Two Object-Oriented Approaches to the Parallelization of Diffpack

Xing Cai

Department of Informatics, University of Oslo. P.O. Box 1080, Blindern, N-0316 Oslo, Norway
email: xingca@ifi.uio.no
phone: (+47) 22840068, *fax:* (+47) 22852401
URL: http://www.ifi.uio.no/~xingca/

Abstract

We present two object-oriented approaches to parallelizing Diffpack, which is a large scale software environment for scientific computation. The two approaches are different in that one directly addresses the parallelization of Diffpack linear algebra operations, while the other promotes parallelism at a higher level and naturally incorporates modern domain decomposition methods into parallel computation. In both approaches object-oriented programming techniques are used extensively to ensure code flexibility and extensibility. We also present in this paper simplified coding examples of parallelizing an existing sequential simulator. It is shown that the parallelization approaches are systematic and easy to use. In addition, related CPU-measurements are included for studying parallel efficiency of the resulting parallel simulators.

1 Introduction

The application of object-oriented (OO) programming techniques in developing sequential software for scientific computation has become increasingly popular in recent years. This trend may be attributed to several powerful features of OO programming. Take for instance the programming language C++, its features of modularity, polymorphism and inheritance suit very well for organizing large scale software environments for scientific computation; see e.g. [3]. The direct benefits are, among other things, good code re-usability and extensibility. As for computation performance, recent research (see e.g. [1]) shows that "performance-conscious" usage of the OO features, i.e. restricting object orientation to the high-level administrative code, does not affect the overall efficiency.

However, applying OO programming techniques in developing large scale parallel software libraries for scientific computation is not yet a mainstream practice. One recent forum for the topic found place at the ISCOPE'97 workshop (see [13]). In general, some of the research effort in this field concentrates on inventing new parallel programming languages with OO features, whereas the rest effort is on using existing programming languages. OVERTURE [4] and PETSc [2] are two projects of the latter type. OVERTURE uses C++ and mainly deals with parallel computation on structured grids, while PETSc neatly embeds OO features into the C programming language and is capable of large scale parallel computation on unstructured grids.

The "unpopularity" of using C++ in writing parallel software is largely due to performance consideration, since the procedural programming language FORTRAN 77 is widely accepted to have performance advantages. Nevertheless, restricting OO features to high-level administration while using low-level FORTRAN-like constructs and C-style arrays in computation-intensive codes can reduce the OO overhead to a negligible level. On the other hand, OO programming techniques are actually well-suited for handling the demanding task of parallelizing large scale sequential software libraries, where one has to address complex issues such as message passing instrumentation and parallel performance acquisition. During our recent work of migrating the OO scientific computation environment Diffpack [9, 16] onto multi-processor platforms, we found out that OO programming techniques can be used to greatly ease the parallelization process. More specifically, we have been able to extract all the parallelization related codes into a few small add-on libraries, while keeping the huge sequential libraries almost intact. The result is easy production of portable, extensible and user-friendly Diffpack simulators with high parallel efficiency. The purpose of this paper is therefore to illustrate two such OO parallelization approaches and report some computational results obtained on the HP V2500 system in particular.

The emphasis of Diffpack is on the numerical solution of partial differential equations (PDEs). For such computations, the most CPU-intensive parts are the construction of linear systems of equations and the solution of these systems. Take for instance the finite element method (FEM), carrying out linear system construction in parallel is straightforward, provided that there exists a balanced partition that decomposes the global FEM grid into a collection of subgrids. In fact, construction of the sub-systems, which make up the *virtual* global linear system, can be done entirely independently on individual processors. Parallelization of the solution process is however more demanding, because different processors need to work cooperatively and exchange nodal values between neighboring subgrids. Iterative solution methods, which are used exclusively in solving large scale linear systems, consist mainly of three types of operations: vector addition, inner product between two vectors and matrix-vector product. While vector addition is intrinsically parallel, parallelization of the other two operations can be realized by first invoking local operations restricted to the subgrids and then adding up or updating the sub-results. We note that the main computation in these parallel operations can be done by applying existing sequential routines on the sub-vectors/sub-matrices. This thus brings our first parallelization approach, termed as the linear-algebra-level (LAL) approach, where we create a small add-on library for grid partitioning, neighbor subgrid recognition and communication between subgrids. We mention that OO programming techniques not only help to produce a flexible and extensible grid partition algorithm hierarchy capable of treating different situations, but also enable a seamless coupling between the old sequential linear algebra libraries and the new add-on library.

While the LAL approach follows the common parallelization practice, we have also found another parallelization approach that uses object orientation at an even higher level. The so-called simulator-parallel (SP) approach has its mathematical foundation in overlapping domain decomposition (DD) methods, where the global solution of a PDE is sought by an iterative process in which subproblems are repeatedly solved with updated boundary conditions. The iterative DD solution process is inherently parallel, and an extra gain is that such multilevel methods have extremely good numerical efficiency. More importantly, parallelization in the fashion of DD permits the usage of an existing sequential simulator as an individual unit on each subproblem. Of course, these sequential simulators now need to work under a global administration, and additional communication functionalities are required. To allow users to use the SP approach in a systematic and efficient way, we have created a generic implementation framework into which users can easily adapt existing sequential simulators without having to worry about the details of global administration and inter-processor communication.

The rest of the paper is organized as follows. Section 2 explains the LAL parallelization approach in detail. Afterwards, Section 3 concentrates on the SP approach and the associated generic implementation framework. In Section 4, we apply both the parallelization approaches to the same test problem and study some numerical experiments. A more demanding case of parallelizing multigrid V-cycles on a hierarchy of highly unstructured grids will be demonstrated i Section 5 before we end the paper with some concluding remarks. C++ terminologies will be used in the rest of the paper where applicable.

