PARALLEL/VECTOR IMPLEMENTATION OF A SPLIT-CHARACTERISTIC BASED FEM FORMULATION FOR SHALLOW WATER EQUATIONS

Ramiro Brito Willmersdorf
Paulo Roberto Maciel Lyra
Universidade Federal de Pernambuco, Departamento de Engenharia Mecânica, 50740-530, Recife, PE, Brasil.
Pablo Ortiz
Centro de Estudios de Técnicas Aplicadas, CEDEX, Alfonso XII,3,E-28014, Madrid, Spain.

Abstract

The solution of large scale geophysical flows is important for realistic prediction of natural phenomena. Such analysis demand very long computational times and large amount of computer memory. We present a parallel implementation for shared memory computers of a finite element method for the solution of the Shallow Water equations. A semi-explicit split Characteristic-Galerkin formulation is used, which removes the wave celerity from the computation of the stability limit. This formulation allows the use of Galerkin-type spatial discretization with fractional time steps, in which the velocities are computed explicitily and the pressure filed is computed through the solution of a system of linear algebraic equations, originated from a Laplacian-type equation, solved using the conjugate gradient method. The parallelization of the computational system is done with the introduction of compiler directives to force the most important loops of the program to run concurrently on more than one processor. We analyse the performance of the parallel implementation and robustness of the formulation with simulation of some model problems and some realistic geophysical flows.

Keywords: Parallel Processing, Finite Element Method, Hydrodynamic Flow Simulation

1. INTRODUCTION

Shared memory parallel computers have become commodity items. Dual processor motherboards for personal computers are readily available and quite affordable, while motherboards with four processors, suitable to very high performance workstations, are easy to find and cost a few thousand dollars. A complete parallel computer, with four very high performance Xeon processors, whose performance is higher than most scientific supercomputers of a few years ago, can be bought for less than ten thousand dollars.

Multitasking operating systems can and do make use of this multiple processors to run several jobs simultaneously, therefore increasing the throughput of the system. In the field of computational mechanics, however, the main attraction of these multiprocessor systems is to reduce the running time of numerical simulation. We will show how easily we can obtain very good performance gains on a program for solving shallow water equations. The structure of this program, in particular, the parts of the program that are responsible for most of the running time, are very similar to programs that are used for the solution of incompressible viscous flows, Euler equations and even the full Navier-Stokes equations, therefore the same good performance can be expected in these programs.

2. SHALLOW WATER ALGORITHM

Zienkiewicz and Ortiz (1995) developed the algorithm and computer implementation for the solution of the shallow water equations, and these are described in detail in his work. The algorithm is based on a fractional step procedure. The semi-explicit formulation allows the critical time step to depend on the velocity and not on the wave celerity, as is the case with fully explicit Taylor-Galerkin approximation. This formulation is therefore better suited to the analysis of long wave propagation in shallow waters.

The pressure (or elevations of the free surface)is calculated by solving a Laplacian-type equation. This self-adjoint equation ensures that the Galerkin space discretization is optimal and allows the use of a conjugate gradient solver. The velocities are computed explicitly in two stages with the characteristic-Galerkin method, first ommiting the pressure gradient term from the momentum equations and then using the new computed pressure to correct the velocity terms.

3. COMPUTATIONAL ASPECTS

One of the simplest, but most useful, tools for analysing and improving the performance of a computer program is the run time execution profile. This is a record of how much time is spent on each routine. Modern development environments have sophisticated graphical tools to help creating and analysing the profile, and can even use the profile information during the optmization phase to increase the performance of the program. The shallow water solver used in this work is relatively small, with no great algorithimic complexity, so the simple unix tools, “prof” and “gprof” were used.

