ground water

Methods Note/

A Parallel PCG Solver for MODFLOW

by Yanhui Dong[1] and Guomin Li[2]

Abstract

In order to simulate large-scale ground water flow problems more efficiently with MODFLOW, the OpenMP programming paradigm was used to parallelize the preconditioned conjugate-gradient (PCG) solver with in this study. Incremental parallelization, the significant advantage supported by OpenMP on a shared-memory computer, made the solver transit to a parallel program smoothly one block of code at a time. The parallel PCG solver, suitable for both MODFLOW-2000 and MODFLOW-2005, is verified using an 8-processor computer. Both the impact of compilers and different model domain sizes were considered in the numerical experiments. Based on the timing results, execution times using the parallel PCG solver are typically about 1.40 to 5.31 times faster than those using the serial one. In addition, the simulation results are the exact same as the original PCG solver, because the majority of serial codes were not changed. It is worth noting that this parallelizing approach reduces cost in terms of software maintenance because only a single source PCG solver code needs to be maintained in the MODFLOW source tree.

Introduction

MODFLOW (Harbaugh et al. 2000) has been widely used to simulate ground water flow using the finite-difference method for decades. While this code performs satisfactorily in solving most relatively simple two- or three-dimensional problems, it takes a large computational effort when applied to solve problems with massive grids, such as a large-scale ground water flow model. There are two ways to significantly shorten computing time. The first way is to improve and refine the computational methods, such as preconditioned conjugate gradient (PCG) methods (Hill 1990a) and the link-algebraic multi-grid (LMG) Package (Mehl and Hill 2001) linking MODFLOW to an algebraic multi-grid (AMG) solver for matrix equations. The other way is using parallel computing methods to improve the computational performance of the numerical codes on supercomputers. Recent parallel computer architectures provide both increased performance with respect to execution time and offer an amount of memory storage that significantly exceeds traditional single central processing unit (CPU) computers. In this paper, an incremental parallelization method is employed to parallelize the PCG solver package (Hill 1990b) of MODFLOW. Incremental parallelization supports a smooth transition from serial to parallel processing on shared memory multiprocessors by allowing parallelizing only parts of a given serial program. Shared memory multiprocessors have the ability for all processors to access all memory as global address space, enabling interaction and synchronization with each other through shared variables. This kind of parallel computer is readily available due to the emerging trends of multicore CPU that combines two or more independent cores into a single package composed of a single integrated circuit.

The purpose of this paper is to present a parallel version of the PCG package for MODFLOW, which allows parallel processing on shared memory multiprocessors. First, the OpenMP programming paradigm is sketched. Then, it is used to parallelize the PCG solver package.

[1]Key Laboratory of Engineering Geomechanics, Institute of Geology and Geophysics, Chinese Academy of Sciences, P. O. BOX 9825 Beijing, 100029 China.

[2]Corresponding author: Guomin Li, Key Laboratory of Engineering Geomechanics, Institute of Geology and Geophysics, Chinese Academy of Sciences, P. O. BOX 9825 Beijing, 100029 China; guominli@mail.iggcas.ac.cn

Received January 2009, accepted May 2009.

doi: 10.1111/j.1745-6584.2009.00598.x

Finally, performance results of the parallel PCG solver are reported.

Methods

Open MP Programming Paradigm

OpenMP (OpenMP Architecture Review Board 2005) is an application program interface (API) that may be used to explicitly direct multithreaded, shared memory parallelism. It is comprised of three primary API components: compiler directives, runtime library routines, and environment variables, and works in conjunction with FORTRAN, C, or C++. OpenMP has recently emerged as a shared memory standard, which is jointly defined and endorsed by a group of major computer hardware and software vendors.

OpenMP, based upon the existence of multiple threads in the shared memory programming paradigm, uses the fork-join model of parallel execution (Quinn 2004). When the program begins execution, only a single thread, called the master thread, is active (Figure 1). The master thread executes the sequential portions of the algorithm. At those points where parallel operations are required, the master thread forks (creates or awakens) additional threads. The master thread and the created threads work concurrently through the parallel section. At the end of the parallel code, the created threads die or are suspended, and the flow of control returns to the single master thread, called a join. A sequential program is a special case of an OpenMP program: it is simply one with no fork/joins in it. OpenMP programs range from those with only a single fork/join around a single loop to those in which most of the code segments are executed in parallel. Hence OpenMP supports incremental parallelization, the process of transforming a sequential program into a parallel program one block of code at a time. This feature is very helpful on paralleling the PCG solver.