2 The LAL approach

Frankly, parallel computation was not considered in the initial design of Diffpack at the beginning of the nineties. As the migration of existing sequential Diffpack simulators to multi-processor platforms is inevitable in the near future, there is a strong desire for a pain-free transition. In the following, we will explain why our LAL approach is able to keep the existing large scale Diffpack sequential linear algebra libraries almost intact, by creating a new add-on library containing parallelization related codes. The result is that Diffpack users can easily transform their sequential simulators into parallel ones by only adding a few lines of new code. Most typically, such parallel Diffpack simulators share the *same* codes as their sequential counterparts, and users are able to run the simulators both concurrently and serially.

2.1 Analysis of typical Diffpack computation

The main strength of Diffpack is in the numerical solution of PDEs. Linear systems that arise from discretizing PDEs using methods such as finite difference, finite element and finite volume are almost exclusively of sparse pattern. Only nodes lying close together have non-contribution to each other. Therefore if the original global grid is partitioned into smaller subgrids, nodes in different subgrids normally do not interact, except those lying on or in the vicinity of the internal boundaries. This simplifies the parallelization work somewhat, because our main concern will be parallelizing linear algebra operations involving particularly sparse matrices. In other words, we do not have to write completely general "black-box" parallel linear algebra routines. Instead, we can use routines in ScaLAPACK for operations involving e.g. dense matrices. We mention that this can be done by creating matrix/vector types of ScaLAPACK in Diffpack, as has been done in the sequential Diffpack linear algebra libraries to use some LAPACK routines.

From now on we will concentrate on FE computation since it is the most common type in Diffpack. For any Diffpack FE simulator, the computationally intensive parts are the assembly of systems of linear equations and the solution of them. The assembly process is inherently parallelizable because the calculation of element matrices and vectors can be done *completely independently* of each other. Iterative solution methods, which are used almost exclusively for solving such sparse systems, have mainly three types of operations: vector addition, inner product between two vectors and matrix-vector product. Assume that the global FE grid is partitioned into a number of smaller subgrids. While vector addition is embarrassingly parallel, vector inner product and matrix-vector product can be done in parallel by first invoking operations restricting to the subgrids and then adding up or updating the sub-results. An important observation here is that existing sequential routines can be used directly to obtain the sub-results. The extra work is information exchange between subgrids. Note also that there is no need to physically store global matrices or vectors on any single processor, because these global matrices/vectors can be represented *virtually* by the distributed components on subgrid.

Preconditioning is often needed when linear systems are solved iteratively. Using the above "divide-and-conquer" approach for creating parallel preconditioners may sometimes affect the numerical property of certain preconditioners. For achieving good preconditioning effect and parallelism we refer readers to Section 3 where the SP approach can be used to make parallel DD preconditioners.

2.2 Grid partition

An even distribution of the overall work among processors is essential for achieving good parallel efficiency. For PDEs to be discretized on a global grid, it is possible to first partition the global grid into a number of smaller subgrids and then carry out discretization restricted on those subgrids. For FE computation, each element of the global grid must exist in at least one subgrid, depending on whether the partition is overlapping or non-overlapping. We note that even for a non-overlapping grid partition, neighboring subgrids share nodes on the internal boundaries between them. Consequently, overhead due to "redundant" computation on those internal boundary nodes is inevitable. Mathematically, a non-overlapping grid partition is equivalent to solving a graph problem (see e.g. [11, 15]). The need for overlapping subgrids arises mainly in preconditioning computation, e.g., overlapping DD methods typically require certain amount of overlapping elements between neighboring subgrids.

There are several ways of creating subgrids. One way is to start with a global grid and do the grid partition as mentioned above. The other way is to let each processor independently generate its subgrid when enough geometry information is available. To offer Diffpack users with flexibility in the subgrid construction process, we have thus defined a *pure virtual* base class with name `GridPart`:

```
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 its versions of the member function `makeSubgrids` are required to be implemented in a derived class. We mention that `GridPart_prm` is a simple object containing diverse parameters and `GridFE` represents a standard Diffpack FE grid. As concrete subclasses of `GridPart`, we have already created the following ones:

1. `UnstructuredGridPart` – creates subgrids by partitioning an unstructured global grid. The number of resulting subgrids is arbitrary. Class `UnstructuredGridPart` is in fact a pure virtual class itself. Different implementations of non-overlapping grid partition give rise to different unstructured grid partitioner. An example is subclass `MetisGridPart` derived from `UnstructuredGridPart` where the METIS algorithm [14] is implemented.
2. `UniformGridPart` – creates subgrids by a regular partition of a structured global grid.

3. FileSourceGridPart – reads ready-made subgrids from data files.

It should also be mentioned that Diffpack users can easily extend the GridPart hierarchy by deriving new subclasses where the member function makeSubgrids is implemented.

2.3 An add-on library

The class hierarchy of GridPart is actually the first part of an add-on library containing codes that are parallelization specific. As a second part of the add-on library, we introduce user-friendly routines for inter-processor communication. This is done in form of a new class whose simplified definition is as follows:

```
class GridPartAdm : public SubdCommAdm
{
protected:
  Handle(GridPart_prm) param;
  Handle(GridPart) partitioner;
  bool overlapping_subgrids;
  // ...
public:
  GridPartAdm ();
  virtual ~GridPartAdm ();
  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);
};
```

In addition to offering high-level inter-processor communication routines, class GridPartAdm also administrates the process of grid partition. We can see from above that GridPartAdm has pointers to a GridPart_prm object and a GridPart object. (In Diffpack, Handle(ABC) means a smart pointer to an object of type ABC.) It is inside the GridPartAdm::prepareSubgrids function where an object of GridPart is instantiated to carry out the grid partition. The actual type of the grid partitioner is determined at *run-time* according to user input. The function prepareCommunication is used for communication pattern recognition and set-up of internal data storage for inter-processor messages. It takes as input a Diffpack DegFreeFE object, which is responsible for associating unknowns of the linear system with grid points. Diffpack users have thus access to all the functionalities of the add-on library through GridPartAdm, this is very convenient for developing parallel codes. The member function updateGlobalValues ensures that every node, which lies in more than one subgrid, is updated to have the *same* value in every subgrid, whereas the function updateInteriorBoundaryNodes concerns only nodes that lie exactly on the internal boundaries. Finally, member functions matvec, innerProd and norm carry out respectively the parallel version of matrix-vector product, inner product between two vectors and norm of a vector.

2.4 Code portability

Only *standard* MPI [17] routines have been used in the low-level inter-processor communication. This makes the add-on library *fully portable* across different parallel platforms. These MPI routines are all located inside GridPartAdm and are deliberately hidden from the user. This is done to 1. increase user-friendliness of the codes, 2. enable quick change of the message passing protocol without affecting other parts of the library, 3. allow easy adaption/modification of the low-level communication using special programming directives available on a specific parallel platform. So far, prototypical systems of the parallel Diffpack have been successfully installed on major parallel platforms such as SGI Cray Origin 2000, HP V2500 and IBM SP2. And we can also report an equally smooth installation on a Scali parallel machine [18], which is essentially a cluster of PC processors.

2.5 Coupling with the sequential libraries

One of the main objectives of the LAL approach is to maintain the existing sequential Diffpack linear algebra libraries as intact as possible. To enable a seamless coupling between the sequential libraries and the new add-on library, we have devised the following "dummy" interface class `SubdCommAdm`, which is pure virtual and all its member functions have empty implementation.

```
class SubdCommAdm
{
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);
};
```

In fact, class `SubdCommAdm` is the *only* new class we have to insert into the standard Diffpack linear algebra libraries. Of course, several classes in the standard linear algebra libraries need a slight code extension. Take for instance class `LinEqSystemPrec` that contains, among other things, the system matrix, the solution vector and the right-hand side vector. We put the following two lines in the class definition:

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

In above, the member function `attachCommAdm` can be used to set the smart pointer `comm_adm` to a `SubdCommAdm` object. Consequently, the member function `LinEqSystemPrec::matvec` is modified as follows:

```
void LinEqSystemPrec:: matvec (const LinEqVector& c, LinEqVector& d)
{
  // Amat is a smart pointer to the system matrix
  if (comm_adm.ok())  // does comm_adm point to an SubdCommAdm object?
    comm_adm->matvec (Amat.get(), c, d);  // parallel computation
  else
    Amat->prod (c, d);                  // sequential computation
  matvecCalls++;
}
```

It is obvious that such extensions of the standard Diffpack linear libraries will not cause any change when executing any sequential code, because an object of type `SubdCommAdm` is never allowed and therefore the test

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

will never return a true value. However, when the new add-on library is used, we will be able to create objects of `GridPartAdm`, which is defined as a subclass of `SubdCommAdm` (see Section 2.3). A seamless coupling between the standard Diffpack linear algebra libraries and the new add-on library thus arises.

2.6 A short summary

The LAL approach is very easy to use and only requires the user to insert into the original sequential simulator a few extra calls of parallelization specific routines offered by `GridPartAdm`. Simplified examples will be given in Section 4. In addition, the LAL approach maintains the availability of the rich collection of different components of the Diffpack linear algebra libraries. For example, all the iterative methods for solving linear systems (except the multilevel variants that will be addressed by the SP approach) and the majority of the preconditioners and convergence monitors are available for parallel computation.

3 The SP approach

Domain decomposition (DD) methods, see e.g. [8, 19, 20], are well known to have superior numerical efficiency in solving PDEs. Roughly speaking, these methods search for the solution of the orignal large problem by iteratively solving smaller subproblems, so these methods possess inherent parallelism. The so-called overlapping Schwarz methods are a sub-category of the DD methods and operate on overlapping subdomains. They have a simple algorithmic structure because there is no need to solve special interface problems as is required in non-overlapping DD methods. An important observation about overlapping Schwarz methods is that the subproblems arise from restricting the original PDE(s) onto the subdomains. So every subproblem is a miniature of the original large problem. Moreover, the subproblems can often be solved by the same sequential method that was designed to solve the original large problem. In regard of implementation, this means that an existing sequential simulator can be used as the subdomain solvers, probably after some minor modifications. We therefore proposed in [5] a SP parallelization approach that aims to apply extended/modified sequential simulators as an individual unit in parallel Schwarz methods. It proves that viewing the overlapping Schwarz methods at the level of subdomain simulators, instead of at the level of distributed linear algebra operations, promotes reuse of existing sequential simulators.

OO programming techniques are also advantageous in this parallelization approach not only because they are convenient for minor extension/modification of an existing sequential simulator, but more importantly because they can be used to build up a generic implementation framework that encapsulates the necessary inter-processor communication and global administration. The framework promotes a systematic parallelization approach and thus greatly simplifies user's coding effort. To summarize, the SP approach reads: *Assign one processor with one or several sequential simulators each responsible for one subdomain; The coordination of the computation among the processors is left to a global administrator implemented at a high abstraction level close to the mathematical formulation of DD.* Figure 1 depicts the situation. The reason for allowing multiple subdomains per processor is to increase the flexibility, because partition of the physical domain (e.g. according to its shape) can then be independent of the number of available processors.

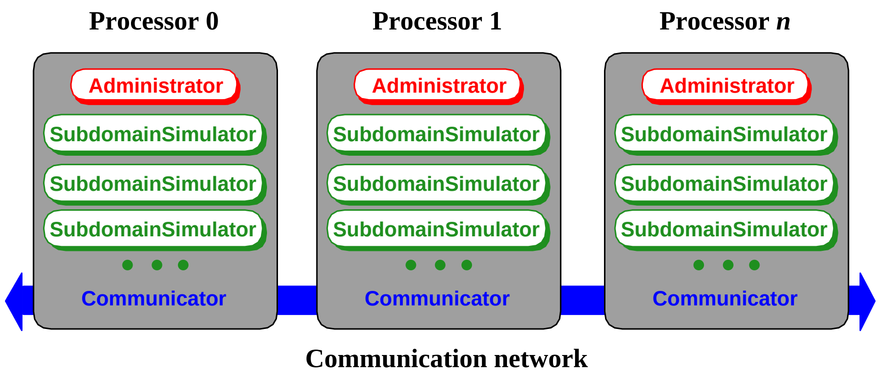

Fig. 1: A generic framework associated with the simulator-parallel parallelization approach.

In the SP approach, we only work with local PDE problems, the data distribution is implied by an overlapping grid partition, so there is no explicit need for global representation of data. Moreover, the local administration of each subdomain simulator allows flexible choice of its own solution method, preconditioner, stopping criterion etc. Most importantly, parallelization at the level of subdomain simulators opens the possibility of reusing existing reliable and optimized sequential simulators. The parallelization approach can essentially take any sequential simulator, which handles arbitrary grid and boundary conditions and which is capable of assembling and solving a linear system $\mathbf{A}_i\mathbf{x}_i = \mathbf{b}_i$. Numerically, the combination of efficient subdomain solvers and the overlapping Schwarz method ensures the overall numerical efficiency of the resulting parallel simulator. The SP approach strongly promotes code reuse, because most of the global administration and the inter-processor communication can be extracted into add-on libraries.

3.1 Design overview

We consider a generic implementation framework consisting of three main parts: the sequential subdomain simulators, a communication part and a global administrator. One class hierarchy with base class `SubdomainSimulator` is built to give a generic representation of *any* sequential subdomain simulator. Different classes in the `SubdomainSimulator` hierarchy are designed to be used in different situations. The communication part makes extensive use of the functionalities offered by `GridPartAdm`. OO programming techniques also inject great flexibility into the design of the global administrator. It allows the user to choose, among other things, whether to use DD as a preconditioner or a stand-alone iterative solver. The part inside the global administrator that makes connection with the subdomain simulators and the communication part can also easily be modified by the user.

3.2 Subdomain simulators

Class `SubdomainSimulator` gives a generic representation of any sequential subdomain simulator in our framework. The local data structure of such a simulator contains, among other things, the subgrid, the local stiffness matrix, the local right-hand side vector and the local solution vector. The basic functionalities include a numerical discretization scheme and an assembly process for building up the local linear system. A linear algebra toolbox is also necessary for `SubdomainSimulator` to control the choice of the local solution method, preconditioner, stopping criterion etc. We have made most of the member functions in `SubdomainSimulator` pure virtual, which need to be overridden in a derived subclass. These member functions constitute a standard interface shared by all the subdomain simulators. It is through this standard interface that the communication part and the global administrator of the implementation framework operate. Adapting an existing sequential simulator is easy, because most of the work consists merely of binding the pure virtual member functions in `SubdomainSimulator` to the concrete member functions in the sequential simulator. The situation is illustrated in Figure 2. As subclasses of `SubdomainSimulator`, we have derived `SubdomainFEMSolver` for simulators solving a scalar/vector elliptic PDE discretized by FEMs, and `SubdomainFDMSolver` for simulators using finite difference discretizations. A subclass `SubdomainMGSolver` of `SubdomainFEMSolver` is also derived for representing simulators that use multigrid (MG) V-cycles (see e.g. [12]) in subdomain solves. In short, the user is provided with a selection of ready-made classes. The user can either use these ready-made classes directly in a specific application or derive from them new subclasses to incorporate new adaptations.

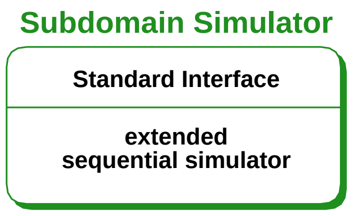

Fig. 2: A generic representation of a sequential subdomain simulator.

3.3 Communication

For parallel FE simulations, the concrete communication between processors is in form of exchanging messages and is handled by objects of type `CommunicatorFEMSP`. Class `CommunicatorFEMSP` is implemented as a subclass of `GridPartAdm` to directly inherit the inter-processor communication routines. On each processor, there is one such object connecting to the local subdomain simulators. Through its smart pointers to the `SubdomainFEMSolver` objects class `CommunicatorFEMSP` can retrieve and modify internal data belonging to the subdomain simulators.

3.4 The global administrator

Overlapping Schwarz methods can either work as stand-alone iterative solution methods or as preconditioners for Krylov subspace methods. The first objective of the global administrator is therefore to offer the above choice at *run-time*. If the user decides to use an overlapping Schwarz method as the preconditioner for a Krylov subspace method, the administrator should also let him/her pick up a particular Krylov method in the standard Diffpack way. We have thus devised a main administrator class `PdeFemAdmSP` whose basic structure is as follows:

```
class PdeFemAdmSP
{
```

```
protected:
  ParaPDESolver_prm psolver_prm;
  Handle(ParaPDESolver) psolver;
  Handle(SPAdmUDC) udc;
// ...
};
```

In the above code segment, `ParaPDESolver_prm` is an object containing many parameters to be chosen by the user at run-time. Among the parameters there are

1. flags indicating whether the overlapping Schwarz method should be used as a preconditioner,
2. number of maximum DD iterations, prescribed accuracy and type of the convergence monitor etc.

The second component of class `PdeFemAdmSP` is `ParaPDESolver` that has two subclasses. The first subclass `BasicDDSolver` represents an overlapping Schwarz method to be used as a stand-alone iterative solver, and the second subclass `KrylovDDSolver` works as a preconditioner. At run-time, when the user has chosen the parameters by e.g. filling items on a user-friendly menu, a concrete parallel PDE solver is created. In other words, `Handle(ParaPDESolver)` will be bound to a concrete object of `ParaPDESolver`:

```
psolver.rebind(psolver_prm.create());
```

The parallel solution of a PDE will be carried out by the virtual member function `ParaPDESolver::solve`. For `BasicDDSolver` the `solve` function has the following implementation:

```
bool BasicDDSolver:: solve ()
{
  // udc is of type SPAdmUDC*
  iteration_counter = 0;
  while (iteration_counter++<max_iterations && !satisfied())
    udc->oneDDIteration (iteration_counter);
  return convflag;
}
```

The last component of `PdeFemAdmSP` is `SPAdmUDC`, which connects the global administrator with the subdomain simulators and the communication part. In a parallel simulation, one `SPAdmUDC` object resides on each processor and has one `CommunicatorFEMSP` object plus one or several `SubdomainFEMSolver` objects under its control. We remark that "UDC" stands for "user-defined-codes" and is used here to indicate that the user has the possibility of making modifications of its member functions. One of the main member functions of `SPAdmUDC` is `oneDDIteration` whose simplified implementation reads:

```
void SPAdmUDC:: oneDDIteration (int iteration_counter)
{
  if (use_coarse_grid)
    coarseGridCorrection ();
  for (int i=1; i<=num_local_subdomains; i++) {
    local_fem_solvers(i)->updateRHS ();
    local_fem_solvers(i)->solveLocal ();
  }
  updateGlobalValues ();
}
```

4 Two test cases

We present in this section two test cases of parallelizing an existing sequential Diffpack simulator, using the two OO parallelization approaches. First, we briefly describe the mathematical model for the test problem. Then we highlight the coding effort required by the two parallelization approaches. Thereafter, we give concrete CPU-measurements of executions on different numbers of processors, thus demonstrating the parallel efficiency of the resulting parallel simulators.

4.1 The test problem

For simplicity, we study the following 2D Poisson equation posed on the unit square $\Omega = (0,1)^2$:

$$-\nabla^2 u \quad = \quad f \quad \text{in } \Omega,$$

where u has known value on $\partial\Omega$. We have chosen a right-hand side f so that $u = -xe^y$ is the analytical solution. Finite element discretization is used to produce the linear system. For the test problem we use 460,800 triangular elements which give rise to a linear system of 231,361 unknowns.

4.2 Coding effort required by the LAL approach

Assume there exists a sequential Diffpack simulator, say class `PoissonSolver`, for solving the Poisson equation. The coding effort required by the LAL approach is merely adding a few lines of new code. So it suffices to enclose the parallelization specific codes by the `#ifdef PARALLEL_CODE` and `#endif` directives, thus maintaining a single code for both the sequential and parallel simulators. Typically, the user needs to add in the definition of class `PoissonSolver` a smart pointer of a `GridPartAdm` object, i.e.,

```
#ifdef PARALLEL_CODE
  Handle(GridPartAdm) grid_part_adm;
#endif
```

Then, there remain two modifications to be done of the original sequential routines. The first modification is to let the `GridPartAdm` object produce the subgrid, i.e.,

```
#ifdef PARALLEL_CODE
  grid_part_adm->scan (menu);
  grid_part_adm->prepareSubgrids ();   // grid generation and partition
  grid.rebind (grid_part_adm->getSubgrid());        // new way
#else
  grid.rebind (new GridFE);
  readOrMakeGrid (grid(),menu.get("global grid"));  // old way
#endif
```

In the above code segment, `menu` is an object of type `MenuSystem`, which allows flexible ways of parameter input, and `grid` is a smart pointer to a Diffpack finite element grid. The second required code modification is to add the following two lines of new code:

```
#ifdef PARALLEL_CODE
  grid_part_adm->prepareCommunication (dof.getRef());
  lineq->attachCommAdm (grid_part_adm.getRef());
#endif
```

The first line invokes `GridPartAdm::prepareCommunication` before any inter-processor communication can take place. The second line attaches the `GridPartAdm` object to `lineq`, which points to a Diffpack linear algebra control object. The linear algebra control object will then invoke all the other `attachCommAdm` functions, including e.g. the one belonging to `LinEqSystemPrec`. In this way, necessary communication and synchronization will be enforced automatically during the later solution process.

Thereafter, the user needs to recompile the modified `PoissonSolver` class using an additional compiler option `-DPARALLEL_CODE`. Finally, a new linkage of all the object files together with the add-on Diffpack library is necessary for the parallelized simulator to run on multiple processors.

4.3 Coding effort required by the SP approach

There are essentially two jobs that need to be done in the SP parallelization approach: 1. Extend the existing sequential simulator to fit into the implementation framework; 2. Make a connection between the extended sequential simulator and the global administrator. More specifically, the user needs to derive a new simulator class as a subclass of *both* the existing sequential simulator *and* (a subclass of) the generic `SubdomainSimulator`. In our case, the following new simlator class `SubdomainPESolver` will be created:

```
class SubdomainPESolver : public SubdomainFEMSolver,
                          public PoissonSolver
{
protected:
  virtual void initSolField (MenuSystem& menu);
  virtual void createLocalMatrix ();
public:
  SubdomainPESolver () {}
  virtual ~SubdomainPESolver () {}
  virtual void solveLocal ();
};
```

The new class mainly binds the virtual member functions of class `SubdomainFEMSolver` to the specific functions in class `PoissonSolver`. The virtual member function `createLocalMatrix` will call the corresponding member function of `PoissonSolver` to create the local linear system. And the `solveLocal` function typically invokes `PoissonSolver::lineq->solve()`. Finally, the virtual member function `initSolField` is used to set its internal pointers to the concrete local data, so its simplified implementation looks like:

```
void SubdomainPESolver:: initSolField (MenuSystem& menu)
{
  // ...
  subd_lineq.rebind (PoissonSolver::lineq.getRef());
  subd_dof.rebind (PoissonSolver::dof.getRef());
  global_solution_vec.rebind (PoissonSolver::solution.getRef());
}
```

For the second job, a new subclass of `SPAdmUDC` needs to be derived as follows:

```
class PESolverSP: public SPAdmUDC
{
protected:
  virtual void createLocalSolver (Handle(SubdomainFEMSolver)& local_solver)
  {
    SubdomainPESolver* solver = new SubdomainPESolver;
    local_solver.rebind (solver);
  }
// ...
};
```

The main purpose of `PESolverSP` is, as explained before, to attach concrete `SubdomainPESolver` objects to the global administrator. The connection is made in the virtual member function `createLocalSolver`, which sets an internal pointer to every concrete `SubdomainPESolver` object.

When the two new classes are ready, the main parallel program can be written straightforwardly as:

```
int main (int nargs, const char** args)
{
  initDiffpack (nargs,args);
  // ...
  // menu is a given Diffpack input source
  PESolverSP  udc; udc.define (*menu, MAIN);
  PdeFemAdmSP adm; adm.define (*menu, MAIN);
  menu->prompt();
  udc.scan (*menu); adm.scan (*menu);
  adm.attachSPAdmUDC (&udc);
  adm.init ();
  adm.solve ();
}
```

4.4 Some CPU-measurements

The CPU-measurements listed in Tables 1-2 are obtained on an HP V2500 system with 16 PA8500 440 MHz processors. The system has a 8×8 nonblocking switch interconnection and a 1.9 GB/s memory channel bandwidth. The C++ compiler used is 64-bit `aCC` with the following compiler options