A summary of the profile data for the shallow water program is show in table 1 for some representative runs of the program. This profile data was obtained using a PC with a AMD-450Mhz processor and 128Mbytes of memory. We also took profiles with a Sun Enterprise 450, with similar results. In all cases, the programs were run for 200 iterations. In table 1, “Total time” lists the total execution time for the 200 iterations, and `pcgsol`, `get1st`, and `getend` are the percentage of the total execution time spent on each of this routines for each case. The remaining routines are not listed since the time spent on each of them is always very small.

“Small-C”, “Medium-C”, and “Large-C” are model of the same rectangular channel with different meshes. “Severn” is a model of the Severn Estuary in Wales, and “Anular” is a model of an annular harbour. This problems are described in the work of Zienkiewicz and Ortiz (1990). Clearly, any effort to improve the performance of this program must be directed first to the routine `pcgsol`, which accounts for about three quarters of the total run time. Additional effort can be spent on the routine `get1st`, and it is very questionable if any effort to optmize any other routine is worthwhile.

The routine `pcgsol` is an implementation of a Jacobi preconditioned conjugate gradient (PCG) method for the solution of a system of linear equations. This routine is called every time step during the computation of the pressure. The routine `get1st` updates the velocity in an explicit form and prepares the right hand side of system of linear equations for the PCG solution

Table 1: Run time execution profile

	Small-C	Severn	Medium-C	Anular	Large-C
Elements	32	256	860	2739	18430
Nodes	28	172	461	1445	9426
Total time(s)	0.28	3.6	10.5	34.8	282
Step time (s)	0.0014	0.018	0.053	0.174	1.4
pcgsol	54%	75%	75%	76%	76%
get1st	9.7%	14%	9.5%	9.8%	8.0%
getend	9.7%	3.0%	3.8%	3.5%	3.9%

in every time step.

3.1 PCG Operations

The PCG implementation used in this program is based on an EBE (element by element) formulation. In this formulation, the global “stiffness” matrix is never fully assembled, all operations are done with element matrices (King and Sonnad, 1987). There are three basic operations on this PCG implementation: vector additions, vector reductions and matrix vector multiplications (matvecs). These operations are described in detail below.

Vector-additions: These are loops over all the nodes in the mesh, with the following form (in pseudo fortran)

```
do ip = 1, npoin
   rg(ip) = frhs(ip) - apg(ip)
enddo
```

where `npoin` is the number of nodes in the finite element mesh, and `rg`, `frhs` and `apg` are vectors used in the PCG routine. The term “addition” is being used as a generic term for any algebraic operation between two vectors done node by node.

Vector-reductions: These are loops that obtain an scalar value out of a vector that spans all nodes of the mesh of the problem, typically when computing norms. A typical loop would be

```
aln = 0.0
do ip = 1, npoin
   aln = aln + rg(ip)*sg(ip)
enddo
```

where `rg` and `sg` are PCG vectors.

Matvecs: These loops are the core of the PCG routine. They implement the multiplication of the global “stiffness” matrix by a vector an element at a time, without forming the global matrix. A typical loop is

```
do ie = 1, nelm
   i1 = intma(1,ie)
   i2 = intma(2,ie)
   i3 = intma(3,ie)
   apg(i1) = apg(i1)+
      ke(1,ie)*pg(i1)+ke(2,ie)*pg(i2)+ke(4,ie)*pg(i3)
   !  Similar lines for nodes i2 and i3
enddo
```

where `intma` is the nodal conectivity array, `ke` stores the stiffness matrices for all elements, and `apg` and `pc` are PCG vectors. Of course the indices that appear on `ke` depend on the format of the storage of the element matrices. The implementation of the PCG method used in this work consists of a sequence of iterations, repeated until some convergence criterium is met, where each iteration is composed of the above operations. The sequence of global iterations is, by definition, sequential, therefore any attempt to improve the performace of this routine with parallel processing must be applied to the operations done inside each iteration, ie, the loops described above.