More information about OpenMP can be found at the web site: http://www.openmp.org.

Parallelizing the PCG Solver Package

In this study, an incremental approach to parallelize the PCG solver package with OpenMP was carried out. Using OpenMP directives, the parallelization can be applied separately to individual subroutines without changing the rest of the (serial) program. Moreover, since parallelism is specified via compiler directives, only a single source tree needs to be maintained.

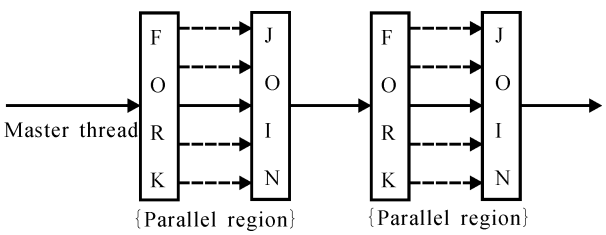

Figure 1. The fork/join parallelism of OpenMP.

PCG solver package is a numerical code to be used with MODFLOW. The solver uses the preconditioned conjugate-gradient method to solve the equations produced by the model for hydraulic head. It includes two preconditioning options: modified incomplete Cholesky preconditioning (MICCG) and polynomial preconditioning (POLCG). Owing to its stronger preconditioner, MICCG is generally more efficient than POLCG. However, POLCG is a good parallel algorithm, whereas MICCG is not. In the study, POLCG was fully parallelized to take full advantage of parallel architectures, whereas MICCG was partially parallelized.

The ability of OpenMP to support incremental parallelization allows one to profile the execution of the sequential solver program, sort the program blocks according to how much time they consume, consider each block in turn beginning with the most time-consuming, parallelize each block amenable to parallel execution, as well as stop when the effort required to achieve further performance improvements is not warranted.

Profiling the execution of the sequential PCG solver code reports that the blocks with a three-level nested DO-Loop took a large part of execution time. The block structure is:

```
DO K = 1, NLAY
    DO I = 1, NROW
        DO J = 1, NCOL
                ...
            Some computation
                ...
        CONTINUE
    CONTINUE
CONTINUE
```

The variables K, I, and J are the loop counter of each DO-Loop. The variables NLAY, NROW, and NCOL denote the number of layers, rows, and columns in the model grid respectively. The block with a three-level nested DO-Loop will consume considerable execution time for a large model. Therefore, the parallelization focuses on how to parallelize the three-level nested DO-Loop blocks.

There are three issues that must be obeyed when parallelizing the blocks with OpenMP.

Grain Size

By analyzing the block code, each of the three loops could be executed in parallel. If parallelizing the inner loop, the program will fork and join threads for each iteration of the outer loop. The fork/join overhead may very well be greater than the time saved by dividing the execution of the n iterations of the inner loop among multiple threads. On the other hand, parallelizing the outer loop, the program only incurs the fork/join overhead once. Grain size is the number of computations performed between communication or synchronization

steps. In general, increasing grain size improves the performance of a parallel program. Making the outer loop parallel results in larger grain size, and therefore the best option.

Since three-dimensional models are widely used recently, parallelizing the K loop is considered. If the model has few layers, especially a two-dimensional model with a single layer, parallelizing the I loop is a better choice.

Control Flow

The three-level nested DO-Loop blocks often have a GOTO statement inside. The GOTO statement is not allowed in parallelization with OpenMP according to the current standard (OpenMP Architecture Review Board 2005). To circumvent this problem, the GOTO statement is removed and an additional IF block is introduced

```
DO K = 1, NLAY
        DO I = 1, NROW
                DO J = 1, NCOL
                        ⋯
                        IF ⋯ GOTO10
                        ⋯

                CONTINUE
        CONTINUE
CONTINUE
```

ensuring that the function performs the same with the original block, presented in the following code fragment.

```
DO K = 1, NLAY
        DO I = 1, NROW
                DO J = 1, NCOL
                        ⋯
                        IF ⋯ THEN
                        ⋯
                        ENDIF
                CONTINUE
        CONTINUE
CONTINUE
```

Data Scoping

To parallelize the three-level nested DO-Loop blocks, simply direct the compiler to execute the iterations of the loop indexed by K in parallel. However, extra attention should be paid to the variables accessed by the threads. By default, all variables are shared except loop index K. That makes it easy for threads to communicate with each other, but it can also cause problems. When multiple threads try to execute different iterations of the K loop in parallel, all of the threads try to initialize and increment the same shared variable I and J, increasing the chance that threads will not execute all NROW iterations of I loop and all NCOL iterations of J loop. The solution is clear that all three indexes of the three-level nested DO-Loop should be made private variables by using the OpenMP private () clause.