```
-O -DHP_ACC_Cplusplus +DA2.0W -DREGEX_MALLOC=1
```

The HP MPI library is of version 1.03. The CPU-times are measured using the standard `MPI_Wtime` command.

As being explained before, building the virtual global linear system is inherently parallel, because there is no need for communication while subdomains construct their own sub-systems. Therefore we only measure the CPU-times spent on the solution process, as a function of number of processors P. The conjugate gradient (CG) method is used to solve the virtual global linear system, where we regard that convergence is reached if the global residual is reduced by a factor of 10^5. For the parallel simulator resulting from the LAL approach, no preconditioner is used so that the needed number of CG iterations is independent of P. We use METIS for grid partition when P is given as input at run-time. For the parallel simulator resulting from the SP approach, we use a *fixed* grid partition with 32 overlapping subgrids. For example, when 4 processors are in use, each processor is responsible for 8 subdomains. One DD iteration is used as the preconditioner for the CG method. Keeping a fixed grid partition, while allowing P to change, we ensure that the total computational work amount, obtained by adding up work on all the P processors, remains constant. To give the reader an idea of the actual situation of load imbalance, we also list $\max N_i$ and $\min N_i$ for the LAL approach, where N_i denotes the number of grid points on subgrid number i.

P	CPU	Speedup	$\max N_i$	$\min N_i$
1	1025.67	~	231,361	231,361
2	492.92	2.08	115,934	115,929
3	371.22	2.77	77,480	77,448
4	254.30	4.03	58,125	58,104
6	172.67	5.94	38,890	38,836
8	124.29	8.25	29,212	29,160
10	96.49	10.63	23,433	23,353
12	79.19	12.95	19,562	19,481
16	58.00	17.68	14,691	14,642

Tab. 1: CPU-times of the CG solution process used by the parallel simulator produced by the LAL approach.

P	CPU	Speedup
1	55.60	~
2	27.89	1.99
4	14.24	3.90
8	7.06	7.88
16	3.61	15.40

Tab. 2: CPU-times of the CG solution process used by the parallel simulator produced by the SP approach.

Comments. The extraordinary super-linear speedup result in Table 1 should be attributed to the effect of cache usage. As P increases, the subgrids become smaller and therefore fit better into the local cache. The communication overhead thus becomes invisible. For Table 2, however, there is no cache advantage for $P = 16$ over $P = 1$, because all the subgrids have a *fixed* size and fit comfortably into the cache. In the extreme case of $P = 1$, for instance, a single processor solves in each DD iteration 32 subproblems one by one, without having to solve one large global problem. The reader should also note that Table 2 demonstrates the superior numerical efficiency of the DD preconditioner, which only needs 12 CG iterations to achieve convergence. Whereas for Table 1, the CG method needs 956 iterations because no preconditioner is in use.

5 One example of parallel multigrid

We consider a 2D solution domain Ω as shown in Figure 3, where we solve a pressure equation

$$-\nabla \cdot (K \nabla u) = f.$$

In the above equation, coefficient K is a piecewise constant function and has value 0.1 inside the low permeable zone in the middle and 1.0 elsewhere. The right hand side f consists of three exponential bell shape functions and we have $\int_\Omega f = 0$. For boundary conditions we have $\partial u/\partial n = 0$ on the entire $\partial\Omega$, except that we enforce $u = 0$ at the lower left corner of Ω to obtain a unique solution.

For such a problem, adaptive local grid refinement (see e.g. [10]) is a useful technique to increase grid resolution in places where necessary. After several such grid refinements, a hierarchy of nested unstructured grids arises. MG methods suit very well for these grid hierarchies (see [6]). Our SP parallelization can also easily be used to parallelize such methods. We will not go into coding details here, because the parallelization is similar to that discussed in Section 4.3. The difference is that we replace `SubdomainFEMSolver` and `SPAdmUDC` with `SubdomainMGSolver` and `SPAdmHierUDC`, respectively. We mention that class `SPAdmHierUDC` is derived from `SPAdmUDC` for handling such parallel MG cycles.

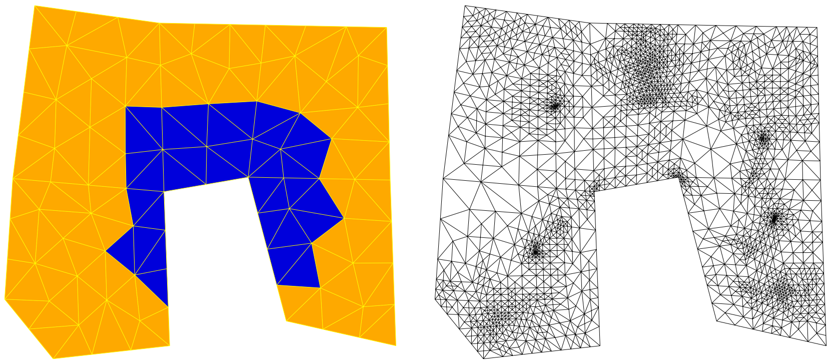

Fig. 3: The solution domain of a pressure equation. The left figure depicts the coarsest grid, the right figure shows the grid after 8 adaptive local grid refinements.

As a test case we have the coarsest grid with 138 triangular elements and run 16 grid refinements, so that the finest global grid has 2,082,625 nodes. For the resulting linear system of equations, we apply a parallel MG V-cycle as preconditioner for a stabilized bi-conjugate-gradient (BiCGStab) method. Convergence is declared when the global residual in its discrete L_2-norm is smaller than 5×10^{-7}. Some CPU-measurements of the solution process are listed in Table 3.

P	CPU	Speedup	max N_i	min N_i
1	352.53	$\sim$	2,082,625	2,082,625
2	180.06	1.96	1,053,505	1,047,265
3	121.28	2.91	705,313	700,497
4	94.02	3.75	532,417	529,705
6	66.86	5.27	360,825	351,393
8	52.09	6.77	273,722	269,865
10	44.38	7.94	222,193	217,305
16	32.13	10.97	141,418	139,793

Tab. 3: CPU-times of the BiCGStab solution process used by a parallel simulator arising from the SP approach. The BiCGStab method is preconditioned by one parallel multigrid V-cycle.

Comments. The relatively poor speedup result shown in Table 3 can be attributed to at least two factors. 1. Load imbalance is quite severe for this test problem on unstructured grids. 2. The size of the inter-processor messages increases as the result of overlapping subgrids. In addition, it is suspected that the problem size, with a total number of 2,082,625 unknowns on the finest global grid, poses a challenge to the HP V2500 system in use. We mention that near perfect speedup for exactly the same problem was obtained on a larger SGI Cray Origin 2000 system (see [6]).

