into sub-categories such as high-end massive parallel computers, low-cost PC clusters, networks of workstations, etc. Finally, in the middle of the hardware architecture spectrum, we find *distributed shared-memory* machines, which are in fact clusters of shared-memory parallel computers.

## 1.2 The Message-Passing Programming Model

The different types of hardware architecture have their impact on parallel software programming. Two major parallel programming models are thus so-called *shared memory* and *message passing* models, which arise from the shared memory and distributed memory hardware architectures, respectively. The shared memory programming model assumes that every process can directly access the global data structure of a parallel program. Each process is implicitly assigned to work on a portion of the global data. Moreover, work load partitioning and inter-process communication are hidden from the user. This programming model is well suited for static and regular data structures, leading normally to parallel execution at the loop level.

In the message passing model, however, it is necessary to partition a global problem into several smaller sub-problems and assign them to different processes. Each process has direct access only to its sub-problem data structure. In order for the processes to solve the global problem collaboratively, the user has to insert communication commands in the parallel program. Collaboration between the processes is achieved by *explicit message passing*, i.e., different processes communicate with each other in form of sending and receiving messages. We remark that message passing enables not only data transfer, but also synchronization, which refers to waiting for the completion of the involved processes, at certain pre-determined points during execution. As the message passing model is more flexible and can also be realized on any shared memory machine, we focus on this programming model. It has been applied in the development of the parallel Diffpack libraries, and will be assumed throughout the rest of this chapter.

*Remark.* Strictly speaking, the term *process* is different from the term *processor.* A process is an instance of a program executing autonomously, see e.g. [8, Ch. 2.2], whereas a processor is a hardware unit of a computer on which one or several processes can be executed. However, assuming the most common situation where only one process within a parallel computation is executed on one processor, we loosely use these two terms interchangeably in the rest of this chapter.

### 1.3 A Multicomputer Model

To simplify the description and analysis of the forthcoming parallel algorithms, we adopt the following model of a theoretical *multicomputer* (see e.g. [11, Ch. 1.2]), which is in fact an idealized model of distributed memory systems.

> A multicomputer has $P$ identical processors, where every processor has its own memory. Each processor has control of its own local computation, which is carried out sequentially. The processors are interconnected through a communication network, so that any pair of two processors can exchange information in form of sending and receiving messages. Moreover, communication between any two processors is of equal speed, regardless of physical distance.

### 1.4 Two Examples of Parallel Computing

*Addition of Two Vectors.* Addition of two vectors is a frequently encountered operation in scientific computing. More precisely, we want to compute

$$\mathbf{w} = a\mathbf{x} + b\mathbf{y},$$

where $a$ and $b$ are two real scalars and the vectors $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{w}$ are all of length $M$. In order to carry out the vector addition in parallel, it is necessary to first partition $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{w}$ among the processors. That is, each processor creates its local storage to hold a portion of every global vector. Using an integer $p$, which has value between 0 and $P-1$, as the processor rank on a $P$-processor multicomputer, we will have for processor number $p$:

$$\mathbf{x}_p \in \mathbb{R}^{M_p}, \quad \mathbf{y}_p \in \mathbb{R}^{M_p}, \quad \mathbf{w}_p \in \mathbb{R}^{M_p}, \quad M = \sum_p M_p \,.$$

The above partitioning means that for every global index $i = 1, 2, \ldots, M$, there exists an index pair $(p, j)$, $0 \leq p < P$, $1 \leq j \leq M_p$, such that the correspondence between $i$ and $(p, j)$ is one-to-one. Note that if $M = PI$ for some integer $I$, a global vector can be divided into $P$ equally sized sub-vectors, each containing $I = M/P$ entries. (For the case where $M$ is not divisible by

$P$, we refer to Section 3.2.) A partitioning scheme that lets each sub-vector contain contiguous entries from the global vector can be as follows:

$$I = \frac{M}{P}, \quad p = \lfloor \frac{i}{I} \rfloor, \quad j = i - pI\,.$$

Note that we use integer division in the above formula to calculate the $p$ value.

Once the partitioning is done, addition of the two vectors can be carried out straightforwardly in parallel by

$$\mathbf{w}_p = a\mathbf{x}_p + b\mathbf{y}_p, \quad p = 0, 1, \ldots P-1\,.$$

*Remarks.* The partitioning introduced above is *non-overlapping* in that each entry of the global vectors $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{w}$ is assigned to *only one* processor and $M = \sum_p M_p$. Furthermore, the local vector addition operation can be carried out by each processor, totally independent of other processors.

*Inner Product of Two Vectors.* Unlike the above vector addition operation, most parallel computations require inter-processor communication when being carried out on a multicomputer. An example is the parallel implementation of the inner-product between two vectors. More precisely, we have $\mathbf{x} \in \mathbb{R}^M$ and $\mathbf{y} \in \mathbb{R}^M$, and want to compute

$$c = \mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{M} x_i y_i\,.$$

Again, a partitioning of the global vectors $\mathbf{x}$ and $\mathbf{y}$ needs to precede the parallel inner-product operation. When each processor has the two sub-vectors ready, we continue with a local computation of the form:

$$c_p = \mathbf{x}_p \cdot \mathbf{y}_p = \sum_{j=1}^{M_p} x_{p,j} y_{p,j}\,.$$

Then, we can obtain the correct global result by

$$c = \sum_{p=0}^{P-1} c_p, \tag{1}$$

which can be realized by, e.g., asking every processor to send its local result $c_p$ to all the other processors. In this case, the parallel inner-product operation requires an all-to-all inter-processor communication, and we note that the correct global result $c$ will be available on every processor.

*Remark.* An alternative way of realizing (1) will be to designate a so-called *master* processor, which collects all the local results $c_p$ for computing $c$ and then broadcasts the correct global result to all the processors. Consequently, the above all-to-all communication is replaced by first an all-to-one communication and then a one-to-all communication.

## 1.5 Performance Modeling

We now study how the execution time of a parallel program depends on the number of processors $P$. Let us denote by

$$T = T(P)$$

the execution time for a parallel program when being run on $P$ processors. Performance analysis is to study the properties of $T(P)$, which mainly consists of two parts: arithmetic time and message exchange time. We remark that the above simple composition of $T(P)$ is a result of using the multicomputer model from Section 1.3. In the present section, we assume that the global problem size is fixed and thus only study how $T$ depends on $P$. Assuming also that computation and communication can not be performed simultaneously, we can come up with the following idealized model:

$$T(P) = \max_{0 \le p < P} I_p \tau_A + \max_{0 \le p < P} T_C^p, \tag{2}$$

where $I_p$ and $\tau_A$ are the number of arithmetic operations on processor number $p$ and the time needed by one single arithmetic operation, respectively. Moreover, $T_C^p$ is processor number $p$'s total communication time, which is spent on the involved message exchanges. More specifically, we have

$$T_C^p = \tau_C(L_1^p) + \tau_C(L_2^p) + \ldots, \tag{3}$$

where $L_1^p$ is the length of message number 1, $L_2^p$ is the length of message number 2, and so on. Moreover, the terms $\tau_C(L)$ in (3) are used to model the exchange cost of a message of length $L$ between two processors:

$$\tau_C(L) = \tau_L + \xi L. \tag{4}$$

Here, $\tau_L$ is the so-called *latency*, which represents the startup time for communication, and $1/\xi$ is often referred to as the *bandwidth*, which indicates the rate at which messages can be exchanged between two processors.

Recall that one major motivation of parallel computing is to reduce the execution time. The quality of a parallel program can thus be indicated by two related quantities: *speed-up* $S(P)$ and *efficiency* $\eta(P)$. More precisely,

$$S(P) = \frac{T(1)}{T(P)}, \quad \eta(P) = \frac{S(P)}{P},$$

where $T(1) = \hat{I}\tau_A$ is the execution time of a purely sequential program. We note that $T(1)$ is free of communication cost and the following observation is valid:

$$\hat{I} \le \sum_{p=0}^{P-1} I_p \le P \left( \max_{0 \le p < P} I_p \right) \quad \Rightarrow \quad T(1) \le P\,T(P)\,.$$

This observation also implies that

$$S(P) \leq P \tag{5}$$

and

$$\eta(P) \leq 1\,. \tag{6}$$

In practice, one normally has to be satisfied with $S(P) < P$ due to the communication cost. However, in certain special situations, *superlinear* speed-up, i.e., $S(P) > P$, can be obtained due to the cache effect. In other words, $\tau_A$ in this case effectively becomes smaller when $P$ is so large that local data fit into the processor cache, and, at the same time, the resulting efficiency gain exceeds the overhead communication cost.

## 1.6 Performance Analysis of the Two Examples

*Addition of Two Vectors.* For the example of adding two vectors in parallel, which was described in Section 1.4, we have

$$T(1) = 3M\tau_A \quad \text{and} \quad T(P) = \max_{0\leq p<P}(3M_p\tau_A) = 3I\tau_A, \quad I = \max_p M_p\,.$$

Therefore

$$S(P) = \frac{T(1)}{T(P)} = \frac{3M\tau_A}{3I\tau_A} = \frac{M}{I}\,.$$

Consequently, when the computational load is perfectly balanced among the processors, i.e., $I = M/P$, we obtain perfect speed-up

$$S(P) = P$$

and efficiency $\eta(P) = 1$.

*Inner Product of Two Vectors.* For the example of computing the inner-product in parallel, the situation is different because inter-processor communication is involved. We see that

$$T(1) = (2M - 1)\tau_A$$

and

$$T(P) = \max_{0\leq p<P}\left((2M_p - 1)\tau_A + (P-1)(\tau_C(1) + \tau_A)\right)\,.$$

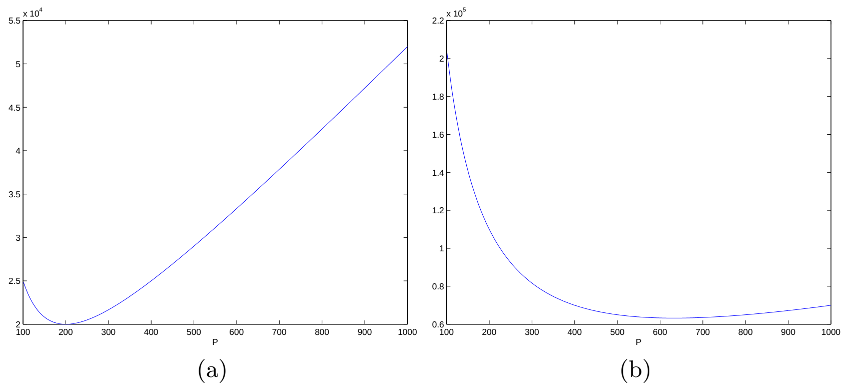

**Fig. 1.** Two plots of the function $T(P) = 2M/P + \gamma P$, where $\gamma = 50$. The value of $M$ is $10^6$ in (a) and $10^7$ in (b), respectively.

*Remark.* The part $(P-1)(\tau_C(1) + \tau_A)$ is due to the communication and computation cost associated with $c = \sum_p c_p$. That is, every processor sends its local result $c_p$ to all the other processors. The global result $c$ will be calculated on every processor. However, we note that real applications will use so-called *collective communication* routines for computing $c = \sum_p c_p$, where the total cost is of order $\log P(\tau_C(1) + \tau_A)$; see [4, Ch. 8].

Let us introduce the factor

$$\gamma = \frac{\tau_C(1)}{\tau_A}\,.$$

Under the assumption that $P \ll M$ and the load balance is perfect, i.e., $\max_p M_p = M/P$, we arrive at

$$S(P) = \frac{T(1)}{T(P)} \approx \frac{2M\tau_A}{2\frac{M}{P}\tau_A + P\tau_C(1)} = \frac{2M\tau_A}{\left(2\frac{M}{P} + \gamma P\right)\tau_A} = \frac{P}{1 + \frac{\gamma P^2}{2M}}\,.$$

Consequently, to achieve perfect speed-up, i.e., $S(P) \approx P$, it is necessary to have

$$\frac{\gamma P^2}{2M} \ll 1,$$

which is either due to an extremely fast communication speed (very small $\gamma$), or due to very large long vectors (very large $M$), or both reasons. The effect of $M$ is illustrated by the two plots in Figure 1 It is also obvious that a small value of $\gamma$ is always favorable for achieving good speed-up.

## 2 A Different Performance Model

The performance model (2) given in the preceding section is of rather limited usage in practice, due to the following reasons:

1. It is extremely difficult to come up with an accurate count $I_p$ for all the involved arithmetic operations in complicated parallel computations.
2. It is not realistic to assume that all the different arithmetic operations take constant time $\tau_A$. For instance, one division of two floating point numbers may be 8 times as costly as one multiplication. Besides, for more complicated operations, such as the calculation of a sine function, it is difficult to determine the exact cost.
3. The cost model (4) for communication is only valid for one-to-one inter-processor communication and therefore not valid for the collective inter-processor communication that is frequently encountered.

In the following, we propose a different performance model for $T(M, P)$, where $M$ represents the global size of the computation. This performance model does not make explicit use of $I_p$, $\tau_A$, $\tau_L$ and $\xi$.

### 2.1 The 2D Case

Suppose we are interested in parallel computations where the amount of local computations per processor is linearly proportional to the number of subgrid points. In addition, the involved inter-processor communication can be of two types:

1. Exchanging long vectors that contain boundary data between immediate neighboring processors;
2. Every processor collectively sends and receives short messages from all the other processors, as needed during parallel calculation of inner-product.

Let us consider a 2D computational grid with $M$ grid points and a multicomputer with $P$ processors. For simplicity, we consider a 2D lattice having $\sqrt{M}$ points in each direction, i.e., the total number of grid points is

$$M = \sqrt{M} \times \sqrt{M}\,.$$

Let us also assume that we can organize the multicomputer in a two-dimensional array of

$$P = \sqrt{P} \times \sqrt{P}$$

processors.

On each processor, there are

$$\frac{\sqrt{M}}{\sqrt{P}} \times \frac{\sqrt{M}}{\sqrt{P}} = \frac{M}{P}$$

computational points. The boundary of each subdomain consequently consists of $\mathcal{O}(\sqrt{M}/\sqrt{P})$ points. Here, the symbol '$\mathcal{O}$' indicates that the number of points on the boundary is proportional to $\sqrt{M}/\sqrt{P}$. In other words, we do not distinguish between schemes that exchange values on one or several layers of boundary points.

For this 2D case, the overhead due to the first type of communication is of order $\mathcal{O}(\sqrt{M}/\sqrt{P})$. The overhead due to the second type of communication will be of order $\mathcal{O}(\log P)$, assuming use of an effective implementation of such collective communication (see [4, ch. 8]). Since the computation time is of order $\mathcal{O}(M/P)$, we argue that the overall execution time can be modeled by

$$T(M,P) = \alpha\frac{M}{P} + \beta\frac{\sqrt{M}}{\sqrt{P}} + \varrho\log P, \tag{7}$$

where $\alpha$, $\beta$ and $\varrho$ are three constant parameters to be determined by applying e.g. the method of least squares (see [7, ch. 8]) to a series of measurements of $T$ for different values of $M$ and $P$.

