SQU Journal for Science, 17 (1) (2012) 147-156

Solving the Flood Propagation Problem with Newton Algorithm on Parallel Systems

Chefi Triki

Department of Mechanical and Industrial Engineering, Sultan Qaboos University, Muscat, Oman, Email:chefi@squ.edu.om.

ABSTRACT: In this paper we propose a parallel implementation for the flood propagation method Flo2DH. The model is built on a finite element spatial approximation combined with a Newton algorithm that uses a direct LU linear solver. The parallel implementation has been developed by using the standard MPI protocol and has been tested on a set of real world problems.

KEYWORDS: Newton equations, Direct solver, MPI, Load balancing.

حل مشكلة انتشار الفيضانات باستخدام خوارزمية نيوتن على أجهزة الحواسيب المتوازية

شافي التريكي

ملخص: تقترح هذه الورقة طريقة لحل مشكلة انتشار الفيضانات باستخدام برنامج Flo2DH على أجهزة الحواسيب المتوازية. وقد تم بناء نموذج الحل بالاستناد إلى تقريب العناصر المحدودة مع خوارزمية نيوتن التي تستخدم التحليل LU المباشر. كما تم وضع الحل على أجهزة الحواسيب المتوازية باستخدام بروتوكول MPI وجرى اختباره على مجموعة من المسائل التطبيقية.

1. Introduction

The solution of most of the mathematical models arising in several contexts of hydraulics and hydrology still remains a big challenge. The choice of the most appropriate method for the solution of the problem under examination is often subject to a trade-off between numerical stability and computational complexity. Moreover, the time execution is often too long to be acceptable for the requirements of the decision makers. This paper has the intention of contributing in this direction for a particular problem: the flood propagation modeling. The aim is to provide a parallel implementation of the well known software Flo2DH, a code built on a finite element spatial approximation combined with a Newton algorithm to solve the linear systems. More specifically, we focus our attention on the solution of the linear system arising in every iteration of the Newton method. We discuss the advantages of direct methods (vs. iterative methods) and we propose a parallel implementation based on an LU decomposition approach.

The parallel solution of direct methods has been the subject of intensive research activity to solve many problems arising in a wide range of contexts (Scott, 2001; Scott, 2002; Amestroy *et al*., 1998; Mallya *et al*., 1997). Moreover, several existing approaches have been proposed for the parallel LU decomposition and for the parallelization of Flood modeling systems (Hluchy *et al*., 2002, 2006). However, to the best of our knowledge, this paper represents a first contribution for the parallel solution of the LU decomposition within the Flo2DH code to solve the flood propagation problem. The only work dealing with this problem which we are aware of, has implemented an iterative method (a stabilized version of the conjugate gradient algorithm) that allows only an approximate solution to be obtained and not the exact one (Abdeouahed *et al*., 2000). Our parallel direct approach is, thus, useful, not only when a high numerical precision is required but also when it can serve as a benchmarker for the development of iterative methods within a flood propagation solver.

The paper is organized as follows: in Section 2 we define the application context of our contribution. In Section 3 we present our parallel approach and discuss the different issues related to its implementation. In Section 4 we report our experimental results, and in Section 5 we give some concluding remarks.

2. Computational fluid mechanics

Computational fluid mechanics is a tool that helps to solve a wide range of problems in fluid mechanics and heat transfer. We usually categorize common fluids as either gas or liquid. A fluid is any material medium that can not support shearing or stresses and remain at rest, from the macroscopic standpoint. The most important properties of fluids are density, viscosity, surface tension, cavitation and boiling.

Fluid flow plays an important role in practice. A detailed description of the flow is usually necessary for better understanding and ultimately solving a number of industrial and scientific problems. Each kind of flow has its proper mathematical description. By mixing together different kinds of flows, the mathematical description becomes more complex. This description uses sets of partial differential equations which in most of the cases have no analytical solution. For this reason numerical methods are used to solve this kind of problems.

In computational fluid mechanics, the flow region or calculation domain is usually divided into a large number of finite volumes or cells. The governing partial differential equations are discretized by using a finite difference, a finite volume or a finite element technique. An example of a small 2-dimensional (2D) finite element (FE) mesh is depicted in Figure 1. Such a discretization generates a set of algebraic equations (corresponding to the respective partial differential equations) which are solved by numerical procedures. The resulting solution thus represents the values of the dependent variables at discrete locations, and the intermediate values are obtained by interpolation.