Another problem is the race condition, in which the computation exhibits nondeterministic behavior when performed by multiple threads accessing a shared variable. For example, consider the case in which data are overwritten by one thread before that data can be read by another thread. To illustrate, consider the following code fragment.

The directive-pair !$OMP PARALLEL DO/!$OMP END PARALLEL DO indicates that the different iterations can be executed by different threads in parallel. The shared () clause declares variables in its list to be shared among all threads in the team. The private () clause indicates that each thread maintains a private copy of the loop index N. The update of SRNEW must be performed by only one thread at a time. Otherwise, a race condition on variable SRNEW occurs. The OpenMP reduction () clause, which creates for each thread a private copy of the variables that appear in its list, should be employed. Thus, a private copy of the variable SRNEW is created for each thread. At the end of the reduction, the variable SRNEW is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

Performance Results

This section shows how the parallel PCG solver performed when solving large ground water problems with MODFLOW-2000. The first test exhibits the impact of three different FORTRAN compilers. The second test examines the performance for different domain sizes of the same problem. POLCG was used to solve ground water flow problems in these two tests. The third test shows the comparisons between POLCG and MICCG.

All timing experiments were performed on a CentOS 5.0 workstation equipped with two quad-core 2.66 GHz Intel Xeon processors and 16 GB RAM. The speedup, defined as the ratio of the serial PCG solver computing time and the parallel PCG solver computing time, is used to assess the performance of parallel computing.

Test 1—Different Compilers

The first test case, TWRI_LARGE, has 160 rows, 160 columns, and 40 layers, simulating steady-state flow. In order to determine the impact of different compilers, three executables have been generated by the Intel FORTRAN compiler 10.1 (ifort), the Lahey FORTRAN 8.1 (lfc), and the GNU FORTRAN compiler 4.2 (gfortran) using *–openmp*, *–openmp*, *and –fopenmp* compiler flags, respectively. Eight threads were used to run the parallel program in this case.

Execution times and speedup for the three compilers are shown in Figure 2. A maximum speedup of 5.31 times was achieved by using gfortran. However, the execution time is also the longest. On the contrary, the execution time by using ifort is shortest, but the speedup is smallest. It is noticeable that ifort makes most of the Intel Xeon processor, on which the workstation is based. Provided

```
!$OMP PARALLEL DO
!$OMP& SHARED (NODES, IBOUND, SS, RES)
!$OMP& PRIVATE (N)
!$OMP& REDUCTION (+: SRNEW)
      DO N = 1, NODES
            IF (IBOUND (N).GT.0) SRNEW = SRNEW + SS (N)*RES (N)
      ENDDO
!$OMP END PARALLEL DO
```

the workstation consists of AMD-based processors, the speedup for ifort should not be less than 3.83, which is approximately the average speedup achieved on our workstation for the three compilers.

Test 2—Different Domain Sizes

The following test case is presented of regional ground water flow in Beishan area, which is being studied as a potential site for a Chinese high-level nuclear waste geologic repository. The study area includes above 70,000 km^2 and lies within the area bounded by latitude 40° and 42° North and longitude 96° and 100° East. A numerical model of three-dimensional flow has been applied for different domain sizes (Table 1). The executable has been generated by the GNU FORTRAN compiler 4.2 (gfortran) using *–fopenmp* compiler flags in this test.

The benefit of parallelization is shown in Figure 3, which reports the execution times using one to eight threads for the three domain sizes. The speedup of No. 3 model goes up as the threads increasing and the maximum speedup of 4.49 times was achieved by using eight threads. For No. 1 model and No. 2 model, the speedup rises slowly, even goes down a little for No. 1 model when more than five threads are used. It may indicate that overhead of communication and synchronization begins to affect the performance and a larger speedup number could not be achieved by additional threads. In general, the larger the domain size, the greater the speedup.