3.2 Get1st Operations

This routine prepares the right hand side of the sistem of algebraic linear equations for the PCG solution. It consists of a large loop of all elements, done on every time step. In this loop, most operations are local to each element. There are two situations when global values are updated. Global vectors are either indexed (therefore, updated) directly by the number of the element being considered, or they are indexed by the number of the nodes of the element being considered. We will see later that these are very different when this loop is to be executed in parallel. A simplified representation of the typical operations in this loop is:

```
do ie = 1, nelm
   area = 0.5 * geome(7,ie)    !  Local scalar variable
   ···                         !  Many local operations
   do ino = 1, 3               !  Compute element variables
      localy = ···
      ···                      !  More local element operations
   enddo
   ···                         !  Many local loops as above
   fxsec(ie) = ct2 * delte(ie) !  Global update by elm.  number
   ···
   do in = 1, 3                !  Global update by node number
      ip = intma(in,ie)
      rhs(1,ip) = rhs(1,ip) + localy
   enddo
   ···                         !  Many operations as above
enddo
```

4. PARALLEL IMPLEMENTATION

Most, if not all, modern shared memory parallel computers are built following the CREW (concurrent read exclusive write) model (see Akl, 1989). Different processors are allowed to read the same memory location at the same time, however they cannot update the same memory location concurrently. The easiest and less intrusive way to parallelize the routines discussed above is doing it loop by loop, either automatically or with the introduction of compiler directives in the code. We used a Sun Microsystems Enterprise 450, with four 233Mhz processors, running Solaris 2.6, and all programming was done in the Workshop environment using version 4.2 of the fortran 77 compiler. The command line used for compilation of a single program module named `pcgsol.f` was:

```
f77 -fast -parallel -mp=sun -reduction -loopinfo -vpara
                  -c pcgsol.o pcgsol.f
```

where `-fast` activates sequential optmization, `-parallel` turns on automatic and manual loop parallelization, `-mp=sun` chooses Sun's syntax for parallelization directives, `-reduction` turns on recognition of vector reductions and `-loopinfo` and `-vpara` are informational options. These options are fully documented online, on Sun Microsystems documentation site: `http://docs.sun.com/`. In this program, each routine is in its own separate file, so this command line was used to compile just two files, `pcgsol.f` and `get1st.f`. The remaining modules were compiled with the sequential optmization only. Of course, compatible options must also be passed to the linker when building the complete program.

When the compiler parallelizes a loop, each processor executes a subset of the loop iterations. It is necessary that the iterations be independent of each other, ie, an iteration must not depend on results of previous iterations. It is also necessary to ensure that more than one processor cannot write to same memory location at the same time, but there is no restriction of concurrent reads of the same location.

The actual scheduling of the iterations, ie, which iteration is assigned to each specific processor, depends on a policy that can be choosen by the user. For this problem though, all the iterations last the same time, there is no serious load balancing problem. The default scheduling policy, where iterations are assigned to processors in a round robin way, works very well and was adopted (see Sun's online documentation site).

The operating system must make calls to system libraries every time a parallel loop is started. This is very expensive in comparison with the startup of a sequential loop. This parallel overhead is very important, and only long loops can be parallelized efficiently. In fact, too short loops can actually run slower in parallel than sequentially. In many cases, the compiler cannot determine the number of iterations of a loop during compilation, so it generates both a sequential and a parallel version of the loop and decides which one to use when the program runs. Many times the programmer does know which loops will be short and can avoid this extra overhead with a compiler directive to force sequential compilation (`C$PAR DOSERIAL`).

4.1 PCG Operations

To improve the performance of the PCG routine, we have to parallelize the three types of loops described above: vector additions, vector reductions and matrix vector multiplications. The first two require no modifications to the source code, and are automatically recognized by the compiler, while the third requires a small change to source code.

Vector-additions: All iterations of the loop are obviously independent, and there is no possibility of different processors writing to same memory location, since the `ip` are necessarily different for different processors. The compiler recognizes this and these loops are automatically parallelized.