It is worthwhile noting that most of the problems arising in computational fluid mechanics are characterized by a large number of equations and, consequently, the solution process may take several hours. In order to decrease the execution time it is important to take advantage of parallel computing facilities to run, at least, the most time consuming parts of the simulation.

In this paper we focus on the solution of the flood propagation problem by using the public domain Flo2DH simulator for which we propose a parallel implementation.

3. Parallel implementation

Before presenting our parallel implementation, we need to describe with some details the characteristics of the simulator Flo2DH.

3.1 Flo2DH simulator

Flo2DH is part of the United States Federal Highway Administration's Finite Element Surface-Water Modeling System (FESWMS). It is a computer program that simulates the movement of water and noncohesive sediment in rivers, estuaries, and coastal waters (Froehlich, 2002).

The sequential version of the Flo2DH simulator uses the finite element method for the differential

equations computation (Froehlich, 2002; Dabaghi *et al.*, 2004). The solution space (water surface) is divided into a number of elements depending on the expected precision (see Figure 1). More elements in the computational model means higher resolution and precision, but also more computational load.

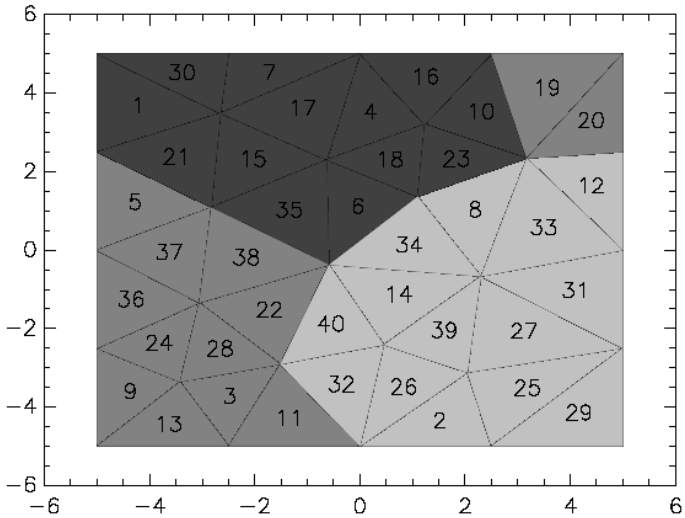

Figure 1. carre test problem - A small 2D FE mesh (partitioned into 3 sets).

The Flo2DH simulator is written in Fortran 77 and Fortran 90 programming languages. The source code, around 2000 pages long, contains a lot of physical and mathematical computations that are not only very complex, but also characterized by a high degree of dependence. (Indeed, our tentative of assigning the task of automatic parallelization of the whole code to a parallel compiler has given very poor results). Therefore parallelizing the whole code seems to be laborious and not promising. For this reason we decided to focus our attention on the most time consuming parts of the code with the aim of decreasing the total execution time of the simulator.

Even though intuition led us to think that the solution of the linear set of equations at each step of the Newton method is the most time consuming part of the code, we decided to carry out a time measurement experiment in order to confirm our intuition. The experiment also had the objective of discovering the percentage of the total time which this consumes and the theoretical speedups that may be expected.

On the basis of our experiment, carried out on a set of 5 test problems with different sizes and by using one processor of the SGI Origin 300 supercomputer, a subroutine called xFront turned out to be the most computationally extensive fraction of Flo2DH. The results reported in Table 1 show, indeed, that the code spends up to 80% of the total time in executing subroutine xFront, whereas no one of the other subroutines of Flo2DH takes a significant amount of time.

Table 1. Execution times (seconds) and percentages

Problem	carre	madora	etape1	solstat	etape2
number of nodes	97	437	6413	4223	13876
xFront	0.46	1.55	7.85	34.86	205.6
total time	0.58	1.85	15.8	41.94	255.9
xFront/total time	0.79	0.84	0.50	0.83	0.80

By analyzing the source code we observe that:

- subroutine xFront is responsible of generating the system of linear equations for each element and then for calling subroutine xAssemble;
- xAssemble is responsible for forming the matrix **A** from the values of each element and for solving the system of equations (Froehlich, 2002):

$$\mathbf{Ax} = \mathbf{b}, \tag{1}$$

where **A** is a large sparse matrix, **b** is a vector, and **x** is the variables vector.