Table 2 shows the results of the required memory vs. number of threads for the three models. Only 8 or 12 more kilobytes of memory are required for one additional thread. More memory is required with each thread that is launched with variables being shared and/or restricted on shared memory computers. However, it is minor compared to the memory taken by the model itself.

The simulation results of the parallelized version using different threads are exactly the same as the original version.

The speedup of 1.89 with two threads for No. 3 model is remarkably good. The simulation of MODFLOW, which is typically running on desktop PCs equipped with dual-core or quad-core processors, clearly gains great benefit from the parallel PCG solver since this immediately leads to a reduced response time of the whole MODFLOW execution.

It is needed to mention that parallelizing the I loop is a better choice for models with only a single layer. This parallelizing option was also tested for the same problem which is discretized into 1000 rows, 1000 columns, and one layer. A maximum speedup of 4.25 times can be achieved using eight threads with this parallel PCG solver.

Table 1
Three Models with Different Domain Sizes

Model	Layers, Rows, Columns	Nodes
No. 1	200, 200, 10	400,000
No. 2	1000, 1000, 10	10,000,000
No. 3	1000, 1000, 40	40,000,000

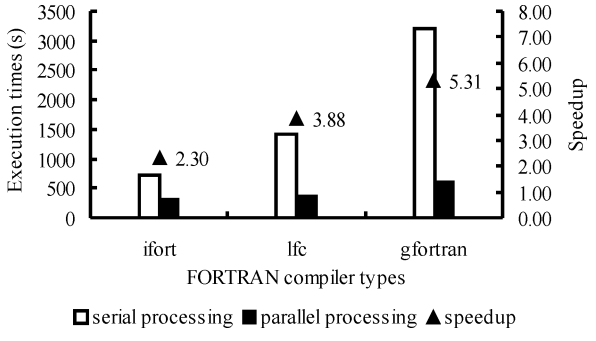

Figure 2. Execution times and speedup for three compilers using eight threads.

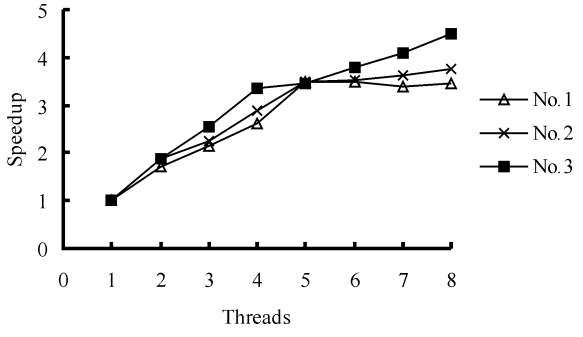

Figure 3. Number of threads vs. speedup for different domain sizes.

Table 2
The Required Memory vs. Number of Threads for the Three Models (kB)

	Threads							
Model	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
No. 1	37,252	37,264	37,272	37,280	37,288	37,296	37,304	37,312
No. 2	892,256	892,268	892,276	892,284	892,292	892,300	892,308	892,316
No. 3	3,470,384	3,470,396	3,470,404	3,470,412	3,470,420	3,470,428	3,470,436	3,470,444

Table 3
Comparison of Computational Differences between POLCG (in parallel) and MICCG (Both in Serial and in Parallel)

		MICCG (in serial)		MICCG (in parallel)		POLCG (in parallel)	
Problem	**Nodes (Layers, Rows, Columns)**	**CPU Time (s)**	**Memory (MB)**	**CPU Time (s)**	**Memory (MB)**	**CPU Time (s)**	**Memory (MB)**
1	1,024,000 (160, 160, 40)	430	91	304	91	601	91
2	40,000,000 (1000, 1000, 40)	10,627	3376	7330	3377	10,018	3389
3	100,000,000 (1000, 1000, 100)	38,981	8393	28,942	8394	24,637	8424

Test 3—Comparisons between POLCG and MICCG

For comparison, three numerical models with different domain size were solved using both POLCG and MICCG. First, MICCG was used to solve the problems in serial. Then, both POLCG and MICCG were used in parallel respectively with eight threads on the workstation. The executable used was generated by the GNU FORTRAN compiler 4.2 (gfortran).