Vector-reductions: These loops are potentially problematic, since all iterations write to the same variable. This reduction cannot, as it is stated, be done in parallel. It is necessary to introduce a temporary variable for each processor, in which partial results for each processor are accumulated. When all processors finish, the partial results are accumulated by one of them into the final variable. The compiler does all of this automatically when given the appropriate option (`-reduction`).

Matvecs: There is obviously a problem with these loops. The loop is indexed by element number, while the global vector is updated by the number of the nodes of the element. In principle, it is quite possible that different processors be assigned different elements with a common node. It is then possible that these two (or more) processors try to update the same position of the vector `apg` at the same time, which would immediately crash the program.

This kind of data dependency is a traditional problem for vector processing, and it has been dealt with historically by “colouring” the mesh (see Hughes, 1987). To colour the mesh means to divide the mesh in disjoint subsets of elements, ensuring that no two elements in each subset share a node. This technique neatly solves the data dependency problem and has been sucessfully used for parallelization of finite element programs. Each colour is processed sequentially, and all processors work concurrently on the elements of each colour. As the elements of each colour do not have common nodes, all updates can be done in parallel in complete safety.

This approach has some drawbacks, however, for the kind of problem we were solving in the kind of parallel computer we were using. The meshes for our typical problems usually have a few thousand elements. These meshes are coloured with eight to eleven colours, on the average. The lenght of the inner loop over the elements of each colour is the number of elements divided by the number of colours times number of processors. In a typical case, we would have the number of elements divided by forty, or about forty or fifty elements per processor. In many of the examples we tried, we had just ten or even less elements per processor. The parallel overhead completely dominated the run time of these loops and we could not get acceptable efficiency. We decided to try a different approach. We split the loop in two, in the following manner:

```
do ie = 1, nelm
   i1 = intma(1,ie)
   i2 = intma(2,ie)
   i3 = intma(3,ie)
   lpg(1,i1) = apg(i1) + ke(1,ie)*pg(i1) + ke(2,ie)*pg(i2) +
      ke(4,ie)*pg(i3)
   !  Similar lines for other nodes
enddo
do ipoin = 1, npoin
   do ie = 0, iecount(ipoin)-1
      icurr = lstart(ipoin) + ie
      apg(ipoin) = apg(ipoin) + lpg(enodn(icurr),elmnod(icurr))
   enddo
enddo
```

where `lpg` is an array for the local element vectors and `iecount`, `lstart`, `enodn` and `elmnod` define, for each node, a list of elements that contain the node. The list is simple data structure: `iecount` stores the number of elements to which the node belongs, `lstart` is the position where the list of the elements for that node starts, `elmnod` is the number of the element that has that node and `enodn` is the position of the node in the connectivity array of the element. The remaining variables are as above. Both these loops are recognized by the compiler and parallelized automatically.

The matvec loops can be understood as a three stage operation, even if these three stages are not explit. First, data is *gathered* from the global vector into a local vector (with size equal to the number of nodes of the element). Then element matrix is multiplied by the local vector, storing the result in the same local vector. Finally, the result of the multiplication (the local vector) is accumulated in the global vector (a *scatter-add* operation.)

It is safe to do the first stage in parallel with a loop indexed by element number since there is no problem in reading the same memory location concurrently. The second stage can also be done in parallel in the same loop, without any mesh colouring, *if* there is a separate local vector for each element. Of course, this results in a considerable increase in the memory usage of the program. As the program use triangular elements, a vector of three times the number of elements in the mesh is necessary. Fortran 77 does not have dynamic memory allocation, so a vector of three times the *maximum* number of elements must be allocated during compilation. There are schemes to simulate dynamic memory allocation in fortran 77 programs, and, if lack of compliance with the language standard is acceptable, full memory allocation can be used. We decided not to employ any of these schemes, neither to use fortran 90 memory management constructs, to maintain compatibility with the original program and because our problems do not really use that much memory.