Forming and solving the system of linear equations $\mathbf{Ax} = \mathbf{b}$ is, thus, computationally the most intensive part of Flo2DH. Parallelizing the subroutine xFront represents an opportunity to solve the flood propagation problem faster.

According to Amdahl's law and the time measurements reported in Table 1 it is possible to estimate the maximum theoretical speedup expected from running subroutine xFront on an 8-processor machine (the case of test problem madora that has a fraction of 16% of non parallelizable code):

$$S = \frac{1}{0.16 + \frac{1-0.16}{8}} \doteq 3.77 . \tag{2}$$

However, besides calling xAssemble, xFront calls also a set of subroutines which are responsible for hydrodynamics computations and that are not suitable to be parallelized because of the data dependence. This fact, together with the unavoidable communication overhead, should further worsen the above theoretical speedups of our parallel implementation.

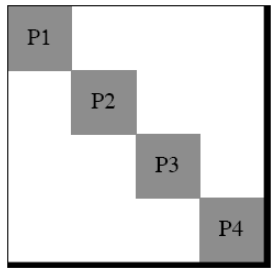

Figure 2. Sparse matrix in DBBD form with processes assignment.

3.2 Parallel frontal solver

In the sequential version of Flo2DH, the frontal method is used to solve the set of equations $\mathbf{Ax} = \mathbf{b}$. This method makes an LU decomposition of the matrix **A** followed by a forward and a backward substitution. An attractive feature of this method is that the workload could be separable into small tasks, and it is not necessary that each task has the whole matrix **A** available in order to make the LU decomposition. Computations are just done over a small part of the whole matrix **A**. This part, called a front, is formed by the summation of the equations for each element. The variables which are fully summed are then decomposed and eliminated from the front (Froehlich, 2002). An efficient implementation should ensure that the size of the front matrix must be very small compared with matrix **A** in order to save memory.

The next step consists in choosing a parallelization approach for solving the $\mathbf{Ax} = \mathbf{b}$ system. There are two main approaches: direct and iterative algorithms. Even though direct methods need a considerable number of floating point operations (high load), they are characterized by a high stability of the solution. For this reason we decided to implement a direct method, giving, in this way, particular attention to the solution's precision at the cost of a possible slight loss in efficiency.

The parallel frontal method works in a similar way to the sequential frontal method. The whole load is divided into a number of tasks that are assigned to the parallel processes. Each process has its own front and performs its computation over a proper part of the matrix **A**. The algorithm can be particularly suitable for parallelization whenever: first, the matrix **A** has the following doubly bordered block diagonal (DBBD) form:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A_1} & & & & \mathbf{C_1} \\ & \mathbf{A_2} & & & \mathbf{C_2} \\ & & \ddots & & \vdots \\ & & & \mathbf{A_n} & \mathbf{C_n} \\ \tilde{\mathbf{C}}_1 & \tilde{\mathbf{C}}_2 & \cdots & \tilde{\mathbf{C}}_n & \mathbf{E} \end{bmatrix} \tag{3}$$

where submatrices $\mathbf{A_i}$ are $n_i \times n_i$, $\mathbf{C_i}$ are $n_i \times k$ and $\tilde{\mathbf{C}}_\mathbf{i}$ are $k \times n_i$, and second, $k \ll n_i$ (i.e. the black part of the matrix in Figure 2 is as small as possible).