Problem 1 is the TWRI_LARGE model in test 1. Problem 2 is a model from test 2 and problem 3 is a refined model. Comparisons are made based on the execution time and memory required. Table 3 shows the execution time and the memory required.

Three results could be observed from Table 3. First, parallel POLCG can reduce the execution time compared to serial MICCG for problem 2 and 3. Parallel POLCG could reduce execution times about 37% compared with serial MICCG for problem 3. Second, it is clear that execution times using parallel MICCG are about 1.4 times faster than those using serial MICCG. Finally, the changes of memory requirements are minor compared to the memory taken by the models themselves. Memory requirements of parallel POLCG are almost same as those of parallel MICCG and there are almost no changes of memory requirements between parallel MICCG and serial MICCG.

These results indicate that both parallel POLCG and partially parallel MICCG are very effective. This parallel PCG solver is a good choice for ground water flow problems, especially models with massive grids.

Conclusions

By specifying parallelizing compiler directives that are ignored as comments by uniprocessor compilers, the PCG solver package of MODFLOW is partially parallelized using OpenMP, the de facto standard for shared-memory parallel computing. This way, a single version of PCG solver is used for both serial and parallel computing, considerably reducing the human effort for software maintenance and development.

OpenMP provides for a compact, yet powerful programming model for shared memory programming and is an attractive technique to reduce the execution times arising from sophisticated MODFLOW simulations for real-world case studies in ground water modeling. Execution times using the parallel PCG solver are typically about 1.40–5.31 times faster than execution times using MODFLOW's PCG2 Package. In summary, with the increased availability of multicore processors in commercial off-the-shelf hardware, the parallel PCG solver can be an exciting tool for large-scale ground water flow modeling.

OpenMP is designed for shared memory systems. However, Intel's Cluster OpenMP supports running an OpenMP program on distributed shared memory systems such as clusters, which are a substantial part of the parallel computing environment. Future research directions will involve the parallelization of the PCG solver of MODFLOW on more advanced parallel computers consisting of network architecture.

Software Availability

The source code of the parallel PCG solver may be requested free of charge via e-mail from the corresponding author of this article (guominli@mail.iggcas.ac.cn).

Acknowledgments

We would like to thank Dr. Mary C. Hill for her encouragement and perspective. We would also like to thank Dr. Richard L. Naff for his help with the research. We are grateful for review comments and suggestions from Dr. Steffen Mehl, Dr. Yu-Feng Lin, and the anonymous reviewer that greatly improved the manuscript. This research was supported by Grant No. kzcx2-yw-116 from the Chinese Academy of Sciences' Knowledge Innovation Program. Partial support was also provided by Grant 40672170 from the National Natural Science Foundation of China and Research Fund for the Environmental and Grant D07050601510000 from Beijing Science and Technology Program.

References

Harbaugh, A.W., E.R. Banta, M.C. Hill, and M.G. McDonald. 2000. MODFLOW-2000, the U.S. Geological Survey modular ground-water model—User guide to modularization concepts and the Ground-Water Flow Process. U.S. Geological Survey Open-File Report 00-92. Reston, Virginia: USGS.

Hill, M.C. 1990a. Solving groundwater flow problems by conjugate-gradient methods and the strongly implicit procedure. *Water Resources Research* 26, no. 9: 1961–1969.

Hill, M.C. 1990b. Preconditioned conjugate-gradient 2 (PCG2), a computer program for solving ground-water flow equations. U.S. Geological Survey Water-Resources Investigations Report 90-4048. Denver, Colorado: USGS.

Mehl, S.W., and M.C. Hill. 2001. MODFLOW-2000, the U.S. Geological Survey modular ground-water model –User guide to the link-AMG (LMG) package for solving matrix equations using an algebraic multigrid solver. U.S. Geological Survey Open-File Report 01-177. Denver, Colorado: USGS.

OpenMP Architecture Review Board. 2005. *OpenMP Application Program Interface*. Version 2.5. http://www.openmp.org/mp-documents/spec25.pdf.

Quinn, M.J. 2004. *Parallel Programming in C with MPI and OpenMP*. Beijing: McGraw-Hill and Tsinghua University Press.

Author's Note: We declare that we have no conflict of interest.