*Remarks*

1. We have assumed that the cost of the first type of communication depends only on the amount of data to be exchanged between two processors. In other words, we have ignored latency in (7). Most likely, this is probably appropriate for properly scaled problems. However, if we keep $M$ fixed and increase $P$ to be close to $M$, the model will produce a poor approximation. In such a case the execution time $T$ will eventually be dominated by the overhead of latency.
2. Note also that model (7) does not consider the cost of waiting for a communication channel to be available. That is, we assume that any pair of processors can communicate with each other undisturbed. This assumption is valid for e.g. clusters of PC nodes that are connected through a high-speed switch.
3. We have also neglected the overhead of synchronization that will arise during the second type of communication, in case of poor computational load balance among the processors.
4. In practical computations, the convergence of iterative schemes may slow down as $M$ increases. We do not take this effect into account. A more realistic model can be formulated as:

$$T(M,P) = \alpha\frac{M^{\delta}}{P} + \beta\frac{\sqrt{M}}{\sqrt{P}} + \varrho\log P,$$

   where $\delta \geq 1$.

*A Numerical Experiment.* We consider a 2D parallel computation that involves 400 Conjugate Gradient iterations. Based on a set of actual timing measurements, see Table 1, we have used the method of least squares to compute the parameters $\alpha$, $\beta$ and $\varrho$ to produce the following model:

$$T(M,P) = 1.086\cdot 10^{-3}\frac{M}{P} + 6.788\cdot 10^{-4}\frac{\sqrt{M}}{\sqrt{P}} + 2.523\cdot 10^{-1}\log P\,. \tag{8}$$

**Table 1.** Actual timing measurements of a 2D parallel computation that involves 400 Conjugate Gradient iterations.

| $M$ | $P=2$ | $P=4$ | $P=8$ | $P=16$ |
|---|---|---|---|---|
| 40,401 | 21.61 | 10.81 | 5.96 | 3.57 |
| 90,601 | 49.43 | 24.85 | 12.73 | 7.14 |
| 160,801 | 87.67 | 45.07 | 23.06 | 12.21 |
| 251,001 | 136.36 | 69.32 | 35.67 | 18.10 |

Thereafter, we use an additional set of timing measurements of the *same* computation to check the accuracy of model (8). Note that this additional timing set was not used for estimating $\alpha$, $\beta$ and $\varrho$ in (8). In Table 2, the actual timing measurements are compared with the estimates, which are the italic-fonted numbers in parentheses. We observe that the estimates are fairly accurate.

**Table 2.** Comparison between the actual timing measurements and the estimates (in parentheses) produced by model (8).

| $M$ | $P=2$ | $P=4$ | $P=8$ | $P=16$ |
|---|---|---|---|---|
| $641,601$ | 348.58 (*348.93*) | 175.55 (*174.92*) | 89.16 (*88.02*) | 45.39 (*44.68*) |

### 2.2 The 3D Case

We can argue similarly for the 3D case, as for the above 2D case. Suppose we have a computational lattice grid with $M = M^{1/3} \times M^{1/3} \times M^{1/3}$ points and the multicomputer has a three-dimensional array of $P = P^{1/3} \times P^{1/3} \times P^{1/3}$ processors.

On each processor, there are

$$\frac{M^{1/3}}{P^{1/3}} \times \frac{M^{1/3}}{P^{1/3}} \times \frac{M^{1/3}}{P^{1/3}} = \frac{M}{P}$$

computational points. The number of points on the boundary surface of one subdomain is of order

$$\frac{M^{1/3}}{P^{1/3}} \times \frac{M^{1/3}}{P^{1/3}} = \frac{M^{2/3}}{P^{2/3}}.$$

So the communication cost is of order $\mathcal{O}(\frac{M^{2/3}}{P^{2/3}})$, and the computation time is of order $\mathcal{O}(\frac{M}{P})$. Therefore, the overall execution time can be modeled by

$$T(M,P) = \alpha\frac{M}{P} + \beta\frac{M^{2/3}}{P^{2/3}} + \varrho \log P\,. \tag{9}$$

### 2.3 A General Model

In general, we can argue that $T(M,P)$ can be modeled by

$$T(M,P) = \alpha \frac{M}{P} + \beta \left(\frac{M}{P}\right)^{\frac{d-1}{d}} + \varrho \log P, \tag{10}$$

where $d$ is the number of spatial dimensions. The remarks given in Section 2.1 also apply to the models (9) and (10).

## 3 The First MPI Encounter

Recall that we have briefly explained the basic ideas of the message passing programming model in Section 1.2. The acronym MPI stands for *message passing interface* [3], which is a library specification of message passing routines. MPI has a rich collection of such routines and is the de-facto standard for writing portable message passing programs. This section aims to introduce the reader to basic MPI routines by showing a couple of simple MPI programs in the C language. A complete comprehension of the MPI routines is not strictly necessary, since those low-level routines will be replaced by more flexible high-level Diffpack counterparts in parallel Diffpack programs.

### 3.1 "Hello World" in Parallel

The most important thing to keep in mind when writing an MPI program is that the same program is to be executed, simultaneously, as multiple processes. These processes are organized, by default, in the context of a so-called MPI communicator with name `MPI_COMM_WORLD`. Inside `MPI_COMM_WORLD`, each process is identified with a unique rank between 0 and $P-1$, where $P$ is the total number of processes. The processes can refer to each other by their ranks. The following short program demonstrates the MPI routines for determining the number of processes and the process rank, while also showing the most fundamental pair of MPI routines, namely `MPI_Init` and `MPI_Finalize`. Calls to these two routines must enclose any other other MPI routines to be used in an MPI program.

```c
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I'm No.%d of %d procs.\n",my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

The compilation of the above MPI program, assuming the file name `hello.c`, can be done by, e.g.,

```
mpicc hello.c
```

The resulting executable, say `a.out`, can be run by e.g. the following command:

```
mpirun -np 48 a.out
```

We note that the above `mpicc` and `mpirun` commands are standard in many MPI implementations. The execution result in each process printing out a message. More precisely, if the MPI program is executed as e.g. 48 processes, the process with rank 5 will print out the following message:

```
Hello world, I'm No.5 of 48 procs.
```

### 3.2 Computing the Norm of a Vector

Our next example concerns the computation of the Euclidean norm of a long vector $v = (v_1, \ldots, v_M)$. The vector is to be distributed among a number of processes, i.e., we cut up $v$ into $P$ pieces. Process number $p$ owns the local vector segment: $v_1^p, \ldots, v_{M_p}^p$, where $M_p$ is the number of vector components on process $p$. For instance, if vector $v$ is of length $4 \cdot 10^6$ and we have four processes available, we would typically set $M_p = 10^6$ and store $(v_1, \ldots, v_{10^6})$ in the process with rank 0, $(v_{10^6+1}, \ldots, v_{2\cdot 10^6})$ in the process with rank 1, and so on.

In the general case, where we assume that $M$ may not be divisible by $P$, we can use a simple (but not optimal) workload partitioning strategy by assigning the remainder from $M/P$ to the last process. More precisely, we first set $r = (M \bmod P)$, and then calculate the length of the local subvectors by $M_p = (M - r)/P$ for $p < P - 1$, and $M_{P-1} = (M - r)/P + r$. Using integer division in C we can simply write

```
M_p = M/P;  r = M % P;
```

As for the implementation, we assume that the process with rank 0 is in charge of reading $M$ from some user input, e.g., the command line. Then the value $M$ is broadcast to all the other processes. The next task of the program is to create the local segment of the vector and fill the vector components with appropriate numbers. Assume for simplicity that the components of vector $v$ have e.g. values $v_i = 3i + 2.2$, $1 \leq i \leq M$. Then, on process $p$, local component $v_j^p$ equals $3(j + pM_p) + 2.2$, $1 \leq j \leq M_p$.

After having initialized the local vector segment, we compute the square of the norm of the current segment: $N_p = \sum_{j=1}^{M_p} (v_j^p)^2$. The next step is to add the $N_p$ values from all the processes and write out the global norm $\sqrt{\sum_p N_p}$ of the $v$ vector in one of the processes, e.g. process 0. We mention that a minor modification of the following implementation can be used to compute the inner-product between two vectors, described in Section 1.4.

```
#include <stdio.h>
#include <mpi.h>
#include <malloc.h>
#include <math.h>

int main(int argc, char** argv)
{
  int P, my_rank, i, j;
  int M = 1000;  /* length of global vector, default 1000 */
  int M_p;    /* length of vector segment for this process */
  int r;      /* the remainder of the length (for last proc) */
  double local_norm2, global_norm2;
  double* vector;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &P);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  /* only the master process should read input */
  if (my_rank==0 && argc>1)
    M = atoi(argv[1]);

  /* broadcast the length to the other processes */
  MPI_Bcast (&M, 1, MPI_INT, 0, MPI_COMM_WORLD);

  M_p = M/P;  r = M % P;

  /* the remaining components will be placed on the last process, so
     the remainder variable is set to 0 on all other processes. */
  if (my_rank < P-1)   r = 0;

  /* create the vector segment on this process */
  vector = (double*)malloc((M_p+r)*sizeof(double));

  /* initialize vector  (simple formula: vector(i) = 3*i + 2.2) */
  for (j = 0; j < M_p+r; j++) {
    i = j + 1 + my_rank * M_p; /* find global index */
    vector[j] = 3*i + 2.2;
  }

  /* Compute the _square_ of the norm of the local vector segment
     (cannot simply add local norms; must add square of norms...) */
  local_norm2 = 0.;
  for (j = 0; j < M_p+r; j++)
    local_norm2 += vector[j]*vector[j];

  /* let the master process sum up the local results */
  MPI_Reduce (&local_norm2,&global_norm2,1,MPI_DOUBLE,
              MPI_SUM,0,MPI_COMM_WORLD);

  /* let only the master process write the result to the screen */
  if (my_rank==0)
    printf("\nThe norm of v(i) = 3*i + 2.2, i = (1,...,%d) is %g\n",
           M, sqrt(global_norm2));

  free (vector);

  MPI_Finalize ();
  return 0;  /* successful execution */
}
```

In the above MPI program, `MPI_Bcast` is used to "broadcast" the value of `M`, which is an integer available on process 0, to all the other processes. Moreover, the `MPI_Reduce` command collects the local results `local_norm2` from all the processes and calculates the global result `global_norm2`, which will be available on process 0. We remark that both the MPI commands need to be invoked on *all* the processes, even though the input or output value is only available on process 0.

## 4 Basic Parallel Programming with Diffpack

Developing a parallel program is always more complicated than developing a corresponding sequential program. Practice shows that transforming a parallel algorithm into running code is often a time-consuming task, mainly because the parallel programming tools are primitive. The standard message passing protocol in wide use today is MPI, but MPI programming tends to be notoriously error-prone due to the many low-level message passing details, which need to be taken care of by the programmer.

To increase the human efficiency in developing parallel computer codes, we should develop a software environment where the programmer can concentrate on the principal steps of parallel algorithms, rather than on MPI-specific details. A desired situation will be that a programmer can start with developing a sequential solver and then in just a few steps transform this solver to a parallel version. Realization of such a software environment is indeed possible and requires a layered design of software abstractions, where all explicit MPI calls are hidden in the most primitive layer, and where the interface to message passing tools is simple and adapted to the programming standard of sequential solvers. We have developed this type of software environment in Diffpack, and the usage of the environment will be explained in the present section.

### 4.1 Diffpack's Basic Interface to MPI

MPI routines normally have a long list of input/output arguments. The task of calling them can be considerably simplified by using the strengths of C++. That is, overloaded functions, classes, and dynamic binding can be used to build a generic and simple interface to a subset of MPI routines that is needed for solving partial differential equations in parallel. In Diffpack, this interface consists of a class hierarchy with base class `DistrProcManager` (distributed process manager), which gives a generic representation of all the commonly needed message passing calls in the Diffpack framework. Class `DistrProcManagerMPI` is one concrete subclass where all the virtual member functions of `DistrProcManager` are implemented using MPI calls. The reason for keeping such a class hierarchy is to make a switch of message passing protocols painless to the user. That is, we can e.g. make another subclass `DistrProcManagerPVM` in the parallel Diffpack library, so that the user can use the other message passing standard PVM [10] without having to change the source code of his parallel Diffpack application. Therefore the reader only needs to get familiar with the functionality of class `DistrProcManager`.

*The DistrProcManager Class.* A Diffpack programmer normally calls a member function belonging to the `DistrProcManager` class, instead of calling a primitive message passing routine directly. Diffpack has a global handle[1] variable with name `proc_manager`, which is available everywhere in a program. At the beginning of every parallel Diffpack simulation, the `proc_manager` variable is bound to an object of some subclass of `DistrProcManager` (see above). As usual in object-oriented programming, the programmer can always concentrate on the base class interface to a class hierarchy, so we will only discuss `DistrProcManager` in the following.

Overloading is an attractive feature of C++ that distinguishes between calls to functions having the same name but with different argument lists. This has been used in the `DistrProcManager` class to allow the user to call, for instance, the `broadcast` function with a variable to be broadcast as parameter. The C++ run-time system will automatically choose the appropriate `broadcast` function depending on the argument type. On the other hand, using MPI's `MPI_Bcast` function directly would have required the datatype, along with several other arguments, to be specified explicitly.

The ability to use default arguments in C++ is another feature that is used frequently in `DistrProcManager`. Most calls to MPI routines require a communicator object as argument. However, the average user is unlikely to use any other communicator than `MPI_COMM_WORLD`. By using this communicator as a default argument to all the functions in `DistrProcManager`, the users are allowed to omit this argument unless they wish to use a different communicator. By taking advantage of overloaded functions and arguments with default values, the interface to common MPI calls is significantly simplified by the `DistrProcManager` class. The main results are twofold: (i) novice programmers of parallel computers get started more easily than with pure MPI, and (ii) the resulting code is cleaner and easier to read.

In the same manner as overloaded versions of the `broadcast` function are built into `DistrProcManager`, other useful functions are also present in

[1] A Diffpack handle is a smart pointer with internal functionality for reference counting.

this class. These include, e.g., the functions for sending and receiving messages, both blocking (`MPI_Send` and `MPI_Recv`) and non-blocking (`MPI_Isend` and `MPI_Irecv`). Overloaded versions of the functions cover communication of real, integer, string, and Diffpack vector variables. For example, native Diffpack vectors like `Vec(real)` can be sent or received directly, without the need for explicitly extracting and communicating the underlying C-style data structures of the Diffpack vector classes.

*Some Important DistrProcManager Functions.* To give the reader an idea about programming with the `DistrProcManager` interface, we list some of its important functions (often in their simplest form):

```
int getNoProcs();       // return number of processes
int getMyId();          // return the ID (rank) of current process,
                        // numbered from 0 to getNoProcs()-1
bool master();          // true: this is the master process (ID=0)
bool slave();           // true: this is a slave process (ID>0)

void broadcast (real& r);   // send r to all the processes
void broadcast (int&  i);
void broadcast (String& s);
void broadcast (VecSimple(int)&  vec);
void broadcast (VecSimple(real)& vec);

// blocking communications:
void send (real& r, int to = 0, int tag = 0);
void recv (real& r, int from,   int tag);
void send (int&  i, int to = 0, int tag = 0);
void recv (int&  i, int from,   int tag);
void send (VecSimple(real)& vec, int to = 0, int tag = 0);
void recv (VecSimple(real)& vec, int from,   int tag);

// non-blocking communications:
void startSend (real& r, int to = 0, int tag = 0);
void startRecv (real& r, int from,   int tag);
void startSend (int&  i, int to = 0, int tag = 0);
void startRecv (int&  i, int from,   int tag);
void startSend (VecSimple(real)& vec, int to = 0, int tag = 0);
void startRecv (VecSimple(real)& vec, int from,   int tag);
void finishSend ();
void finishRecv ();
```

We remark that using the so-called blocking versions of the send and receive functions imply that communication and computation can not take place simultaneously. In contrast, the non-blocking version allows overlap between communication and computation. It is achieved by first invoking a `startSend` routine, then issuing some computation routines, and finally calling `finishSend`.

### 4.2 Example: The Norm of a Vector

For the purpose of illustrating how one can use class `DistrProcManager` as a high-level MPI interface, we re-implement here our simple example from Section 3.2 concerning parallel computation of the norm of a vector.