6 Concluding remarks

The two OO parallelization approaches work at different levels. The LAL approach aims to provide Diffpack users with parallel matrix-vector operations that are enabled by the communication functionalities embedded in an add-on library. The OO add-on library can be seamlessly coupled with sequential Diffpack libraries. And the coding effort required by the LAL approach is minimal. The SP approach helps Diffpack users to implement parallel multilevel methods, which have extremely high numerical efficiency for solving PDEs. The associated generic implementation framework is user-friendly and promotes a systematic coding process.

Although examples included in this paper only address the parallel solution of a single PDE in 2D, it should be emphasized that both the OO parallelization approaches are readily applicable to 3D problems and systems of PDEs. We refer to [5, 6, 7] for more such parallel applications. Figure 4 illustrates two *non-matching* subgrids that can be handled by a parallel simulator produced by the SP approach.

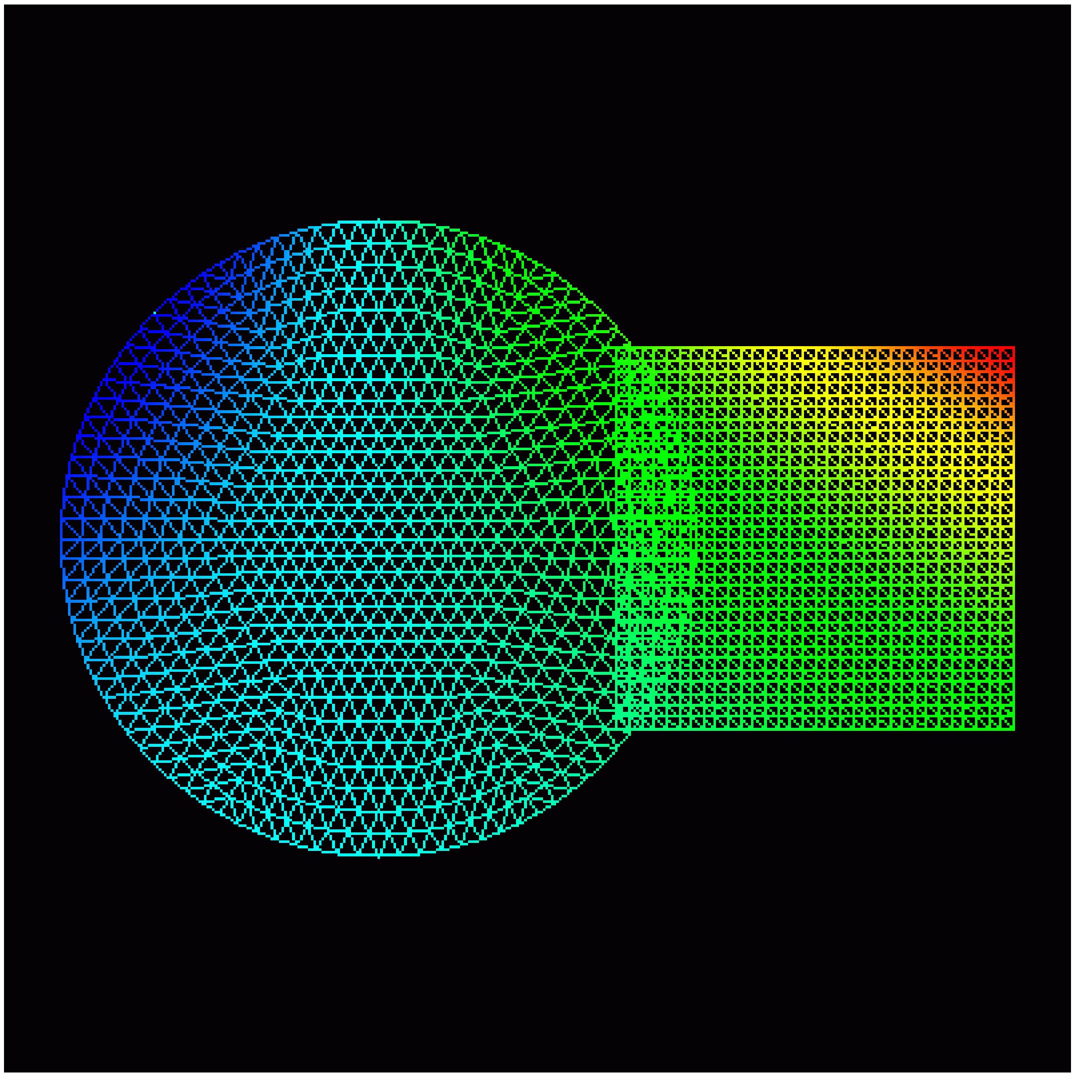

Fig. 4: Two non-matching subgrids.

As for future work, the most important task will be to further reduce the overhead in parallel computation. More specifically, effort is needed to find better grid partitioning algorithms in respect of load balance. This applies especially to cases where overlap between neighboring subgrids is desired. To minimize the communication overhead, it is probably advantageous to overlap computation and communication in e.g. matrix-vector products. This, however, requires modifications of the existing Diffpack matrix and vector data types. Finally, to achieve highest possible overall performance of any parallel simulator, it is also necessary to optimize the sequential operations within individual subdomains. In this connection, we may consider incorporating "alien" mathematical subroutines offered by e.g. MLIB.

Acknowledgements. The work is partially supported by the research Council of Norway (NFR) under Grant 110673/420 (*Numerical Computations in Applied Mathematics*) and through a grant of computing time (*Programme for Supercomputing*). The author is very grateful for the help received from the staff at the HPC project at the University of Tromsø. The author also thanks Dr. Are Magnus Bruaset, Dr. Hans Petter Langtangen and Prof. Aslak Tveito for numerous valuable discussions.

References

1. E. Arge, A. M. Bruaset, P. B. Calvin, J. F. Kanney, H. P. Langtangen, C. T. Miller, *On the numerical efficiency of C++ in scientific computing.* In M. Dæhlen and A. Tveito (eds): Numerical Methods and Software Tools in

Industrial Mathematics, Birkhäuser, Boston, 1997, pp. 91–118.

2. S. Balay, W. D. Gropp, L. C. McInnes, B. F. Smith, *Efficient management of parallelism in object-oriented numerical software libraries.* In E. Arge et al. (eds): Modern Software Tools for Scientific Computing, Birkhäuser, Boston, 1997, pp. 163–202.
3. J. J. Barton, L. R. Nackman, *Scientific and Engineering C++. An Introduction with Advanced Techniques and Examples.* Addison-Wesley, 1994.
4. D. L. Brown, W. D. Henshaw, D. J. Quinlan, *Overture: an object-oriented framework for solving partial differential equations.* In Y. Ishikawa et al. (eds): Scientific Computing in Object-Oriented Parallel Environment, Springer-Verlag Lecture Notes in Computer Science 1343, 1997, pp. 177–184.
5. A. M. Bruaset, X. Cai, H. P. Langtangen, A. Tveito, *Numerical solution of PDEs on parallel computers utilizing sequential simulators.* In Y. Ishikawa et al. (eds): Scientific Computing in Object-Oriented Parallel Environment, Springer-Verlag Lecture Notes in Computer Science 1343, 1997, pp. 161–168.
6. X. Cai, K. Samuelsson, *Parallel multilevel methods with adaptivity on unstructured grids.* Preprint 1998-5, Department of Informatics, University of Oslo.
7. X. Cai, *Numerical Simulation of 3D Fully Nonlinear Water Waves on Parallel Computers.* In B. Kågström et al. (eds): Applied Parallel Computing, PARA'98, Springer-Verlag Lecture Notes in Computer Science 1541, 1998, pp. 48–55.
8. T. F. Chan, T. P. Mathew, *Domain decomposition algorithms.* Acta Numerica (1994), pp. 61–143.
9. Diffpack Home Page, *http://www.nobjects.com.*
10. K. Eriksson, D. Estep, P. Hansbo, C. Johnson, *Introduction to adaptive methods for differential equations.* Acta Numerica (1995), pp. 105–158.
11. C. Farhat, M. Lesoinne, *Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics.* Internat. J. Numer. Meth. Engrg. 36 (1993), pp. 745–764.
12. W. Hackbusch, *Multi-Grid Methods and Applications.* Springer, Berlin, Heidelberg, 1985.
13. Y. Ishikawa, R. R. Oldehoeft, J. V.W. Reynders, M. Tholburn (Eds.) *Scientific Computing in Object-Oriented Parallel Environment.* Springer-Verlag Lecture Notes in Computer Science 1343, 1997.
14. G. Karypis, V. Kumar, *METIS: Unstructured graph partitioning and sparse matrix ordering system.* Department of Computer Science, University of Minnesota, Minneapolis/St. Paul, MN, 1995.
15. B. W. Kernighan, S. Lin, *An efficient heuristic procedure for partitioning graphs.* The Bell System Technical Journal 49 (1970), pp. 291–308.
16. H. P. Langtangen, *Computational Partial Differential Equations – Numerical Methods and Diffpack Programming.* Springer-Verlag, 1999.
17. Message Passing Interface Forum, *MPI: A message-passing interface standard.* Internat. J. Supercomputer Appl. 8 (1994), pp. 159–416.
18. Scali Home Page, *http://www.scali.com.*
19. B. F. Smith, P. E. Bjørstad, W. D. Gropp, *Domain Decomposition, Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, 1996.
20. J. Xu, *Iterative methods by space decomposition and subspace correction.* SIAM Review 34 (1992), pp. 581–613.