In our case it is possible to decompose matrix **A** into the above form by using the partitioning and resequencing algorithms that will be described in the sequel of this section. Thus, the partial frontal decomposition is performed on each of the submatrices:

$$\mathbf{A^i} = \begin{bmatrix} \mathbf{A_i} & \mathbf{C_i} \\ \tilde{\mathbf{C}}_\mathbf{i} & \mathbf{E_i} \end{bmatrix}. \tag{4}$$

Parallel frontal algorithm:

1. Factorization
   - perform a partial LU decomposition of each $\mathbf{A^i}$ (**in parallel**)
   - form the interface problem by summing the Schur complement matrices remaining after performing all the possible eliminations on the submatrices
   - factorize the interface matrix
2. Forward elimination
   - perform a forward elimination on each submatrix (**in parallel**)
   - send data to the interface problem
   - perform a forward elimination on the interface matrix
3. Back substitution
   - perform a back substitution on the interface matrix
   - send data to all the processors
   - perform a back substitution on the submatrices (**in parallel**)

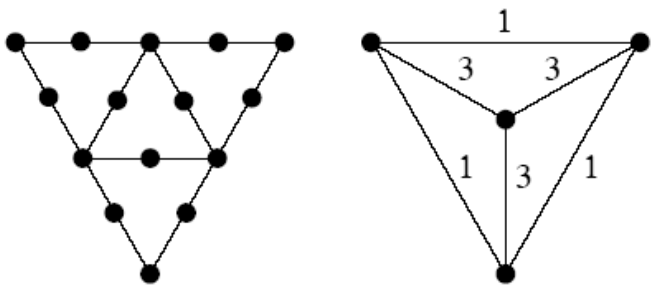

Figure 3. A 4-element problem (left) and its graphical representation (right).

The master-slave paradigm has been chosen to implement the parallel frontal method and the library MPI has been used a message-passing protocol. Besides assigning the tasks to be performed by each process, the master manages also the communication (signals and data) with the slaves. No inter-slave communication happens in our implementation. This parallel frontal method is particularly suitable for a coarse-grained parallelism. As the number of processes increases the communication becomes very time consuming and the algorithm's performance decreases. Moreover, in any parallel implementation, it is necessary to take care of some important issues such as load balancing, resequencing, and code optimization. This will be the subject of the remaining part of this section.

3.3 Load balancing

In order to obtain an efficient load balance it is important, in our case, that matrix (3) has all the matrices $\mathbf{A_1}, \cdots, \mathbf{A_n}$ of equal size and, furthermore, has $k \ll n_i$, a characteristic that strongly depends on the mesh topology.

The load balancing problem can be transformed into a graph partitioning problem that is defined as follows: each vertex represents the computation to be performed over a data unit and each edge indicates the dependence between data units. In the case of the frontal method a data unit corresponds to an element and the dependence (edge) between elements corresponds to the eventual sharing of at least one node. The weight of an edge is defined as the number of nodes that are shared by two elements, as shown in Figure 3, whereas all the vertices are assigned the weight 1.

The graph partitioning problem consists in dividing the graph into disjunctive sets. Each vertex can belong to only one set and the following conditions have to be satisfied:

- the sums of vertex weights for all the sets are of (almost) equal value;
- the sum of edges between the sets has to be as small as possible.

The graph partitioning problem is known to be NP-hard (Wikipedia). Thus, for its solution, it is necessary to use heuristic approaches since exact methods can not be satisfactory. In our implementation we have used the graph partitioning software Chaco (Hendrickson and Lelan, 1994). This software package provides several graph partitioning heuristics and has the advantage of being extremely configurable. On the basis of an empirical consideration we have chosen to implement, in our parallel code, the spectral partitioning method combined with the Kerningham-Lin refinement approach (Rotta, 2008). The solution quality of the graph partitioning heuristic can be seen, for example, in Figure 1 for problem carre and in Figure 4 for problem ourika (the colors refer to the elements to be assigned to the processes).

3.4 Element resequencing

In frontal methods the order in which the elements are processed is very important in order to keep the front as small as possible. As a result, once the graph partitioning has been done, it is necessary to resequence the elements to define the order in which the vertices should be processed. Since the graph resequencing problem is NP-hard (George and Liu, 1981) we have implemented two different heuristics for its solution, the first based on the Reverse Ordering Algorithm of Cuthill and McKee (George and Liu, 1981) and the second, a greedy approach.