```
#include <MenuSystem.h>
#include <Vec_real.h>
#include <DistrProcManager.h>
#include <initDpParallelBase.h>

int main(int argc, const char** argv)
{
  initDpParallelBase (argc, argv);
  initDiffpack (argc, argv);

  int M;      // length of global vector
  int M_p;    // length of vector segment for this process
  int r;      // the remainder of the length (for last proc)

  // only the master process (with rank 0) should read input:
  if (proc_manager->master())
    initFromCommandLineArg ("-L", M, 1000);

  // broadcast the length to the other processes
  proc_manager->broadcast(M);

  const int P = proc_manager->getNoProcs();
  M_p = M/P;  r = M % P;

  // the remaining components will be placed on the last process, so
  // the remainder variable is set to 0 on all other processes.
  if (proc_manager->getMyId() < P-1)   r = 0;

  // create the vector segment on this process:
  Vec(real) vector(M_p + r);

  // initialize vector  (simple formula: vector(i) = 3*i + 2.2)
  int i,j;
  for (j = 1; j <= vector.size(); j++) {
    i = j + proc_manager->getMyId() * M_p; // find global index
    vector(j) = 3*i + 2.2;
  }

  // Compute the _square_ of the norm of the local vector segment
  // (cannot simply add local norms; must add square of norms...)
  real local_norm2 = sqr(vector.norm()); // or vector.inner(vector)
  real global_norm2;
  // let the master process (with rank 0) sum up the local results
  proc_manager->reduce(local_norm2, global_norm2, PDP_SUM);

  // let only the master process write the result to the screen:
  if (proc_manager->master()) {
    s_o << "\nThe norm of v(i) = 3*i + 2.2, i = (1,...,"
        << M << ")\n" << " is " << sqrt(global_norm2) << "\n\n";
  }
  closeDpParallelBase();
  return 0;  // successful execution
}
```

We note that there are no direct calls to MPI routines in the above parallel program. All the communications are invoked through the global Diffpack variable `proc_manager`. The first initialization statement of the program, which is `initDpParallelBase`, takes care of `MPI_Init`, `MPI_Comm_rank`, and `MPI_Comm_size`, after having internally instantiated the global variable `proc_manager`. In this example, `proc_manager` is pointing to an object of class `DistrProcManagerMPI`. At the end of the program, `MPI_Finalize` is invoked internally within the closing statement `closeDpParallelBase`. If more advanced functionality for parallelizing, e.g., implicit finite element codes (see Section 6) is desired, the `initDpParallelBase` and `closeDpParallelBase` pair should be replaced by `initDpParallelLA` and `closeDpParallelLA`. Note also that the call of `reduce` in the above program invokes a global collective reduction operation. A global summation is invoked in the current example, due to the input argument `PDP_SUM`, which is Diffpack's equivalent of MPI's `MPI_SUM`. The member function `reduce` must be called on every process, and the result is only available on the master process (with rank 0).

## 5 Parallelizing Explicit FD Schemes

The topic of this section is how to parallelize sequential Diffpack simulators that employ explicit finite difference schemes. We will present a set of programming rules and the associated classes specially designed for this type of parallelization.

### 5.1 Partitioning Lattice Grids

Assume that we want to solve a system of partial differential equations over some spatial domain $\Omega$, which is covered by a rectangular lattice grid. For such a case, the easiest way of partitioning the global computation is to decompose the global lattice grid with cutting lines or planes that are perpendicular to the coordinate axes. This type of domain partitioning ensures that the resulting subdomains $\Omega_s$, $s = 1, \ldots, D$, can also be covered by rectangular lattice subgrids. We note that this rectangular domain partitioning scheme can use fewer space dimensions than that of the global lattice grid. For example, we can partition a 2D global lattice grid with only cutting lines perpendicular to the $x$-axis. However, the resulting subdomain lattice grids will always have the original number of space dimensions.

It is obvious that the above decomposition of the global lattice grid gives rise to a partitioning of the global computation, so that one lattice subgrid is assigned to one process. An immediate observation is that the same explicit finite difference scheme can be applied to each lattice subgrid. However, the

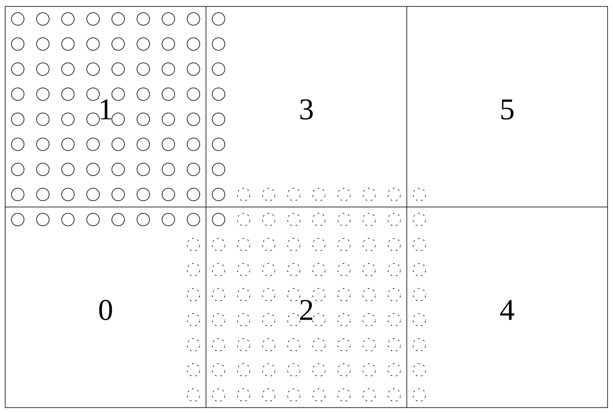

**Fig. 2.** An example of 2D overlapping subdomains.

grid point layers that lie beside the cutting lines or planes need special attention. This is because to update the point values in those layers, we need point values belonging to the neighboring subdomain(s). This can be done by exchanging arrays of point values between neighboring subdomains. A feasible data structure design is therefore to enlarge each lattice subgrid by one or several grid point layers beyond the cutting line or plane. For example, one grid point layer suffices for the standard second-order finite difference approximation $(u_{i-1} - 2u_i + u_{i+1})/\Delta x^2$ to second-order differentiation. The subdomains therefore become overlapping, see Figure 2 for an example of overlapping 2D subdomains.

After the enlargement of the data structure on each subdomain, we distinguish between two types of grid points, in addition to the genuine physical boundary points. These two types of grid points are: *computational points* whose values are updated by local computations, and *ghost boundary points* whose values come from the neighboring subdomains. Figure 3 shows an example of partitioning a global 1D lattice grid, where the vertical lines are the cutting lines. Each subdomain is enlarged with one ghost boundary point at each end. We have plotted grid points of the middle subdomain as circles above the axis, where empty circles represent the computational points and shaded circles represent the ghost boundary points. Note that the ghost boundary points of one subdomain are computational points of neighboring subdomains. Note also that the local computation will only traverse the computational points but make use of the values on the ghost boundary points. In effect, the data structure of each subdomain closely mimics that of the global domain, because we can view the ghost boundary points as having essential boundary conditions that are determined by the neighboring subdomains.

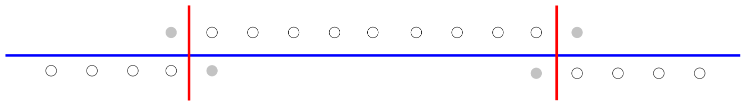

**Fig. 3.** An example of 1D lattice subgrids with ghost boundary points (shaded circles).

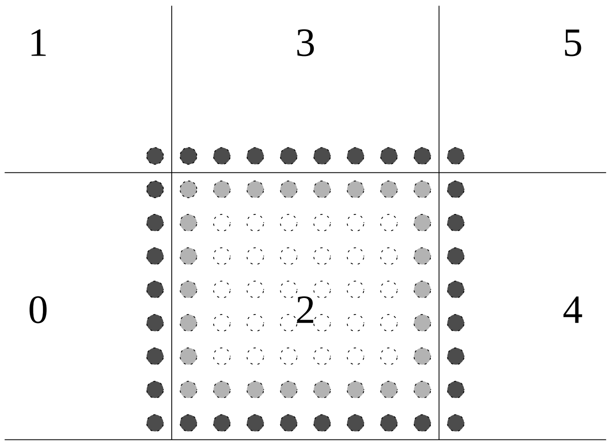

**Fig. 4.** The dark shaded grid points have offset +1, the light shaded points have offset 0, the clear grid points have offset -1 and less.

Another classification of the grid points can be convenient for dealing with communication and computations. Grid points at the boundary of a subdomain are said to have offset +1. The immediate neighboring points within the boundary are said to have offset 0, while the remaining interior points have offset -1 or less, as illustrated in Figure 4. The points with offset +1 are either part of the genuine physical boundary or they correspond to points with offset 0 on a neighboring subdomain. According to the above definitions of ghost boundary points and computational points, grid points with offset +1 are either ghost boundary points or physical boundary points, whereas grid points with zero or negative offset are computational points.

### 5.2 Problem Formulation

The most common application of explicit finite difference schemes is perhaps the solution of temporal evolution equations. This class of computation can be written as

$$u_i^+(x) = \mathcal{L}_i(u_1^-(x), \ldots, u_n^-(x)), \quad i = 1, \ldots, n, \tag{11}$$

where $u_i(x)$ is a function of the spatial coordinates $x \in \Omega$, $\mathcal{L}_i$ is some spatial differential operator (possibly non-linear), and $n$ is the number of unknown functions in the underlying partial differential equation system. The superscript $^+$ denotes an unknown function at the current time level, while the superscript $^-$ refers to a function that is known numerically from previous time levels. Restriction of (11) to the subdomains is straightforward:

$$u_i^{s,+}(x) = \mathcal{L}_i(u_1^{s,-}(x), \ldots, u_{n_s}^{s,-}(x); g^-), \quad i = 1, \ldots, n_s, \ s = 1, \ldots, D\,. \tag{12}$$

The new solution over $\Omega_s$ is $u_i^{s,+}(x)$, while $u_i^{s,-}(x)$ are previously computed solutions over $\Omega_s$. Depending on the discretization of $\mathcal{L}_i$, previously computed functions $u_i^{q,-}(x)$ from *neighboring* subdomains $\Omega_q$, with $q$ in some suitable index set, are also needed when solving the subproblem on $\Omega_s$. This information is represented as the $g^-$ quantity in (12), which refers to the values on the ghost boundary points. In other words, the coupling between the subproblems is through previously computed functions, on the ghost boundary points, when we work with explicit temporal schemes. At a new time level, we can then solve the subproblems over $\Omega_s$, $s = 1, \ldots, D$ in parallel. Thereafter, we need to exchange the $g^-$ information between the subdomains before we proceed to the next time level. The mathematical and numerical details will become clearer when we study a specific initial-boundary value problem below.

One important observation now is that sequential software can be re-used for updating the values on the computational points on each subdomain. For exchanging data between neighboring subdomains, additional functionality needs to be developed. When using C++, Diffpack and object-oriented concepts, we can build the software for a parallel solver in a clean way, such that the debugging of e.g. communication on parallel computers is minimized. First, one develops a standard sequential solver class for the problem over $\Omega$ and tests this solver. Thereafter, one derives a subclass with additional general data structures for boundary information exchange. This subclass contains little code and relies on the numerics of the already well-tested base class for the global problem, as well as on a general high-level communication tool for exchange of boundary information.

The subclass solver will be used to solve problems on every subdomain $\Omega_s$, $s = 1, \ldots, D$. In addition, we need a manager class per subdomain that holds some information about the relations between the subdomains $\Omega_s$. The principle is that the managers have a global overview of the subproblems, while the subproblem solvers have no explicit information about other subproblem solvers. At any time, one can pull out the original sequential solver to ensure that its numerics work as intended. With this set-up, it is fairly simple to take an existing sequential solver, equip it with two extra small classes and turn it into a simulation code that can take advantage of parallel computers. This strategy may have a substantial impact on the development time of parallel simulation software.

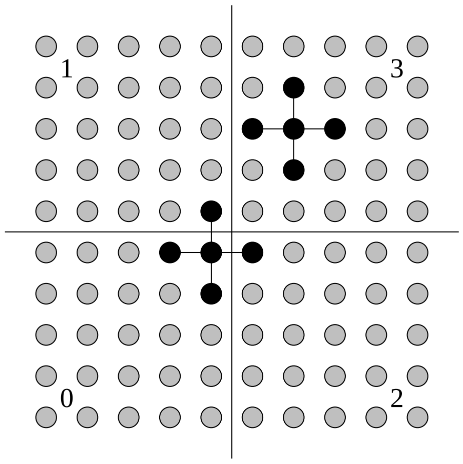

**Fig. 5.** Updating the value in subdomain 0 (points with offset 0) requires values from subdomains 1 and 2. The indicted value in subdomain 3 (point with offset -1) depends only on local information and can be updated without communicating any values from the neighbors.

### 5.3 Hiding the Communication Cost

When applying the finite difference scheme on a subdomain, we need boundary values from the neighboring subdomain at the points with offset +1. These values are used in the difference equations for the points with offset 0. See Figure 5 for the example of a five-point computational stencil. The main idea in the parallel algorithm is to communicate these boundary values while applying the finite difference scheme to all points with offset -1 or less. When the boundary values are received, one can apply the scheme to the points with offset 0 and the solution over a subdomain is complete.

On many parallel computers, computations and communication can be carried out simultaneously. In MPI, for instance, this is accomplished by non-blocking send and receive calls. We can therefore devise the following algorithm for a subdomain solver:

1. Send requested points to all neighboring subdomains (these points will be used as boundary conditions in the neighboring subdomain solvers).
2. Update the solution at all purely inner points (those with offset -1 and less).
3. Receive boundary values at points with offset +1 from all neighboring subdomains.
4. Update the solution at the points with offset 0.

For small grids we may have to wait for the messages to arrive, because the time spent on communication may exceed the time spent on computing inner points. The length of the message depends on the circumference of the grid, while the number of inner values to be computed depend on the area of the grid (cf. Section 2.1). Therefore, as the size of the grid grows, the time spent on computing will grow faster than the time spent on communication, and at some point the communication time will be insignificant. The speedup will then be (close to) optimal; having $P$ processors reduces the CPU time by approximately a factor of $P$.

## 5.4 Example: The Wave Equation in 1D

Consider the one-dimensional initial-boundary value problem for the wave equation:

$$\frac{\partial^2 u}{\partial t^2} = \gamma^2 \frac{\partial^2 u}{\partial x^2}, \quad x \in (0,1),\ t > 0, \tag{13}$$

$$u(0,t) = U_L, \tag{14}$$

$$u(1,t) = U_R, \tag{15}$$

$$u(x,0) = f(x), \tag{16}$$

$$\frac{\partial}{\partial t} u(x,0) = 0\,. \tag{17}$$

Let $u_i^k$ denote the numerical approximation to $u(x,t)$ at the spatial grid point $x_i$ and the temporal grid point $t_k$, where $i$ and $k$ are integers, $1 \le i \le n$ and $k \ge -1$ ($k = 0$ refers to the initial condition and $k = -1$ refers to an artificial quantity that simplifies the coding of the scheme). Assume here that $x_i = (i-1)\Delta x$ and $t_k = k\Delta t$. With the definition of the Courant number as $C = \gamma \Delta t / \Delta x$, a standard finite difference scheme for this equation reads

$$\begin{aligned} u_i^0 &= f(x_i), \quad i = 1, \ldots, n, \\ u_i^{-1} &= u_i^0 + \frac{1}{2} C^2 (u_{i+1}^0 - 2u_i^0 + u_{i-1}^0), \quad i = 2, \ldots, n-1, \\ u_i^{k+1} &= 2u_i^k - u_i^{k-1} + C^2 (u_{i+1}^k - 2u_i^k + u_{i-1}^k), \quad i = 2, \ldots, n-1,\ k \ge 0, \\ u_1^{k+1} &= U_L, \quad k \ge 0, \\ u_n^{k+1} &= U_R, \quad k \ge 0\,. \end{aligned}$$

A domain partitioning for the numerical scheme is easily formulated by dividing the domain $\Omega = [0,1]$ into overlapping subdomains $\Omega_s$, $s = 1, \ldots, D$, and applying the scheme to the interior points in each subdomain. More precisely, we define the subdomain grids as follows.

$$\Omega_s = [x_L^s, x_R^s], \quad x_i^s = x_L^s + (i-1)\Delta x,\ i = 1, \ldots, n_s\,.$$

Here, $n_s$ is the number of grid points in $\Omega_s$. For the end points $x_L^s$ and $x_R^s$ of subdomain $\Omega_s$, we have

$$x_1^s = x_L^s, \quad x_{n_s}^s = x_R^s\,.$$

The physical boundaries are given by

$$x_L^1 = 0, \quad x_R^D = 1\,.$$

The scheme for $u_i^{k+1}$ involves a previously computed value on the grid point to the left ($u_{i-1}^k$) and the grid point to the right ($u_{i+1}^k$). We therefore need one grid cell overlap between the subdomains. This implies some relations between the start and end grid points of the subdomain, which are listed here (although they are not of any practical use in the implementation):

$$x_1^s = x_{n_{s-1}-1}^{s-1},\ x_2^s = x_{n_{s-1}}^{s-1}, \quad x_{n_s-1}^s = x_1^{s+1},\ x_{n_s}^s = x_2^{s+1},$$

for $s = 2, \ldots, D-1$. The relations assume that the subdomains $\Omega_1, \Omega_2, \Omega_3$ and so on are numbered from left to right.

*Remark.* In multi-dimensional problems, a precise mathematical notation for the overlap is probably not feasible. Instead, we only know that there is some kind of overlap, and we need the principal functionality of evaluating $u$ in a neighboring subdomain at a specified point. Let $\mathcal{N}_s$ be an index set containing the neighboring subdomains of $\Omega_s$ (in 1D $\mathcal{N}_s = \{s-1, s+1\}$). In general, we need an interpolation operator $I(x, t; \mathcal{N}_s)$ that interpolates the solution $u$ at the point $(x, t)$, where $x$ is known to lie in one of the subdomains $\Omega_q$, $q \in \mathcal{N}_s$. We note that if the grid points in the subdomain grids coincide, $I(x, t; \mathcal{N}_s)$ just makes an appropriate copy.

Defining $u_i^{s,k}$ as the numerical solution over $\Omega_s$, we can write the numerical scheme for subdomain $\Omega_s$ as follows.

$$\begin{aligned}
u_i^{s,0} &= f(x_i), \quad i = 1, \ldots, n_s,\\
u_i^{s,-1} &= u_i^{s,0} + \frac{1}{2}C^2(u_{i+1}^{s,0} - 2u_i^{s,0} + u_{i-1}^{s,0}), \quad i = 2, \ldots, n_s - 1,\\
u_i^{s,k+1} &= 2u_i^{s,k} - u_i^{s,k-1} + C^2(u_{i+1}^{s,k} - 2u_i^{s,k} + u_{i-1}^{s,k}),\ i = 2, \ldots, n_s - 1,\ k \geq 0,\\
\text{if } x_1^s = 0, \quad u_1^{s,k+1} &= U_L, \quad \text{else} \quad u_1^{s,k+1} = I(x_1^s, t_{k+1}; \mathcal{N}_s), \quad k \geq 0,\\
\text{if } x_{n_s}^s = 1, \quad u_{n_s}^{s,k+1} &= U_R, \quad \text{else} \quad u_{n_s}^{s,k+1} = I(x_{n_s}^s, t_{k+1}; \mathcal{N}_s), \quad k \geq 0.
\end{aligned}$$

Apart from the $s$ superscript and the need to interpolate values between neighboring subdomains, this scheme is identical to the scheme for the global problem. In other words, we can re-use most of the sequential code for the original global problem, provided that this code can deal with an arbitrary subdomain and that we add the interpolation functionality. This latter feature involves communication between subdomains, implying communication between processors.

In the following, we present a plain C program for the one-dimensional wave equation problem and equip it with direct MPI calls for communicating boundary values. We remark that the lower bound of an array index in C starts at 0, instead of 1 in Diffpack. In the program, we also use an integer variable `n_i`, which has value equal to $n_s - 2$. Knowledge about the forthcoming details of the implementation is not required elsewhere in the chapter, so readers who want to concentrate on Diffpack's parallel programming interface can safely move on to the next section.