The third stage, the scatter-add, cannot be done in parallel in a loop indexed by element number without mesh colouring. We decided to separate this stage from the other two, splitting the matvec operation in two loops: the first indexed by element number does the gather and the multiplication; the second, indexed by node number, does only the final scatter-add. This second loop can be safely done in parallel since no two processors are ever assigned the same node number.

4.2 Get1st Operations

The `get1st` operations can be easily parallelized. Global updates by element number are trivially parallelized, just like the vector additions in the PCG routine. Global updates by node number present the same difficulty as the matvecs in the PCG routine, and can be treated in the same way. All local updates can be done in parallel, if the compiler keeps a copy of the variable for each processor (which can easily be requested by the programmer.) The only difficulty is that, due to the size and apparent complexity of the loop, the compiler misdetects data dependencies and does not parallelize the loop. A compiler directive (`C$PAR DOALL`) must be inserted before the loop to force the parallelization. When this directive is used, the compiler must know which variables of the loop are private to each iteration, and which variables are shared among all processors. The compiler has a simple heuristic (scalar variables are private, arrays are shared) that does not work for our program, so we have to state explicitly, with directives, which variables are shared and which are private. The synxtax of these directives is: `C$PAR& SHARED(list of variables)` and `C$PAR& PRIVATE(list of variables)`. All variables that are used in the loop must appear in one of these two lists.

5. RESULTS

Table 2 shows the sequential and parallel run times for the examples described above. Also shown are the speed up (the sequential time divided by the parallel time, ideally, should be equal to the number of processors) and the efficiency (the speed up divided by the number of processors, ideally, should be equal to 1). The parallel overhead is quite evident, since the smallest problems actually run slower in parallel. This is expected. The larger problems, however, present very reasonable speed ups considering how few modifications were made to the program.

6. CONCLUSIONS

We used a shared memory parallel computer to obtain important reductions in the running time of a shallow water program. This economy allows the solution of larger scale problems or provides results in a shorter time frame. The necessary modifications of the source code are very simple and limited to a few routines, and the parallelization directives are easy to use. Parallel computers like the one used is this paper are already readily available, and will certainly take on an even more important role on scientific computation. The techniques presented here can be used to same effect on similar computer programs, and the same performance is expected.

Table 2: Parallel Performance

	Small-C	Severn	Medium-C	Anular	Large-C
Time Steps	2000	1000	1000	1000	500
Seq. time(s)	1.8	12	59	220	528
	Two Processors				
Par. time(s)	2.9	14	44	135	295
Speed Up	.62	.85	1.34	1.76	1.79
Efficiency	32%	42%	67%	88%	89%
	Three Processors				
Par. time(s)	–	–	37	105	220
Speed Up	–	–	1.6	2.1	2.4
Efficiency	–	–	53%	70%	80%
	Four Processors				
Par. time(s)	–	–	36	88	186
Speed Up	–	–	1.6	2.5	2.9
Efficiency	–	–	40%	63%	70%

7. REFERENCES

- Akl, S.G., 1989, “The Design and Analysis of Parallel Algorithms”, Prentice-Hall International, UK, 401p.
- Hughes, T.J.R, Ferencz, R.M. and Hallquist, J.O., 1987, “Large Scale Vectorized Implicit Calculations in Solid Mechanics on a Cray X-MP/48 Utilizing EBE Preconditioned Conjugate Gradients”, Computer Methods in Applied Mechanics and Engineering, vol 61, pp 215-248.
- King, R.B. and Sonnad, V., 1987, “Implementation of an Element-by-Element Solution Algorithm for the Finite Element Method on a Coarse Grained Parallel Computer”, Computer Methods in Applied Mechanics and Engineering, vol 65, pp 47–59.
- Sun Microsystems, 1998, “Sun Workshop Compilers FORTRAN 77 4.2”, Sun Microsystems, USA.
- Zienkiewicz, O.C., and Ortiz, P., 1995, “A Split-Characteristic Based Finite Element Model for the Shallow Water Equations”, International Journal for Numerical Methods in Fluids, vol 20, pp 1061–1080.