Reverse Cuthill-McKee ordering algorithm:

The Cuthill-McKee ordering algorithm divides the nodes into level sets. A level structure rooted at a node r is defined as the partitioning of V into levels $l_1(r), l_2(r), \cdots, l_h(r)$ such that

1. $l_1(r) = \{r\}$ and
2. for $i > 1$, $l_i(r)$ is the set of all nodes that are adjacent to nodes in $l_{i-1}(r)$ but are not in $l_1(r), l_2(r), \cdots, l_{i-1}(r)$.

The Cuthill-McKee algorithm orders, within each level set $l_i(r)$, the first nodes that are neighbours of the first

node in $l_{i-1}(r)$, then those that are neighbours of the second node in $l_{i-1}(r)$, and so on. The Reverse Cuthill-McKee algorithm reverses the order found by Cuthill-McKee.

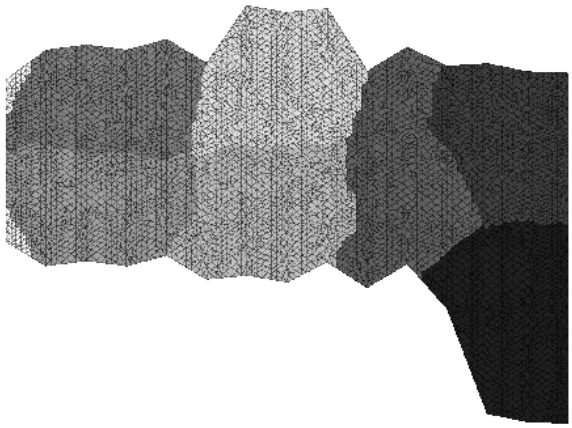

Figure 4. ourika test problem partitioned into 7 sets.

In this work we have also implemented a heuristic based on a greedy approach. In this method the ordering is constructed by taking into account the specific features of the problem under examination so as to minimize some objective quantity at each step of the iterative process. In terms of graph resequencing, the basic ordering process used by our greedy algorithm can be summarized as follows:

Greedy ordering algorithm:

1. Initialization: Construct the undirected graph G^0 corresponding to the mesh;
2. Generic iteration: For $k = 1, 2, \cdots$ until $G_k = \phi$:

   - Choose a vertex v^k from G^k with the minimal number of firstly occurred nodes. If there are several such vertices choose the one with the maximum number of lastly occurred nodes.

   - Eliminate v^k from G^k to form G^{k+1}.

The resulting ordering is the sequence of vertices $\{v_1, v_2, \cdots\}$.

3.5 Performance optimization

Once a functional parallel code has been developed it is necessary to optimize its performance. For this purpose we have used two different techniques:

MPE Library: helps in finding the parts of the parallel program that are suitable for being optimized so that their execution time is reduced. The MPE software package performs this task by using a performance visualization approach.

Complier switch: this task is both architecture and compiler dependent.

3.6 Mesh visualization

Our parallel code has been enriched by certain tools that help both the developer and the user in controlling the execution process and checking the solution quality. Among the tools that are particularly useful for the solution of large scale problems we mention the visualization program plplot library (Lebrun and Furnish, 1994), which has been used to depict the meshes of our test problems in this paper.

4. Numerical results

The validation of our parallel code has been done by using a set of test problems that refer to real world applications. The data concern a set of rivers located in Morocco, Algeria, and Lebanon (Dabaghi *et al*., 2004). The architecture platform used is the SGI Origin 300 running the operating system IRIX 6.5 using the MIPSpro (Fortran and C) compilers. The CPU time has been measured on the master program by using the dtime function.

Our experimental results have been collected in Table 2. We report, for each test problem, the computational time (T) of both subroutine xFront and the whole Flo2DH simulator when executed sequentially, on 4 processors (1 master and 3 slaves), and on 8 processors (1 master and 7 slaves). The corresponding speedups (S) are reported in Table 2 and also depicted (for the case of 8 processors) in Figure 5 (the test problems are identified by their number of nodes). The collected results show that (i) the best speedups are obtained for test problem etape2 which is characterized, according to Table 1, by a low fraction of code that is not parallelizable, (ii) the worst speedup is obtained for test problem etape1 which is characterized by a high fraction of code that is not parallelizable and finaly (iii) the speedup values increase as the number of processes increases for most of the test problems with the exception of carre and madora. This is due to the fact that these two problems are very small, so that the parallelized load is not able to compensate the communication overhead.