```
#include <stdio.h>
#include <malloc.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank, i, n = 1001, n_i, lower_bound;
  double h, x, t, dt, gamma = 1.0, C = 0.9, tstop = 1.0;
  double *up, *u, *um, umax = 0.05, UL = 0., UR = 0.;
  MPI_Status status;

  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  if (my_rank==0 && nargs>1)    /* total number of points in [0,1] */
    n = atoi(args[1]);

  if (my_rank==0 && nargs>2)    /* read in Courant number */
    C = atof(args[2]);

  if (my_rank==0 && nargs>3)    /* length of simulation */
    tstop = atof(args[3]);

  MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast (&C, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast (&tstop, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  h = 1.0/(n-1);   /* distance between two points */

  /* find # of local computational points, assume mod(n-2,size)=0 */
  n_i = (n-2)/size;
  lower_bound = my_rank*n_i;

  up = (double*) malloc ((n_i+2)*sizeof(double));
  u  = (double*) malloc ((n_i+2)*sizeof(double));
  um = (double*) malloc ((n_i+2)*sizeof(double));

  /* set initial conditions */
  x = lower_bound*h;
  for (i=0; i<=n_i+1; i++) {
    u[i] = (x<0.7) ? umax/0.7*x : umax/0.3*(1.0-x);
    x += h;
  }
  if (my_rank==0) u[0] = UL;
  if (my_rank==size-1) u[n_i+1] = UR;
  for (i=1; i<=n_i; i++)
    um[i] = u[i]+0.5*C*C*(u[i+1]-2*u[i]+u[i-1]); /* artificial BC */

  dt = C/gamma*h;
  t = 0.;
  while (t < tstop) {    /* time stepping loop */
```

```
    t += dt;
    for (i=1; i<=n_i; i++)
      up[i] = 2*u[i]-um[i]+C*C*(u[i+1]-2*u[i]+u[i-1]);

    if (my_rank>0) {
      /* receive from left neighbor */
      MPI_Recv (&(up[0]),1,MPI_DOUBLE,my_rank-1,501,MPI_COMM_WORLD,
                &status);
      /* send left neighbor */
      MPI_Send (&(up[1]),1,MPI_DOUBLE,my_rank-1,502,MPI_COMM_WORLD);
    }
    else
      up[0] = UL;

    if (my_rank<size-1) {
      /* send to right neighbor */
      MPI_Send (&(up[n_i]),1,MPI_DOUBLE,my_rank+1,501,MPI_COMM_WORLD);
      /* receive from right neighbor */
      MPI_Recv (&(up[n_i+1]),1,MPI_DOUBLE,my_rank+1,502,
                MPI_COMM_WORLD,&status);
    }
    else
      up[n_i+1] = UR;

    /* prepare for next time step */
    um = u; u = up; up = um;
  }

  free (um); free (u); free (up);
  MPI_Finalize ();
  return 0;
}
```

## 5.5 How to Create Parallel FD Solvers in Diffpack

The process of building a parallel finite difference solver in Diffpack can be summarized as follows. First, one develops a standard sequential solver, using grid and field abstractions (`GridLattice`, `FieldLattice` etc, see [6, Ch. 1.6.5]). The "heart" of such a solver is the loops implementing the explicit finite difference scheme. These loops should be divided into two parts: loops over the inner points and loops over the boundary points. As we shall see later, this is essential for parallel programming. The solver must also be capable of dealing with an arbitrary rectangular grid (a trivial point when using Diffpack grids and fields).

We recall (cf. [6, Ch. 7.2]) that it is customary in Diffpack to assemble independent solvers for the different components in solving a system of partial differential equations. It is therefore useful to have a `scan` function in the solver that can either create the grid internally or use a ready-made external grid. We want the sequential solver to solve the partial differential equations on its own, hence it must create a grid. On the other hand, when it is a part of a parallel algorithm, the grid is usually made by some tool that has an overview of the global grid and the global solution process. This type of flexibility is also desirable for other data members of the solver (e.g. time parameters).

Having carefully tested a flexible sequential solver on the global domain, we proceed with the steps towards a parallel solver as follows:

1. Derive a subclass of the sequential solver that implements details specific to parallel computing. This subclass is supposed to be acting as subdomain solver. The code of the subclass is usually very short, since major parts of the sequential solver can be re-used also for parallel computing. Recall that this was our main goal of the mathematical formulation of the initial-boundary value problems over the subdomains. The subclass mainly deals with exchanging boundary values with neighboring subdomains.
2. Make a manager class for administering the subdomain solvers. The manager has the global view of the concurrent solution process. It divides the grid into subdomains, calls the initialization processes in the subdomain solver, and administers the time loop. The subdomain solver has a pointer to the manager class, which enables insight into neighboring subdomain solvers for communication etc. The code of the manager class is also short, mainly because it can be derived from a general toolbox for parallel programming of explicit finite difference schemes (see `ParallelFD` in Section 5.6).

## 5.6 Tools for Parallel Finite Difference Methods

*Flexible Assignment of Process Topology.* In many applications, the order in which the processes are arranged is not of vital importance. By default, a group of $P$ processes are given ranks from 0 to $P-1$. However, this linear ranking of the processes does not always reflect the geometry of the underlying numerical problem. If the numerical problem adopts a two- or three-dimensional grid, a corresponding two- or three-dimensional process grid would reflect the communication pattern better. Setting up the process topology can be complicated, and the required source code follows the same pattern in all the programs. In Diffpack, the topology functionality is offered by class `TopologyMPI`, which has a subclass `CartTopologyMPI`. By using `CartTopologyMPI` the user can organize the processes in a Cartesian structure with a row-major process numbering beginning at 0 as indicated in Figure 6. The `CartTopologyMPI` class also provides functions that return the rank of a process given its process coordinates and vice versa. We refer to the MPI documentation (e.g. [3]) for more information on the topology concept and functionality.

For example, the user can use the following user-friendly input format

```
CART d=2 [2,4] [ false false ]
```

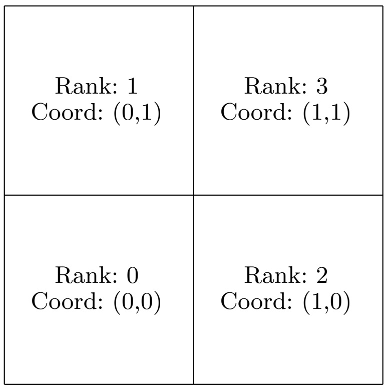

**Fig. 6.** Relation between coordinates and ranks for four processes in a $(2 \times 2)$ process grid.

for setting up a two-dimensional Cartesian process grid with $2 \times 4$ processes, non-periodic in both directions. We remark that being non-periodic in one direction means that the "head" processes of that direction do not communicate with the "tail" processes. We refer to MPI's `MPI_Cart_create`, `MPI_Cart_coords`, and `MPI_Cart_rank` functions for more details.

We remark that some parallel computers do not completely follow the multiprocessor model from Section 1.3. The communication network may be asymmetric, in that a processor has large bandwidth connections to some processors but small bandwidth to others. It is thus important to notice the difference between the logical topology (also called virtual topology) of processes and the underlying physical layout of the processors and the network. A parallel computer may exploit the logical process topology when assigning the processes to the physical processors, if it helps to improve the communication performance. If the user does not specify any logical topology, the parallel computer will make a random mapping, which may lead to unnecessary contention in the communication network. Therefore, setting up the logical topology may give performance benefits as well as large benefits for program readability.

*The Grid and Process Handler: ParallelFD.* Class `ParallelFD` is the parallel toolbox for parallelizing Diffpack FD solvers. It contains functionality generally needed for solving problems using finite difference methods on parallel computers. The `ParallelFD` class uses the basic parallel class `DistrProcManager`, which supplies information such as rank of the process and total number of processes. It also holds the topology object from which we can obtain information related to the multidimensional topology.

One of the main tasks of the manager class is to divide the global grid into subdomains. This general problem is supported by class `ParallelFD`.

The global grid is here divided in a manner such that all the local grids are approximately of the same size, ensuring that the load is evenly balanced between the processes. There are two versions of the function `initLocalGrid`, one for ordinary finite difference grids, and one for staggered grids. The latter version is more advanced and allows a different size of the overlap in the different space directions.

The `ParallelFD` class contains menu functionality for a `TopologyMPI` object and some very useful utility functions for computing the local grid for a processor

```
void initLocalGrid (GridLattice& local_grid,
                    const GridLattice& global_grid);
```

In addition, class `ParallelFD` has functionality for creating a global representation of the unknown field for visualization purposes (the global field can have coarser resolution than the local fields if there is not sufficient memory available). The function

```
void gatherFields(FieldLattice& local_res,
                  FieldLattice& global_res_u);
```

is then used to deliver a local part of a field to the global representation of the field.

*Communication Between Processes: The Points Classes.* A central issue in parallel algorithms for solving partial differential equations is to communicate values at the boundaries between subdomains. Some of the required functionality is common to a wide range of problems, and can hence be collected in a general class, here called `Points`, whereas other types of functionality depend on the particular solution method being used. The special functionality for explicit finite difference methods is collected in class `PointsFD`, which is a subclass of `Points`. The subclass `PointsFD` is mainly concerned with setting up the proper data structures for the information on boundary points.

The `Points` classes offer functionality for

– setting up data structures for the boundary points and values to be sent and received,
– sending boundary values to neighbors,
– receiving boundary values from neighbors.

Figure 7 shows a schematic design of a parallel Diffpack FD simulator. Assuming there exists an original sequential simulator `MyPDE`, the main work of parallelization consists in extending two subclasses from respectively `MyPDE` and `ParallelFD`. Subclass `MyPDEs` is to work as a subdomain solver and subclass `ManagerMyPDE` is a manager for communication related tasks. Note that the two subclasses are to work closely with `PointsFD`. We explain how these two subclasses can be written in the following text concerning the creation of a parallel heat conduction simulator.

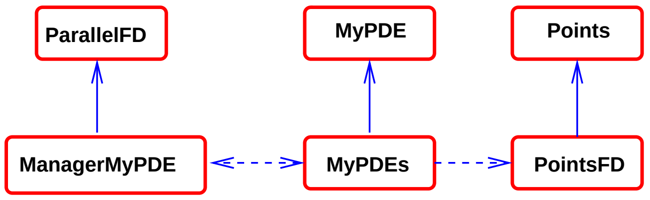

**Fig. 7.** Sketch of a sequential solver, its subclass, manager, and the toolboxes that these classes utilize. Solid arrow indicate class derivation ("is-a" relationship), whereas dashed arrows indicate pointers ("has-a" relationship).

## 5.7 Example: Heat Conduction in 2D

We consider a scaled heat conduction problem:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}, \quad (x, y) \in (0,1) \times (0,1),\ t > 0, \tag{18}$$

$$u(0, y, t) = 0, \quad t > 0, \tag{19}$$

$$u(1, y, t) = 0, \quad t > 0, \tag{20}$$

$$u(x, 0, t) = 0, \quad t > 0, \tag{21}$$

$$u(y, 1, t) = 0, \quad t > 0, \tag{22}$$

$$u(x, y, 0) = f(x, y)\,. \tag{23}$$

Following the notation used in the previous example, we define $u^k_{i,j}$ as the approximation to $u(x, y, t)$ on the spatial grid point $(x_i, y_j)$ at time step $t_k$, where the integers $i$, $j$ and $k$ are given by $i = 1, \ldots, n$, $j = 1, \ldots, m$ and $k \geq 0$, and we have $x_i = (i-1)\Delta x$, $y_j = (j-1)\Delta y$ and $t_k = k\Delta t$.

Discretization of the equations (18–23) using an explicit finite difference scheme gives

$$\begin{aligned}
u^0_{i,j} &= f(x_i, y_j), \quad i = 1, \ldots, n,\ j = 1, \ldots, m, \\
u^{k+1}_{i,j} &= u^k_{i,j} + \Delta t \left( [\delta_x \delta_x u]^k_{i,j} + [\delta_y \delta_y u]^k_{i,j} \right), \quad \begin{array}{l} i = 2, \ldots, n-1, \\ j = 2, \ldots, m-1, \end{array} \quad k > 0, \\
u^{k+1}_{1,j} &= u^{k+1}_{n,j} = 0, \quad j = 1, \ldots, m,\ k \geq 0, \\
u^{k+1}_{i,1} &= u^{k+1}_{i,m} = 0, \quad i = 1, \ldots, n,\ k \geq 0,
\end{aligned}$$

where

$$\begin{aligned}
[\delta_x \delta_x u]^k_{i,j} &\equiv \frac{1}{\Delta x^2} \left( u^k_{i-1,j} - 2u^k_{i,j} + u^k_{i+1,j} \right), \\
[\delta_y \delta_y u]^k_{i,j} &\equiv \frac{1}{\Delta y^2} \left( u^k_{i,j-1} - 2u^k_{i,j} + u^k_{i,j+1} \right).
\end{aligned}$$

We wish to split the domain $\Omega$ into $D$ subdomains $\Omega_s$, where $s = 1, \ldots, D$. However, since we now work with two-dimensional, rectangular, finite difference lattice grids, it is convenient to introduce a subdomain numbering using double indices $(p, q)$ in a "Cartesian" grid of $P \times Q$ subdomains. When we need to switch between a single index $s$ and the pair $(p, q)$, we assume that there is a mapping $\mu$ such that $s = \mu(p, q)$ and $(p, q) = \mu^{-1}(s)$. The subdomains can now be defined as follows.

$$\begin{gathered}
\Omega_{p,q} = [x_L^p, x_R^p] \times [y_B^q, y_T^q] \\
x_i^p = x_L^p + (i-1)\Delta x,\ i = 1, \ldots, n_p \\
y_j^q = y_B^q + (j-1)\Delta y,\ j = 1, \ldots, m_q \\
x_1^p = x_L^p,\ x_{n_p}^p = x_R^p,\ y_1^q = y_B^q,\ y_{m_q}^q = y_T^q, \\
x_L^1 = 0,\ x_R^P = 1,\ y_B^1 = 0,\ y_T^Q = 1\,.
\end{gathered}$$

Here, we have $p = 1, \ldots, P$, $q = 1, \ldots, Q$ and $D = PQ$.

Our numerical scheme has a five-point computational stencil, i.e., for each point, we only need values on points that are one grid cell away. This implies that the optimal overlap between the subdomains is one grid cell. For $p = 2, \ldots, P-1$ and $q = 2, \ldots, Q-1$ we then have the relations

$$\begin{gathered}
x_1^p = x_{n_{p-1}-1}^{p-1}, \quad x_2^p = x_{n_{p-1}}^{p-1}, \quad x_{n_p-1}^p = x_1^{p+1}, \quad x_{n_p}^p = x_2^{p+1}, \\
y_1^q = y_{m_{q-1}-1}^{q-1}, \quad y_2^q = y_{m_{q-1}}^{q-1}, \quad y_{m_q-1}^q = y_1^{q+1}, \quad y_{m_q}^q = y_2^{q+1}\,.
\end{gathered}$$

If either $P = 1$ or $Q = 1$ we note that we have a one-dimensional partitioning of the domain.

The set of neighboring subdomains $\mathcal{N}_{p,q} = \mathcal{N}_{\mu^{-1}(s)}$ is given by

$$\mathcal{N}_{p,q} = \{(p+1, q), (p-1, q), (p, q+1), (p, q-1)\}\,.$$

In conformity with the one-dimensional case (Section 5.4), we denote the interpolation operator by $I(x, y, t; \mathcal{N}_{p,q})$ which interpolates the numerical solution at $(x, y, t)$, where $(x, y)$ is in one of the neighboring subdomains $\Omega_{k,l}$ of $\Omega_{p,q}$, i.e., $(k, l) \in \mathcal{N}_{p,q}$. The numerical equations for subdomain $\Omega_{p,q}$ then read

$$\begin{aligned}
u_{i,j}^{p,q,k+1} &= u_{i,j}^{p,q,k} + \Delta t([\delta_x \delta_x u^{p,q}]_{i,j}^k + [\delta_y \delta_y u^{p,q}]_{i,j}^k), \quad \begin{array}{l} i = 2, \ldots, n_p - 1, \\ j = 2, \ldots, m_q - 1, \end{array} k > 0, \\
u_{1,j}^{p,q,k+1} &= I(x_1^p, y_j^q, t_{k+1}; \mathcal{N}_{p,q}), \quad j = 1, \ldots, m_q,\ k > 0, \\
u_{n_p,j}^{p,q,k+1} &= I(x_{n_p}^p, y_j^q, t_{k+1}; \mathcal{N}_{p,q}), \quad j = 1, \ldots, m_q,\ k > 0, \\
u_{i,1}^{p,q,k+1} &= I(x_i^p, y_1^q, t_{k+1}; \mathcal{N}_{p,q}), \quad i = 1, \ldots, n_p,\ k > 0, \\
u_{i,m_q}^{p,q,k+1} &= I(x_i^p, y_{m_q}^q, t_{k+1}; \mathcal{N}_{p,q}), \quad i = 1, \ldots, n_p,\ k > 0\,.
\end{aligned}$$

For subdomains that border with the physical boundaries, the following updating schemes apply for the points that lie on the physical boundaries:

$$\begin{aligned} u_{1,j}^{1,q,k+1} &= 0, \quad j = 1, \ldots, m_q,\ k \geq 0, \\ u_{n_P,j}^{P,q,k+1} &= 0, \quad j = 1, \ldots, m_q,\ k \geq 0, \\ u_{i,1}^{p,1,k+1} &= 0, \quad i = 1, \ldots, n_p,\ k \geq 0, \\ u_{i,m_Q}^{p,Q,k+1} &= 0, \quad i = 1, \ldots, n_p,\ k \geq 0\,. \end{aligned}$$

It is evident that a sequential program solving the global problem can be reused for solving the problem over each subdomain, provided that the domain can be an arbitrary rectangle and that we add functionality for communicating interpolated values between the subdomain solvers. If the subdomains have coinciding grid points, which is the usual case when working with basic finite difference methods, interpolation just means picking out the relevant grid point values from a neighboring subdomain.

## 5.8 The Sequential Heat Conduction Solver

Following standard Diffpack examples on creating finite difference solvers, see e.g. [6, Ch. 1.7], the heat equation simulator is typically realized as a C++ class with the following content.

```
class Heat2D
{
protected:
  Handle(GridLattice)  grid;     // uniform "finite difference" grid
  Handle(FieldLattice) u;        // the unknown field to be computed
  Handle(FieldLattice) u_prev;   // u at the previous time level
  Handle(TimePrm)      tip;      // time step etc.

  CPUclock clock;                // for timings within the program
  real compute_time;

  int i0, in, j0, jn;            // help variables for loops
  real mu, nu;                   // help variables in the scheme

  real initialField (real x, real y );   // initial condition func.

  virtual void setIC ();
  virtual void timeLoop ();
  void computeInnerPoints ();
  void computeBoundaryPoints ();
  void solveAtThisTimeStep ();
  void updateDataStructures () { *u_prev = *u; }

public:
  Heat2D ();
  virtual ~Heat2D () {}

  void scan (GridLattice* grid_ = NULL, TimePrm* tip_ = NULL);
  void solveProblem ();
  virtual void resultReport ();

};
```

In the above class declaration, we have introduced two non-standard functions `computeInnerPoints` and `computeBoundaryPoints`, which separate the computations on inner and boundary points. This is an important split of the implementation of the scheme, because non-blocking communication calls can then be used to hide the cost of communication. Another non-standard feature is that `scan` can make use of an external grid and external time integration parameters.

The particular numerical problem solved by class `Heat2D` is

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

on the unit square $\Omega = [0,1] \times [0,1]$ with $u = 0$ on the boundary. The initial condition reads

$$u(x,y,0) = \sin \pi x \sin \pi y \,.$$

The `computeInnerPoints` function typically takes the form

```
void Heat2D:: computeInnerPoints()
{
  int i, j;
  ArrayGenSel(real)& U  = u->values();
  ArrayGenSel(real)& Up = u_prev->values();
  for (j = j0+2; j <= jn-2; j++) {
    for (i = i0+2; i <= in-2; i++) {
      U(i,j) = Up(i,j) + mu*(Up(i-1,j) - 2*Up(i,j) + Up(i+1,j))
                       + nu*(Up(i,j-1) - 2*Up(i,j) + Up(i,j+1));
    }
  }
}
```

Here, `i0`, `j0`, `in`, and `jn` are precomputed start and stop indices for the loops over the grid points. We remark that the speed of the loop can be enhanced by working directly on the underlying C array as explained in [6, App. B.6.4]. As always, we emphasize that such optimizations should be performed after the code is thoroughly tested.

Notice that the `i` and `j` loops in `computeInnerPoints` touch neither the subdomain boundary points nor the ghost boundary points. Updating the solution at the subdomain boundaries takes place in `computeBoundaryPoints`:

```
void Heat2D:: computeBoundaryPoints()
{
  int i,j;
  ArrayGenSel(real)& U  = u->values();
  ArrayGenSel(real)& Up = u_prev->values();

  for (j = j0+1; j <= jn-1; j++) {
      U(i0+1,j) = Up(i0+1,j)
                + mu*(Up(i0,j) - 2*Up(i0+1,j) + Up(i0+2,j))
                + nu*(Up(i0+1,j-1) - 2*Up(i0+1,j) + Up(i0+1,j+1));
      U(in-1,j) = Up(in-1,j)
                + mu*(Up(in-2,j) - 2*Up(in-1,j) + Up(in,j))
                + nu*(Up(in-1,j-1) - 2*Up(in-1,j) + Up(in-1,j+1));
    }
    for (i = i0+1; i <= in-1; i++) {
      U(i,j0+1) = Up(i,j0+1)
                + mu*(Up(i-1,j0+1) - 2*Up(i,j0+1) + Up(i+1,j0+1))
                + nu*(Up(i,j0) - 2*Up(i,j0+1) + Up(i,j0+2));
      U(i,jn-1) = Up(i,jn-1)
                + mu*(Up(i-1,jn-1) - 2*Up(i,jn-1) + Up(i+1,jn-1))
                + nu*(Up(i,jn-2) - 2*Up(i,jn-1) + Up(i,jn));
    }

    // boundary values are never recalculated, hence the boundary
    // conditions will not be set at every time step
  }
```

## 5.9 The Parallel Heat Conduction Solver

*The Subclass Solver.* The basic finite difference schemes for updating the solution at inner and boundary points are provided by the sequential solver, which is class `Heat2D` in our current example. Communication of boundary values between processors is the only additional functionality we need for turning the sequential solver into parallel code. We add the communication functionality in a subclass `Heat2Ds`. Its header is like this:

```
class Heat2Ds: public HandleId, public Heat2D
{
  Ptv(int) coords_of_this_process; // my processor coordinates
  Ptv(int) num_solvers;            // # of procs in each space dir
  int s;                           // my subdomain number is s
  VecSimple(int) neighbors;        // identify my neighbors
  ManagerHeat2D* boss;             // my manager (for global info)

  PointsFD bc_points;              // boundary communication points

  void initBoundaryPoints ();      // init the bc_points object
  void setBC1 ();                  // send request for boundary values
  void setBC2 ();                  // receive & insert boundary values
  void solveAtThisTimeStep();

public:
  Heat2Ds (ManagerHeat2D* boss_);
 ~Heat2Ds () {}

  void scan();
  virtual void resultReport();

  friend class ManagerHeat2D;

};
```

Class `Heat2Ds` inherits the functionality from class `Heat2D`, but extends it for parallel computing:

– We have some data members for representing the processor coordinates of the current process, identification of neighbors etc.
– A `PointsFD` structure holds the boundary points for communication (the points to send and receive and their values).
– `initBoundaryPoints` initializes the `PointsFD` structure.
– `setBC1` applies the `PointsFD` structure for sending the boundary values to the neighbors.
– `setBC2` applies the `PointsFD` structure for receiving the boundary values from the neighbors.

Looking at the following parts of the source code, we observe that there is a close relation between the program abstractions and the formulation of the parallel algorithm:

```
void Heat2Ds:: setBC1 ()
{
  bc_points.fillBoundaryPoints(u());
  bc_points.sendValuesToNeighbors(neighbors);
}

void Heat2Ds:: setBC2 ()
{
  bc_points.receiveValuesFromNeighbors(neighbors.size());
  bc_points.extractBoundaryPoints(u_prev());
}

void Heat2Ds:: solveAtThisTimeStep()
{
  setBC1();
  computeInnerPoints();
  setBC2();
  computeBoundaryPoints();
  updateDataStructures();
}
```

The initialization of the `PointsFD` structure goes as follows.