Table 2. Execution times (seconds) and speedups

Problem & # of nodes		**sequential**	**4 processes**		**8 processes**	
		T	**T**	**S**	**T**	**S**
carre	xFront	0.46	0.12	3.83	0.15	3.06
97	FES2DH	0.58	0.22	2.63	0.25	2.32
madora	xFront	1.55	0.90	1.72	0.95	1.63
437	FES2DH	1.85	1.29	1.43	1.34	1.38
etape1	xFront	7.85	5.91	1.32	3.58	2.19
6413	FES2DH	15.8	14.65	1.07	12.27	1.28
solstat	xFront	34.86	22.41	1.55	13.09	2.66
4223	FES2DH	41.94	30.23	1.38	20.92	2.00
ourika	xFront	161.92	125.84	1.29	74.01	2.19
22292	FES2DH	181.41	133.47	2.19	81.67	2.22
etape2	xFront	205.6	90.43	2.27	40.38	5.09
13876	FES2DH	255.9	145.75	1.75	95.81	2.67
normal	xFront	190.16	91.4	2.08	39.7	4.79
13896	FES2DH	276.86	146.8	1.89	95.2	2.91
moyen	xFront	1800	886	2.03	402	4.48
55179	FES2DH	2930	1600	1.83	1113	2.63

These results deserve some comments. First, we note that the collected speedups for Flo2DH are often within the theoretical bounds (expected speedups) calculated in Section 3.1. As far as the parallelized subroutine xFront is concerned, the obtained speedups could be considered satisfactory especially for big problems, i.e. for a high number of nodes. The exception of test problem ourika is explained by the fact that the graph partitioning algorithm was not able, in this case, to ensure that $k \ll n_i$ (see Section 3.3).

In the second comment, we want to stress the fact that even though the parallel version of Flo2DH does not markedly improve the computational performance of the sequential version, it continues to have its practical and empirical importance. From the practical point of view, our implementation has permitted the direct method, and

also, but with a lesser order of importance, the simulator Flo2DH, to be run faster. Whenever additional fractions of the simulator are parallelized, Flo2DH could further benefit in terms of efficiency. From an empirical point of view, our code could become a benchmarker for evaluating the solution quality of other codes using iterative solvers.

These advantages could even become more significant whenever the solution of large scale problems is faced. We expect, indeed, from our parallel code, to be able to solve large problems (with a huge number of elements) that are unsolvable on conventional computers because of lack of memory. High performance facilities will allow, in this case, not only a temporal advantage (speedup) but also a spatial advantage (memory) by exploiting the distributed resources that could not be available otherwise.

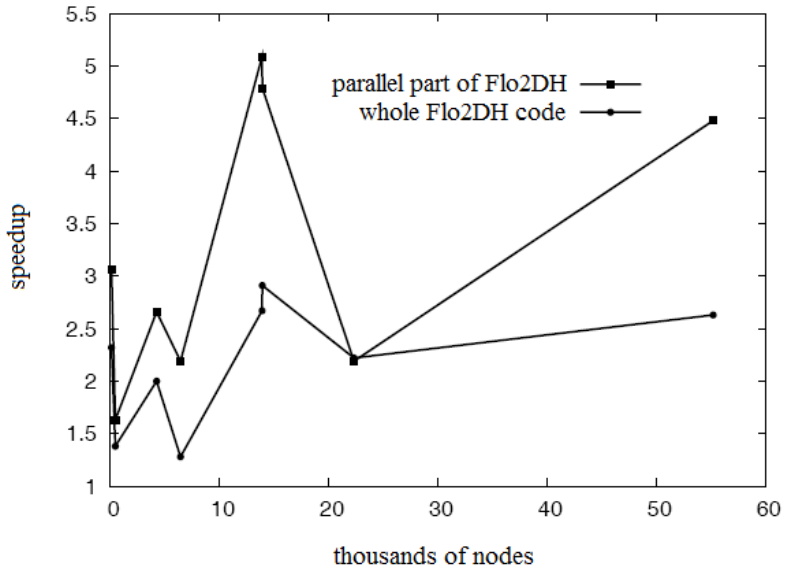

Figure 5. Speedups on 8 processors.