```
void Heat2Ds:: initBoundaryPoints ()
{
  bc_points.initBcPointList(coords_of_this_process, num_solvers, *u);
  bc_points.initBcComm(neighbors, *u, coords_of_this_process);
}
```

*The Manager Class.* The particular `ManagerHeat2D`, inheriting the common `ParallelFD` functionality and making use of the `Heat2Ds` class, will be quite simple and can look like this:

```
class ManagerHeat2D : public ParallelFD, public SimCase
{

  friend class Heat2Ds;

  Handle(Heat2Ds) solver;

  // hold global grid information:
  Handle(GridLattice) global_grid;

  // global variables for the coarse grid used to report the results
  Handle(GridLattice)  global_resgrid;
  Handle(FieldLattice) global_res_u;

  Handle(TimePrm) tip;
  Handle(SaveSimRes) database;

  VecSimple(int) neighbors;

  void scan (MenuSystem& menu);
  void define (MenuSystem& menu, int level = MAIN);
  void timeLoop ();

public:

  ManagerHeat2D ();
  virtual ~ManagerHeat2D () {}

  virtual void adm (MenuSystem& menu);
  void gatherData (FieldLattice& local_res);
  void solveProblem ();
  void resultReport ();

};
```

The computational core of the `ManagerHeat2D` class is simply carried out by invoking the `solveAtThisTimeStep` belonging to class `Heat2Ds`. More specifically, we have

```
void ManagerHeat2D:: timeLoop ()
{
  tip->initTimeLoop();
  while (!tip->finished()) {
    tip->increaseTime();
    solver->solveAtThisTimeStep();
  }
}
```

Finally, the task of collecting the subdomain solutions becomes also simple by calling the `ParallelFD::gatherFields` function. In other words, we have

```
void ManagerHeat2D :: gatherData (FieldLattice& local_res)
{
  ParallelFD::gatherFields(local_res, *global_res_u);

  if (proc_manager->master())
```

```
    database->dump (*global_res_u, tip.getPtr());
}
```

## 6 Parallelizing FE Computations on Unstructured Grids

Finite element methods are often associated with unstructured computational grids. Compared with parallelizing codes for explicit finite difference schemes, see Section 5, the task of parallelizing codes for computation on unstructured grids is more demanding. Many new issues arise, such as general grid partitionings, distribution of unstructured data storage, complex inter-processor communication etc. Parallelizing a sequential finite element simulator from scratch is therefore a complicated and error-prone process.

To provide Diffpack users with a straightforward approach to parallelizing their sequential finite element simulators, we have devised an add-on toolbox that hides the cumbersome parallelization specific codes. Instead, the toolbox offers high-level and user-friendly functionality, such that a Diffpack user needs only to insert a few lines of code into his original sequential simulator to transform it into a parallel one. The parallel toolbox ensures that the computation-intensive operations are run in parallel at the linear algebra level. The resulting parallel simulator will maintain the same portability and flexibility as its sequential counterpart.

This section gives an introduction to the add-on parallel toolbox. We start with a mathematical explanation of how different linear algebra operations can be parallelized in a divide-and-conquer style. Then, we continue with a brief description of how object-oriented programming enables a seamless coupling between the parallel toolbox and the huge library of existing sequential Diffpack codes. Thereafter, we focus on some of the major functions of the toolbox. Finally, the actual parallelization process is demonstrated by an example, before we give answers to some frequently asked questions about this parallelization approach. We mention that there exists another high-level parallelization approach, which incorporates the mathematical framework of domain decomposition methods. The discussion of this topic will be left to another chapter [1] in this book.

### 6.1 Parallel Linear Algebra Operations

Let us consider for instance an elliptic boundary value problem discretized on an unstructured finite element grid. Suppose the solution process involves an iterative linear system solver, then the most computation intensive operations viewed at the level of matrices and vectors are: (i) calculation of element matrices and vectors, and (ii) solution of the resulting linear system of equations. Roughly speaking, the parallelization starts with partitioning the global finite element grid into a set of smaller subgrids to be hosted by different processors of a parallel computer. It is important to note that global matrices and vectors *need not to be constructed physically.* It suffices to deal only with sub-matrices and sub-vectors that are associated with the subgrids. Neighboring processors need to exchange information associated with the nodes that are shared between them. However, the amount of information exchange is rather limited. It will be shown in the following that the linear algebra operations on the original global matrices and vectors can be achieved by operations on the sub-matrices and sub-vectors plus inter-processor communications. We also note that the following parallelization approach and its implementation are only valid for Diffpack finite element simulators using iterative linear system solvers, not direct solvers as e.g. Gaussian elimination.

*Partitioning Unstructured Finite Element Grids.* The essence of parallel computing is to distribute the work load among the processors of a parallel computer. For finite element computations, the work load distribution arises naturally when the entire solution domain is decomposed into a number of subdomains, as a processor only carries out computation restricted to its assigned subdomain. Partitioning a global finite element grid primarily concerns dividing all elements into subgroups. The elements of the global finite element grid $\mathcal{M}$ are partitioned to form a set of smaller subgrids $\mathcal{SM}_i$, $1 \leq i \leq P$, where $P$ is typically the number of available processors. By a *non-overlapping partitioning*, we mean that each element of $\mathcal{M}$ is to belong to only one of the subgrids. Element edges shared between neighboring subgrids constitute the so-called *internal boundaries.* The boundary of a subdomain is either entirely internal or partly internal and partly physical. We note that although every element of $\mathcal{M}$ belongs to a single subgrid after a non-overlapping partitioning, the nodes lying on the internal boundaries belong to multiple subgrids at the same time. In Figure 8, the nodes marked with a circle denote such nodes.

Ideally, the entire work load should be divided evenly among the available processors. Since the work load of a finite element computation is proportional to the number of elements, the subgrids should have approximately the same number of elements. Due to the necessity of exchange of information associated with the internal boundary nodes, a perfect partitioning scheme should minimize both the number of neighbors and the number of internal boundary nodes for each subgrid. In this way, the communication overhead is minimized. General non-overlapping partitioning of unstructured finite element grids is a non-trivial problem. Here, we only mention that it is often transformed into a graph partitioning problem and solved accordingly, see e.g. [2,12].

*Distributed Matrices and Vectors.* When a non-overlapping partitioning of the unstructured global finite element grid is ready, one subgrid is to be held by one processor. The discretization of the target partial differential equation on one processor is done in the same way as in the sequential case, only that

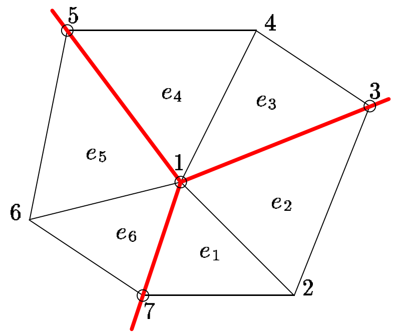

**Fig. 8.** An example of internal boundary nodes that are shared between neighboring subgrids. The figure shows a small region where three subgrids meet. Elements $e_1$ and $e_2$ belong to one subgrid, $e_3$ and $e_4$ belong to another subgrid, while $e_5$ and $e_6$ belong to a third subgrid. Nodes marked with a circle are internal boundary nodes.

the discretization is restricted to the assigned subdomain. The assembly of the local stiffness matrix and right-hand side vector needs contribution *only* from all the elements belonging to the current subgrid. So far no inter-processor communication is necessary. Let us denote the number of nodes in subgrid $\mathcal{SM}_i$ by $N_i$. For any scalar partial differential equation to be discretized on subgrid $\mathcal{SM}_i$, we will come up with a linear system of equations

$$\mathbf{A}_i\mathbf{x}_i = \mathbf{b}_i,$$

where the local stiffness matrix $\mathbf{A}_i$ is of dimension $N_i \times N_i$, and the local right-hand side vector $\mathbf{b}_i$ and solution vector $\mathbf{x}_i$ have length $N_i$. The result of the parallel finite element solution process will be that $\mathbf{x}_i$ holds the correct solution for all the nodes belonging to $\mathcal{SM}_i$. The solution values for the internal boundary nodes are duplicated among neighboring subdomains. In comparison with $\mathbf{A}$ and $\mathbf{b}$, which would have arisen from a discretization done on the entire $\mathcal{M}$, entries of $\mathbf{A}_i$ and $\mathbf{b}_i$ that correspond to the *interior nodes* of $\mathcal{SM}_i$ are identical with those of $\mathbf{A}$ and $\mathbf{b}$. For entries of $\mathbf{A}_i$ and $\mathbf{b}_i$ that correspond to the internal boundary nodes, they are different from those of $\mathbf{A}$ and $\mathbf{b}$, because the contributions from the elements that lie in neighboring subgrids are missing. However, this does not prevent us from obtaining the correct result of different linear algebra operations that should have been carried out on the *virtual* global matrices and vectors. It is achieved by operating solely on local matrices and vectors plus using inter-processor communication, as the following text will show.

*Three Types of Parallel Linear Algebra Operations.* At the linear algebra level, the objective of the parallel solution process is to find the solution of the global system $\mathbf{Ax} = \mathbf{b}$ through linear algebra operations carried out only on the local systems $\mathbf{A}_i\mathbf{x}_i = \mathbf{b}_i$. We emphasize that global matrices and vectors have no physical storage but are represented virtually by local matrices and vectors that are held by the different processors. For iterative solution algorithms, it suffices to parallelize the following three types of global linear algebra operations:

1. Vector addition: $\mathbf{w} = \mathbf{u} + \alpha\mathbf{v}$, where $\alpha$ is a scalar constant;
2. Inner-product between two vectors: $c = \mathbf{u} \cdot \mathbf{v}$;
3. Matrix-vector product: $\mathbf{w} = \mathbf{A}\mathbf{u}$.

*Parallel vector addition.* Assuming that values of $\mathbf{u}_i$ and $\mathbf{v}_i$ on the internal boundary nodes are correctly duplicated between neighboring subdomains, the parallel vector addition $\mathbf{w}_i = \mathbf{u}_i + \alpha\mathbf{v}_i$ is straightforward and needs no inter-processor communication (see Section 1.4).

*Parallel inner-product between two vectors.* The approach from Section 1.4 is to be used. However, the result of the local inner-product $c_i = \sum_j^{N_i} u_{i,j}v_{i,j}$ can not be added together directly to give the global result, due to the fact that internal boundary nodes are shared between multiple subgrids. Vector entries on those internal boundary nodes must be scaled accordingly. For this purpose, we denote by $\mathcal{O}_i$ the set of all the internal boundary nodes of $\mathcal{SM}_i$. In addition, each internal boundary node has an integer count $o_k$, $k \in \mathcal{O}_i$, which is the total number of subgrids it belongs to, including $\mathcal{SM}_i$ itself. Then we can get the adjusted local result by

$$\tilde{c}_i = c_i - \sum_{k\in\mathcal{O}_i} \frac{o_k - 1}{o_k} u_{i,k}v_{i,k} \,.$$

Thereafter, the correct result of $c = \mathbf{u}\cdot\mathbf{v}$ can be obtained by collecting all the adjusted local results in form of $c = \sum_i^P \tilde{c}_i$. This is done by inter-processor communication in form of an all-to-all broadcast operation.

*Parallel matrix-vector product.* We recall that a global matrix $\mathbf{A}$ does not exist physically, but is represented by a series of local matrices $\mathbf{A}_i$. These local matrices arise from a local assembly process that is restricted to each subdomain. The rows of $\mathbf{A}_i$ that correspond to the interior subgrid nodes are correct. A local matrix-vector product $\mathbf{w}_i = \mathbf{A}_i\mathbf{u}_i$ will thus give correct values of $\mathbf{w}_i$ for the interior subgrid nodes. However, the rows of $\mathbf{A}_i$that correspond to the internal boundary nodes are only partially correct. For a specific internal boundary node, some of the elements that share this node belong to the neighboring subdomains. The contributions from the element matrices are thus distributed among the neighboring subdomains. Therefore, the correct value in $\mathbf{w}_i$ for an internal boundary node can only be obtained by adding up contributions from *all* the neighboring subdomains. For instance, in Figure 8, the nodes with global numbers 3,5,7 need contributions from two neighbors, while the node with global number 1 should add together contributions from all three neighbors. This requires inter-processor communication in form of two and two neighboring subdomains exchanging values on the relevant internal boundary nodes.

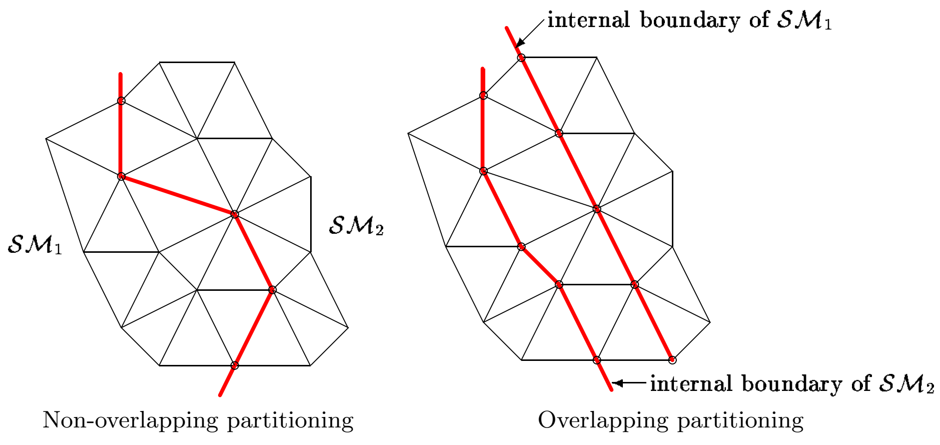
internal boundary of $\mathcal{SM}_1$
$\mathcal{SM}_1$
$\mathcal{SM}_2$
internal boundary of $\mathcal{SM}_2$
Non-overlapping partitioning
Overlapping partitioning

**Fig. 9.** Extension of a non-overlapping partitioning of a finite element grid into an overlapping partitioning.

*Preconditioning and Overlapping Grid Partitioning.* Iterative solvers for linear systems often use preconditioners to speed up the convergence. Solving linear systems in parallel requires therefore parallel preconditioning operations. This is an important but tricky issue. We will leave the discussion of parallel multilevel preconditioners such as domain decomposition methods to a later chapter [1], and consider the other Diffpack preconditioners here. It should be pointed out immediately that many preconditioners such as SSOR relaxation and RILU are inherently sequential algorithms. A remedy is to just let each processor run the corresponding localized preconditioning operations, and then take some kind of average of the different values associated with each internal boundary node from neighboring subgrids, see e.g. [9]. However, a different and possibly weakened preconditioning effect may be expected.

Careful readers will notice, however, that even the intrinsically parallel Jacobi relaxation does not produce the correct result on a non-overlapping partitioning, because the internal boundary nodes will not be updated correctly. That is, the need for an *overlapping* partitioning of $\mathcal{M}$ arises. More specifically, the non-overlapping subgrids are extended slightly so that every node of $\mathcal{M}$, which does not lie on the physical boundary, is an interior node of at least one subgrid. Figure 9 shows an example of extending a non-overlapping partitioning and making it overlapping. A parallel Jacobi relaxation on an overlapping partitioning will be done as follows: each subdomain runs *independently* its local Jacobi relaxation, then local results on the internal boundary nodes are discarded and replaced by the correct results sent from the neighboring subdomains, which have those nodes as their interior grid nodes.

## 6.2 Object-Oriented Design of a Parallel Toolbox

It is now clear that at the linear algebra level the global finite element computational operations can be replaced with localized operations plus necessary inter-processor communication. This is good news regarding the parallelization of Diffpack finite element computations, because we can directly utilize the existing sequential codes for carrying out all the local linear algebra operations. Among other things we are able to use the *same* Diffpack classes for storing the local matrices and vectors, and all the iterative methods for solving linear systems can be used in the same way as before. Of course, new codes need to be written to incorporate two major parallelization specific tasks: general grid partitioning and different forms of inter-processor communication. In this section, we present the design of a small add-on parallel toolbox that contains a number of new C++ classes capable of these two tasks. The add-on toolbox can be coupled seamlessly with the existing huge sequential Diffpack library, thereby allowing Diffpack users to parallelize their finite element computation in a straightforward way. In this connection, object-oriented programming techniques have proved to be successful.

*Coupling with the Sequential Diffpack Library.* One of our objectives is to maintain the comprehensive sequential library of Diffpack independently of the parallel toolbox. The only new class that we have introduced into the sequential Diffpack library is a pure interface class with name `SubdCommAdm`. The following simplified class definition gives an overview of its most important member functions:

```
class SubdCommAdm : public HandleId
{
protected:
  SubdCommAdm () {}
public:
  virtual ~SubdCommAdm () {}
  virtual void updateGlobalValues (LinEqVector& lvec);
  virtual void updateInteriorBoundaryNodes (LinEqVector& lvec);
  virtual void matvec (const LinEqMatrix& Amat,
                       const LinEqVector& c,
                             LinEqVector& d);
  virtual real innerProd (LinEqVector& x, LinEqVector& y);
  virtual real norm (Vec(real)& c_vec, Norm_type lp=l2);
};
```

We note that the constructor of class `SubdCommAdm` is a protected member function, meaning that no instance of `SubdCommAdm` can be created. In addition, all its member functions, which are virtual, have empty definitions. Therefore, it only serves as a pure interface class defining the name and syntax of different functions for inter-processor communication. Inside the parallel toolbox, we have developed a subclass of `SubdCommAdm` in order to implement such communication functions. Here, we mention that the member function `updateInteriorBoundaryNodes` will invoke an inter-processor communication, which ensures that all the internal boundary nodes to have correctly duplicated values among neighboring subgrids. In contrast, the `updateGlobalValues` function concerns overlapping partitionings, i.e., it is meant for all the overlapping nodes shared among neighboring subgrids. Moreover, the member functions `matvec`, `innerProd`, and `norm` are meant to carry out the parallel versions of the matrix-vector product, the inner product between two vectors, and the norm of a vector, respectively. These member functions work for both non-overlapping and overlapping partitionings.

It suffices for the rest of the sequential Diffpack library to only know how inter-processor communication can be invoked following the definition of class `SubdCommAdm`. A few of the linear algebra classes in the sequential Diffpack library were slightly extended to allow a connection with the add-on parallel toolbox. The rest of the sequential Diffpack library classes remain intact. For example, class `LinEqSystemPrec` is now equipped with the following two lines in its definition:

```
  Handle(SubdCommAdm) comm_adm;
  void attachCommAdm (const SubdCommAdm& adm_);
```

The member function `attachCommAdm` can be used to set the internal smart pointer `comm_adm` to a concrete object of type `SubdCommAdm`. Consequently, the member function `LinEqSystemPrec::matvec` is modified as follows:

```
void LinEqSystemPrec::matvec(const LinEqVector& c, LinEqVector& d)
{
  if (comm_adm.ok())
    comm_adm->matvec (*Amat, c, d);  // parallel computation
  else
    Amat->prod (c, d);               // sequential computation
  matvecCalls++;
}
```

It can be observed that, during a sequential Diffpack simulation, the test

```
  if (comm_adm.ok())
```

should return a `false` value, so the parallel version of the matrix-vector product will not be invoked. The situation is the opposite during a parallel Diffpack simulation. It is also important to note that these `if`-tests are hidden from the user and have completely negligible overhead.

To relieve Diffpack users of the burden of inserting the `attachCommAdm` function in many places when parallelizing a sequential Diffpack simulator, the member function `LinEqAdm::attachCommAdm` makes sure that all the involved linear algebra classes call their respective `attachCommAdm` functions. So, for the application programmer, one explicit call of `LinEqAdm::attachCommAdm` is enough before the parallel solution process starts.

*The Parallel Toolbox.* The two main tasks of the toolbox are to provide Diffpack users with different grid partitioning methods and several high-level inter-processor communication functions. In addition, the toolbox also provides parallel versions of some of Diffpack's most frequently used linear algebra operations.

*A hierarchy of grid partitioning methods.* The purpose of grid partitioning is to provide subdomains with subgrids where local discretization can be carried out. Although the most common situation is to start with a global finite element grid and partition it into smaller subgrids, there can be cases where Diffpack users wish to let each subdomain create its local finite element grid directly either by reading an existing Diffpack `GridFE` file or meshing the subdomain based on sufficient geometry information. In order to incorporate different grid partitioning approaches, we have created a class hierarchy with a base class named `GridPart`, whose simplified definition is as follows.

```
class GridPart
{
public:
  GridPart (const GridPart_prm& pm);
  virtual ~GridPart () {}
  virtual bool makeSubgrids ()=0;
  virtual bool makeSubgrids (const GridFE& global_grid)=0;
};
```

We can see that `GridPart` is a so-called pure virtual class in C++, because both versions of the member function `makeSubgrids` are required to be implemented in a derived subclass. At run time, the real work of grid partitioning is done by one of the `makeSubgrids` functions in a specific subclass decided by the Diffpack user. The input argument to the constructor is an object of type `GridPart_prm`, which contains diverse grid partition-related parameters. The following subclasses of `GridPart` are already included in the hierarchy, which also allows a Diffpack user to extend it by developing new subclasses.

1. `GridPartUnstruct`. This class creates subgrids by partitioning an unstructured global finite element grid. The number of resulting subgrids can be arbitrary and is decided by the user at run time. Class `GridPartUnstruct` is in fact a pure virtual class itself. Different graph partitioning methods are implemented in different subclasses of `GridPartUnstruct`. One example is subclass `GridPartMetis` where the METIS algorithm [5] is implemented.
2. `GridPartUniform`. This class creates subgrids by a rectangular partitioning of a structured global grid. The subgrids will also be structured so that the user has more control over the partitioning. This simple and flexible partitioning method allows also a 2D or 3D structured global grid to be partitioned in fewer directions than its actual number of spatial dimensions.
3. `GridPartFileSource`. This class reads ready-made subgrids from Diffpack `GridFE` data files. All the grid data files should be of the format accepted by the standard Diffpack `readOrMakeGrid` function.

*The main control class.* We recall that `SubdCommAdm` is a pure interface class; all its inter-processor communication functions need to be implemented in a subclass before they can be used in parallel Diffpack finite element computations. For this purpose, we have created a subclass of `SubdCommAdm` in the parallel toolbox and named it `GridPartAdm`. Here, object-oriented programming sets `SubdCommAdm` as the formal connection between the existing sequential Diffpack library and the parallel toolbox, while `GridPartAdm` is the actual working unit. It should also be mentioned that class `GridPartAdm` not only implements the different inter-processor communication functions, but at the same time provides a unified access to the different methods offered by the `GridPart` hierarchy. In this way, it suffices for a user to only work with `GridPartAdm` during parallelization of a sequential Diffpack simulator, without making explicit usage of the details of `GridPart` and `GridPart_prm` etc. A simplified definition of class `GridPartAdm` is as follows.

```
class GridPartAdm : public SubdCommAdm
{
protected:
  Handle(GridPart_prm) param;
  Handle(GridPart) partitioner;
  bool overlapping_subgrids;

public:
  GridPartAdm ();
  virtual ~GridPartAdm ();
  void copy (const GridPartAdm& adm);
  static  void defineStatic  (MenuSystem& menu, int level = MAIN);
  virtual void scan          (MenuSystem& menu);
  virtual int getNoGlobalSubds() {return param->num_global_subds;}
  virtual bool overlappingSubgrids(){return overlapping_subgrids;}
  virtual void prepareSubgrids ();
  virtual void prepareSubgrids (const GridFE& global_grid);
  virtual GridFE& getSubgrid ();
  virtual void prepareCommunication (const DegFreeFE& dof);
  virtual void updateGlobalValues (LinEqVector& lvec);
  virtual void updateInteriorBoundaryNodes (LinEqVector& lvec);
  virtual void matvec (const LinEqMatrix& Amat,
                       const LinEqVector& c,
                             LinEqVector& d);
  virtual real innerProd (LinEqVector& x, LinEqVector& y);
  virtual real norm (LinEqVector& lvec, Norm_type lp=l2);
};
```

We can see from above that class `GridPartAdm` has smart pointers to objects of `GridPart_prm` and `GridPart`, through which it administers the process of grid partitioning. Inside the `GridPartAdm::prepareSubgrids` function, an object of `GridPart` is instantiated to carry out the grid partitioning. The actual type of the grid partitioner is determined *at run time* according to the user's input data. We refer to Figure 10 for a schematic diagram describing the relationship between `SubdCommAdm`, `GridPartAdm`, and `GridPart`.

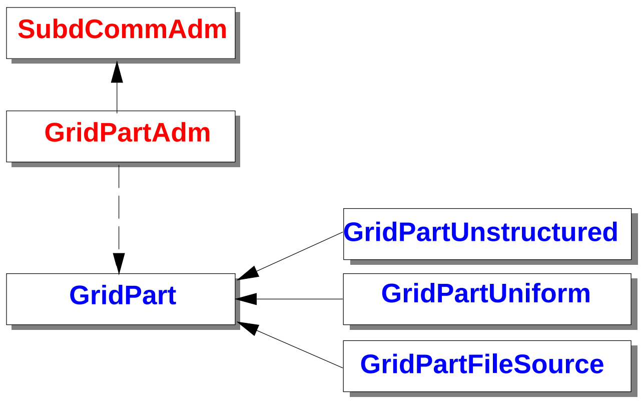

**Fig. 10.** The relationship between `SubdCommAdm`, `GridPartAdm`, and `GridPart`. Note that the solid arrows indicate the "is-a" relationship, while the dashed arrows indicates the "has-a" relationship.

### 6.3 Major Member Functions of GridPartAdm

During parallelization of most sequential Diffpack simulators, an object of `GridPartAdm` is all a user needs in order to access the desired functionality of the parallel toolbox. We will here give a detailed description of the major functions of `GridPartAdm`.

*Grid Partition Related Member Functions.* Since there are many parameters associated with the different grid partitioning approaches, the Diffpack `MenuSystem` class is heavily involved in this process. Thus, almost every parallel Diffpack simulator will need such a statement in its `define` function:

```
GridPartAdm:: defineStatic (menu, level+1);
```

Then, typically inside the corresponding `scan` function, the work of grid partitioning can be carried out as follows:

```
// read information from the input menu
gp_adm->scan (menu);
// assuming that grid is pointing to the global grid
gp_adm->prepareSubgrids (*grid);
// now let grid point to the subgrid
grid.rebind (gp_adm->getSubgrid());
```

Here, `gp_adm` is a `Handle` to a `GridPartAdm` object. The member function `prepareSubgrids` has the responsibility of carrying out the grid partitioning, after which the member function `getSubgrid` will return the reference of the resulting subgrid on every processor. The function `prepareSubgrids` makes use of different grid parameters that are supplied by the member function pair `GridPartAdm::defineStatic` and `GridPartAdm::scan`. In this way, the user can make a flexible choice among different grid partitioning methods at run time. The following are examples of parameter input files for three different ways of producing the subgrids:

1. General partitioning of an unstructured global finite element grid:
   ```
   sub GridPart_prm
   set grid source type = GlobalGrid
   set partition-algorithm = METIS
   set number overlaps = 1
   ok
   ```
   The input value `GlobalGrid` indicates that a global finite element grid is to be partitioned into an arbitrarily given number of subgrids. The number of subgrids is equal to the number of involved processors. For example a run-time command such as
   ```
   mpirun -np 8 ./app
   ```
   means that eight subgrids will be created, each residing on one processor. The above parameter input file also says that the method of the METIS package will be used to do the general grid partitioning and the resulting non-overlapping partitioning is to be extended to create an overlapping partitioning.
2. User-controlled rectangular partitioning of a uniform grid:
   ```
   sub GridPart_prm
   set grid source type = UniformPartition
   set subdomain division = d=2 [0,1]x[0,1] [0:4]x[0:2]
   set overlap = [1,1]
   set use triangle elements = ON
   ok
   ```
   The above parameter input file implies that a global 2D uniform mesh, in form of `GridFE`, will be partitioned into $4 \times 2 = 8$ uniform local grids. We remark that the redundant geometry information (`[0,1]x[0,1]`) on the `set subdomain division` line is ignored by the partitioner. For this example, the parallel finite element application must be started on 8 processors, each taking care of one subdomain. The number of partitions is 4 in the $x$-direction and 2 in the $y$-direction. The resulting subgrids will all have 2D triangular elements, and one element overlap layer exists between the neighboring subgrids in both the $x$- and $y$-directions.
3. Creation of subgrids directly from grid data files:
   ```
   sub GridPart_prm
   set grid source type = FileSource
   set subgrid root name = subd%02d.grid
   ok
   ```
   An assumption for using the above input file is that we have a series of grid data files of format `GridFE`, named `subd01.grid`, `subd02.grid`, and so on. Note also that for this grid partitioning approach, we have to use the version of the `GridPartAdm::prepareSubgrids` function that takes no `GridFE` input argument. This is because no global finite element grid exists physically. In other words, the work of grid partitioning is done by
   ```
     gp_adm->scan (menu);
     gp_adm->prepareSubgrids ();
     grid.rebind (gp_adm->getSubgrid());
   ```

*Communication-Related Member Functions.* The communication pattern between neighboring subgrids that arise from a general grid partitioning is complex. However, for a fixed partitioning, whether overlapping or not, the communication pattern is fixed throughout the computation. The member function `prepareCommunication` thus has the responsibility of finding out the communication pattern *before* the parallel computation starts. One example of invoking the function is

```
gp_adm->prepareCommunication (*dof);
```

where `dof` is a `Handle` to a `DegFreeFE` object. That is, the function takes a Diffpack `DegFreeFE` object as the input argument. For each subdomain, this function finds out all its neighboring subdomains. Additionally, the function also finds out how to exchange information between each pair of neighboring subdomains. The storage allocation for the outgoing and incoming messages is also done in `prepareCommunication`. After that, the member function `attachCommAdm` belonging to class `LinEqAdm` should be called such as:

```
lineq->attachCommAdm (*gp_adm);
```

The design of the parallel toolbox has made the parallelization process extremely simple for the user. The above statement is sufficient for ensuring that necessary inter-processor communication will be carried out by the standard Diffpack libraries when solving the linear systems of equations. However, in certain cases a user needs to explicitly invoke one of the two following functions of `GridPartAdm` for inter-processor communication.

`updateGlobalValues` ensures that all the nodes that are shared between multiple subgrids have correctly duplicated values. For non-overlapping grid partitionings, this means that for a particular internal boundary node, the different values coming from the neighboring subgrids are collected and summed up. For overlapping grid partitionings, every internal boundary node throws away its current value and replaces it with a value provided by a neighboring subgrid, which has an interior node with the same coordinates. If an internal boundary node *lies in the interior* of more than one neighboring subgrid, an average of the values, which are provided by these neighboring subgrids, becomes the new value.

`updateInteriorBoundaryNodes` has the same effect for non-overlapping grid partitionings as `updateGlobalValues`. For overlapping grid partitionings, the function only applies to the internal boundary nodes, whereas values of the other nodes in the overlapping regions are unaffected.

The member functions `matvec`, `innerProd`, and `norm` of class `GridPartAdm` can be used to carry out the parallel version of respectively matrix-vector product, inner product between two vectors, and norm of a vector. Inside these functions, local linear algebra operations are carried out by invoking the corresponding sequential Diffpack functions before some inter-processor communication, see also Section 6.2.

## 6.4 A Parallelization Example

We consider a standard sequential Diffpack finite element simulator `Poisson1` from [6]. The parallelization is quite straightforward, so it is sufficient to insert parallelization specific codes directly into the original sequential simulator, enclosed in the preprocessor directive

```
#ifdef DP_PARALLEL_LA
...
#endif
```

In this way, we will be able to maintain a single code for both the sequential and parallel simulators.

The first thing to do is to insert the following line in the beginning of the header file:

```
#ifdef DP_PARALLEL_LA
#include <GridPartAdm.h>
#endif
```

The next thing to do is to include a handle of a `GridPartAdm` object in the definition of class `Poisson1`, i.e.,

```
#ifdef DP_PARALLEL_LA
  Handle(GridPartAdm) gp_adm;
#endif
```

Thereafter, three modifications remain to be done in the original sequential code. The first modification is to put inside the `define` function the following line:

```
#ifdef DP_PARALLEL_LA
  GridPartAdm::defineStatic (menu,level+1);
#endif
```

The second code modification is to let the `GridPartAdm` object produce the subgrid, i.e., inside the `scan` function:

```
#ifdef DP_PARALLEL_LA
  gp_adm.rebind (new GridPartAdm);
  gp_adm->scan (menu);
  gp_adm->prepareSubgrids (*grid);
  grid.rebind (gp_adm->getSubgrid());
#endif
```

The third required code modification is to add the following two lines of new code, after the `LinEqAdmFE` object is created and its `scan` is called:

```
#ifdef DP_PARALLEL_LA
  gp_adm->prepareCommunication (*dof);
  lineq->attachCommAdm (*gp_adm);
#endif
```

Here, the first of the above two new code lines invokes the member function `GridPartAdm::prepareCommunication`, which must precede all the subsequent inter-processor communications. The second line attaches the `GridPartAdm` object to `lineq`, which points to a Diffpack `LinEqAdmFE` object. As mentioned earlier in Section 6.2, the `LinEqAdmFE` object will then automatically invoke all the other needed `attachCommAdm` functions, including e.g. the one belonging to `LinEqSystemPrec`. In this way, necessary communication and synchronization will be enforced during the solution process later.

The final task is to insert

```
initDpParallelLA (nargs, args);
```

as the first statement in the main program and insert

```
closeDpParallelLA();
```

at the end.

## 6.5 Storage of Computation Results and Visualization

The storage of computation results during a parallel Diffpack simulation can be carried out exactly as before, e.g. through the usage of `SaveSimRes`. The only exceptions are the use of `SaveSimRes::lineCurves` and time series plot for a given spatial point. These should be avoided in a parallel Diffpack simulator. For each processor involved in a parallel Diffpack simulation, the global `casename` variable will be automatically suffixed with `_p0000`, `_p0001`, ..., for respectively processor number one, number two, and so on. That is, each processor has a unique `casename` variable and therefore operates with its own `SimResFile`. In multiple-loop computations, the processor rank is added in front of the multiple-loop identification number, like `_p0000_m01`, `_p0001_m01`.

*Four Useful Tools.* As explained above, each processor generates its own `SimResFile` during a parallel simulation. When the parallel simulation is done, the user will face a series of `SimResFile` databases, possibly after collecting them from different hard disks. To visualize such parallel simulation results, using e.g. Vtk, it is necessary to run the `simres2vtk` filter for each of the `SimResFile`s. In order to relieve the user's effort of having to filter *all* the `SimResFile`s one by one, we have devised a tool named `distrsimres`. This is a Perl script that automatically visits all the `SimResFile`s. An example of using `distrsimres` can e.g. be:

```
distrsimres simres2vtk -f SIMULATION -E -a
```

That is, the user just needs to prefix any `simres2xxx` command with the `distrsimres` script. Similarly, for a multiple-loop simulation, it is necessary to add the loop-number specification to the `casename`, like e.g.

```
distrsimres simres2vtk -f SIMULATION_m02 -E -a
```

The second tool is another Perl script with name `distrsummary`. It can be used to generate a single HTML summary file that contains links to all the `casename_p00xx-summary.html` files, which will be generated by a parallel simulation that uses Diffpack's automatic report generation functionality. The syntax is simply

```
distrsummary -f casename
```

The third tool is a compiled Diffpack application, named `distr2glob`, which can be used to "patch together" chosen field(s) that are stored distributedly in the different `SimResFile`s. To use `distr2glob`, the user specifies a global grid, typically a quite coarse grid of type `GridLattice`. Then `distr2glob` maps the distributed `SimResFile` data onto the specified global coarse grid. The result is a new `SimResFile` that contains the chosen field(s) on the global coarse grid. An example execution of `distr2glob` can be

```
distr2glob -gg coarse.grid -const 0 -f SIMULATION -r 1 -s -a
```

The effect is that the first scalar field (due to the `-r` option), which is stored distributedly in the different `SimResFile`s that share the root casename `SIMULATION` (due to the `-f` option), is mapped onto a global coarse grid given by the grid file `coarse.grid` (due to the `-gg` option) and stored eventually in a new `SimResFile`. In short, the user can use most of the options available for a standard `simres2xxx` filter, such as `-r`, `-t`, `-A`, and `-E`, to choose wanted subdomain field(s). In addition, the new option `-gg` takes a text string of the format acceptable by Diffpack's `readOrMakeGrid` command and the other new option `-const` reads a default value to be assigned to those grid points of the global coarse grid lying outside the original global solution domain.

The fourth tool is a new version of `RmCase` for parallel simulations. By specifying the value of `casename` such as `SIMULATION` or `SIMULATION_m01`, the user can remove all the files related to `casename`.

*A Simple GUI for Visualization.* We have mentioned that the `distr2glob` tool can allow the user to rapidly check the parallel data set against a global coarse grid. However, for more detailed visualization of the parallel computation results, we have created a new Perl script similar to `vtkviz`. The new Perl script is named `distrvtkviz` and its work is to run `vtk` with an input Vtk-Tcl script named `ParaViewer.tcl`. We recall that the work of `vtkviz` is to run `vtk` with `Examiner.tcl` as input. The new features of `ParaViewer.tcl` are that the user can freely add/remove sub-grid data set(s) and concentrate on a particular sub-grid, if desired. To use `distrvtkviz`, the only thing a user needs to do is to run the `distrsimres simres2vtk` command, before starting the GUI by issuing the command

```
distrvtkviz
```

## 6.6 Questions and Answers

1. *When should I use a non-overlapping partitioning of an unstructured finite element grid, when to use an overlapping partitioning?*
   In principle, using an overlapping partitioning is always a *safe* approach in that parallelization of certain operations can not achieve correct result on a non-overlapping partitioning. An example is parallel Jacobi relaxation. However, an overlapping partitioning results in more computation per processor. So it is advantageous to first check whether the two different partitions produce the same result on a coarse global finite element grid before running full-scaled simulations in parallel.
2. *How do I produce a 2D partitioning of a 3D uniform finite element grid?*
   Assume that a $100 \times 100 \times 100$ uniform mesh covering the unit cube is desired to be partitioned into 10 subgrids. The number of partitions in the $x-$, $y-$, and $z-$directions are 5, 2, 1, respectively. In addition, one layer of element overlap is desired between the neighboring subgrids. Then the following information can be used in the input parameter file:
   ```
   sub GridPart_prm
   set grid source type = UniformPartition
   set subdomain division =d=3 [0,1]x[0,1]x[0,1] [0:5]x[0:2]x[0:1]
   set overlap = [1,1,0]
   ok
   ```
3. *When should I use* `GridPartAdm::updateGlobalValues`*?*
   For non-overlapping partitionings, the two functions `updateGlobalValues` and `updateInteriorBoundaryNodes` of class `GridPartAdm` have exactly the same effect. For overlapping partitionings, some situations only require that internal boundary nodes of a subgrid should receive their correct values from neighboring subgrids, where those nodes are interior subgrid nodes. More precisely, the functionality of `updateInteriorBoundaryNodes` is a subset of that of `updateGlobalValues`. So `updateGlobalValues` is always safe to use but may introduce some unnecessary communication overhead.
4. *What do I do when I want to work with both scalar and vector fields in my Diffpack simulator?*
   Assume that all the scalar fields use one `DegFreeFE` object and all the vector fields use another `DegFreeFE` object, while the two `DegFreeFE` objects share the same `GridFE` object. In this situation, two objects of type `GridPartAdm` are needed, one for all the scalar fields, the other for all the vector fields. Since all the fields need to work on the same partitioning of the global finite element grid, only one `GridPartAdm` will have the responsibility of doing the grid partitioning, whereas the other `GridPartAdm` copies the partitioning.
   ```
     Handle(GridPartAdm) adm1, adm2;
     Handle(GridFE) grid;
     Handle(DegFreeFE) dof1, dof2;
     Handle(LinEqAdm) lineq1, lineq2;
     // ....
     adm1->scan (menu);
     adm1->prepareSubgrids ();
   ```

```
grid.rebind (adm1->getSubgrid());
// ...
adm1->prepareCommunication (*dof1);
lineq1->attachCommdm (*adm1);
adm2->copy (*adm1);
adm2->prepareCommunication (*dof2);
lineq2->attachCommdm (*adm2);
```

5. *How do I parallelize a nonlinear PDE solver?*
   Carry out all the above steps, and invoke in addition:
   ```
   nlsolver->attachCommAdm (*gp_adm);
   ```
6. *Is it possible to parallelize explicit FE schemes?*
   Yes. Let us look at `$NOR/doc/Book/src/fem/Wave1` for instance (see [6]). The main simulator class `Wave1` does not use `LinEqAdmFE`, but uses Gaussian elimination to solve the *diagonal* mass matrix `M` at each time level. Because the entries of `M` associated with the internal boundary nodes are not correct, we need the following modification of `M` for each subgrid, once and for all:
   ```
   for (i=1; i<=nno; i++) scratch(i) = M(i);
   gp_adm->updateInteriorBoundaryNodes (scratch);
   for (i=1; i<=nno; i++) M(i) = scratch(i);
   ```
   Besides, an explicit call of `GridPartAdm::updateInteriorBoundaryNodes` is also necessary after each matrix-vector product involving the stiffness matrix `K`:
   ```
   prod (scratch, K, u_prev->values());   // scratch = K*u^0
   gp_adm->updateInteriorBoundaryNodes (scratch);
   ```
7. *Are there any functions of* `SaveSimRes` *that can not be used in parallel simulations?*
   The use of `SaveSimRes::lineCurves` and time series plot for a given spatial point should be avoided in parallel Diffpack simulations.
8. *How do I avoid unnecessary output from every processor?*
   Use the following `if`-test before e.g. an output statement to `s_o`:
   ```
   if (proc_manager->master()) s_o << "t=" << tip->time();
   ```
   Recall that `proc_manager` is a global Diffpack handle variable of type `DistrProcManager`.
9. *Can I use the static functions of* `ErrorNorms` *as before?*
   Almost. Just replace all the static `Lnorm` functions belonging to `ErrorNorms` by the corresponding functions of a new class `DistrErrorNorms`, with a reference to a `GridPartAdm` object as an additional input argument. The only exception is the `ErrorNorms::errorField`, which should be used exactly as before. The following is an example
   ```
   ErrorNorms::errorField (*uanal, *u, tip->time(), *error);
   DistrErrorNorms::Lnorm  // fills the error norms
     (*gp_adm, *uanal, *u, tip->time(),
      L1_error, L2_error, Linf_error, error_itg_pt_tp);
   ```
10. *How do I find the global maximum value of a scalar field*?
    The basic idea is to first find the local maximum value and then use the functionality offered by `DistrProcManager` to find the global maximum value.

```
real global_max, local_max = sub_field.values().maxValue();
proc_manager->allReduce(local_max,global_max,PDP_MAX);
```

11. *Is it possible to solve a global linear system in parallel by Gaussian Elimination?*
    No, this is not possible for the current version of the parallel toolbox. Only parallel iterative solvers are supported. As the default answer to the `basic method` item in a `LinEqSolver_prm` sub-menu is `GaussElim`, the user has to specifically write, e.g.,
    ```
    set basic method = ConjGrad
    ```
    in an input file to the Diffpack `MenuSystem`.

## References

1. X. Cai. Overlapping domain decomposition methods. In H. P. Langtangen and A. Tveito, editors, *Advanced Topics in Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*. Springer, 2003.
2. C. Farhat and M. Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *Internat. J. Numer. Meth. Engrg.*, 36:745–764, 1993.
3. Message Passing Interface Forum. MPI: A message-passing interface standard. *Internat. J. Supercomputer Appl.*, 8:159–416, 1994.
4. I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.
5. G. Karypis and V. Kumar. Metis: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, Minneapolis/St. Paul, MN, 1995.
6. H. P. Langtangen. *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming*. Textbooks in Computational Science and Engineering. Springer, 2nd edition, 2003.
7. D.J. Lilja. *Measuring Computer Performance – A Pratitioner's Guide*. Cambridge University Press, 2000.
8. P.S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.
9. G. Radicati and Y. Robert. Parallel conjuget gradient-like algorithms for solving sparse nonsymmetric linear systems on a vector multiprocessor. *Parallel Computing*, 11:223–239, 1989.
10. V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2:315–339, 1990.
11. E.F. Van de Velde. *Concurrent Scientific Computing*. Springer-Verlag, 1994.
12. D. Vanderstraeten and R. Keunings. Optimized partitioning of unstructured finite element meshes. *Internat. J. Numer. Meth. Engrg.*, 38:433–450, 1995.