5. Conclusions

In this paper we dealt with the flood propagation problem and we proposed a parallel implementation of the simulator Flo2DH for its solution. Our parallelization effort has focused on the solution of the linear system arising in the Newton method since it is the most time consuming part of the simulator. We proposed an appropriate matrix decomposition and we implemented graph-partitioning and element-resequencing strategies in order to balance the load among the processes. We tested the performance of the parallel code on a set of real-life problems and we obtained satisfactory speedups, especially for non trivial problems. Our experimental experience has been restricted to the solution of a set of test problems that are characterized by a limited size. This was due to the difficulty of collecting the necessary data for real large-scale problems. We expect, however, from our parallel code, to be able to solve large scale problems that currently sequential computers, even though quite powerful, are not able to solve.

Finally, we notice that our parallel implementation can be improved in various ways. One of these is to perform the graph partitioning not at each iteration of the Newton method, as in the current version, but only when necessary. In this case it is necessary to define a criterion that decides whether the current partitioning is still valid. This can be the subject of future research.

6. References

ABDEOUAHED, M., KARRAKCHOU, S., NAKHLE, B. and TALAMALI, S. 2000. Elaboration et Implementation HPCN de CRUCID. *Technical Report, INRIA-Rocqencourt*.

AMESTOY, P.R., DUFF, I.S. and L'EXCELLENT, J.-Y. 1998. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Technical Report RAL-TR-1998-051*, Rutherford Appleton Laboratory, Oxford, UK.

HENDRICKSON, B. and LELAND, R. 1994. Chaco: *Software for Partitioning Graphs*. Sandia Technical Report SAND94-2692. (http://www.cs.sandia.gov/~bahendr/chaco.html).

DABAGHI, F., GUELMI, N., HENINE, H., NAKHLE, B., KACIMI, A. and TAIK, A. 2004. *Flood Forecasting and Flood Wave Propagation Modeling*. T.R. D4.2 INRIA.

FROEHLICH, D.C. 2002. *User's Manual for FESWMS Flo2DH*. U.S. Department of Transportation. FHWA-RD-03-053.

GEORGE, A. and LIU, J.W. 1981. *Computer Solution of Large Sparse Positive Definite Matrices*. Prentice Hall, QA 188.G46.

HLUCHY, L., TRAN, V.D., ASTALOS, J., DOBRUCKY, M., NGUYEN, G.T. and FROEHLICH, D. 2002. Flood modeling system and its parallelization. *In Proc. International Conference on Parallel Computing in Electrical Engineering, Poland*, pp. 277-281.

HLUCHY, L., HABALA, O., MALISKA, M., SIMO, B., TRAN, V.D., ASTALOS, J. and BABIK, M. 2006. Grid based flood prediction virtual organization. *In Proc. Second IEEE International Conference on e-Science and Grid Computing, The Netherlands.*

MALLYA, J.U., ZITNEY, S.E., CHOUDHARY, S. and STADTHERR, M.A. 1997. A parallel frontal solver for large scale process simulation and optimization. *AIChE J.*, **43**: 1032-1040.

MPI - *The Message Passing Interface Standard* (http://www-unix.mcs.anl.gov/mpi).

MPE (library and tools) - *Performance Visualization for Parallel Programs* (http://www-unix.mcs.anl.gov/perfvis).

LEBRUN, M.J. and FURNISH, M. 1994. *The PLplot Plotting Library* (http://plplot.sourceforge.net/docbook-manual/plplot-5.9.9.pdf).

ROTTA, R. 2008. *Multi-level Graph Clustering* (http://studiy.tu-cottbus.de/~clustering/algorithms : refinement #kernighan-lin_refinement).

SCOTT, J.A. 2001. The design of a portable parallel frontal solver for chemical process engineering problems. *Computers in Chemical Engineering*, **25**: 1699-1709.

SCOTT, J.A. 2002. Parallel frontal solvers for large sparse linear systems. *Technical Report RAL-TR-2002-033*, Rutherford Appleton Laboratory, Oxford, UK.

WIKIPEDIA. 2012. (http://en.wikipedia.org/wiki/Graph-partitioning).

Received 9 May 2011
Accepted 15 October 2011