XVIII International Conference on Water Resources
CMWR 2010
J. Carrera (Ed)
©CIMNE, Barcelona, 2010

AUTO-OPTIMIZATION ON PARALLEL HYDRODYNAMIC CODES: AN EXAMPLE OF COHERENS WITH OPENMP FOR MULTICORE

Francisco López-Castejón* **and Domingo Giménez**†

*Polytechnic University of Cartagena, Spain
e-mail: francisco.lopez@upct.es

†University of Murcia, Spain
e-mail: domingo@um.es, web page: http://www.um.es/pcgum/

Key words: Parallel code auto-optimization, Hydrodynamic models, COHERENS, OpenMP

Summary. This paper analyses the development of parallel implementations for shared memory systems from simulation codes typically constituted by a number of functions and each function by a number of loops in which some work is carried out on matrices and vectors which represent the state of the simulation at each step. The idea is to show how parallel code can be easily generated, and so the simulations are performed in a more reduced time. Furthermore, some ideas on how to include auto-optimization techniques in this parallel code are studied. With auto-optimized codes the user should have parallel codes available which automatically (without user intervention) select the values of some parameters with which low execution times are obtained.

1 Introduction

The protection of marine systems and the understanding of the physical, chemicals and biological processes present in it, is today a problem of great interest, both for the scientific and private sector. A hydrodynamic marine model (*HMM*) is a numerical, spatial and temporal representation of the most important physical-chemical processes which affect the behaviour of the sea water [10]. The capacity to facilitate an approximate representation of reality makes the *HMM* a useful tool to study the marine environment and it behaviour.

There has been an important advance on the *HMM* in recent years, mainly due to the availability of multicore systems and the development of parallel codes [2, 11]. OpenMP is now the standard tool for developing parallel code for multicore machines . These systems have allowed the generalization of the use of parallel codes to carry out simulations. The use of more complex systems, like supercomputers or clusters of processors is an alternative and allows bigger problems to be solved in lower simulation times, but at the expense of more difficult programming, and furthermore, some times the scientific groups or the small companies do not have easy access to these more complex systems, but do have access to multicore systems, which are now the standard computational systems, and are the basic components of clusters and supercomputers. So, multicores and their parallel programming are at present a fundamental tool to work with *HMM*.

But, the user of the models is normally a non expert in parallelism, and the development of parallel code which efficiently uses the computational resources is not an easy task. There are some parallel simulation packages [8, 9], but the use of parallelized code does not ensure a good use of these and consequently the computational system may not be as efficiently used as it could be. For example, the number of cores to use in the simulation is a factor which affects the execution time, as does the processes distribution (on a mesh algorithm the number of rows and columns of processes), and selecting the optimum number of cores is not simple. So, our proposal is to use an easy methodology to develop parallel code with auto-optimization capacity for multicore systems, so that the code automatically adapts to the problem and the system characteristics to obtain a reduced execution time.

In this paper different strategies for auto-optimization of OpenMP simulation codes are considered. These strategies could be incorporated in parallel *HMM*. The strategy consists basically of the selection of a different number of cores to work in each parallel part of the code, based on the computational cost of each part, so the execution time in each part is close to the lowest possible. The sequential *HMM* COHERENS [7] has been selected to show the parallelization and auto-optimization methodology, but the same ideas can be applied to other simulation codes [1, 3].

2 General structure of COHERENS

The COHERENS model (COupled Hydrodynamical-Ecological model for REgioNal and Shelf seas) was developed between 1990 and 1999 by the *Management Unit of the North Sea Mathematical Models*, *Napier University*, *Proudman Oceanographic Laboratory* and *British Oceanographic Data Centre*, under European project *MAST PROFILE, NOMADS AND COHERENS.*

COHERENS solves the Navier-Stokes equations with the *splitting* method [6, 5]. This method splits the simulation of the water movement in two modes: in the 2D mode (external mode) the mean transport of a column of water is solved, and in the 3D mode (internal mode) the water flow is calculated at each of the levels at which the vertical axis has been discretized. Inside a temporal loop, the values of the variables are calculated; the 2D mode is applied in all the steps of the loop, and the internal mode is applied in only some steps. The number of 2D steps per each 3D step can be decided, and a value of 10 normally gives satisfactory results [4].

3 Paralellization strategy

The execution time of the program has been analysed to decide a good parallelization strategy. To do so, the number of flops (floating point operations) on each function in the program has been obtained. Figure 1 shows the cost of the main functions in COHERENS. For each function, the number of flops has been represented as a function of the variables x (number of nodes in axis X), y (number of nodes in axis Y) and z (number of Z levels).

Each of these main functions is composed of a number of subfunctions. The number of flops associated to each subfunction is obtained, and from that the number of flops of each function. CRRNT2 is the function with the highest computational cost ($350xy + 86xy + 86xy$ flops).

The first step in the parallelization of the code is to determine the variables which are shared by all the threads, and those which must be private and replicated in the different threads. The access to each matrix and vector in each loop is analysed. Figure 2 shows one of these loops. The access to position i in matrices *ydiflv*, *xdiflu*, *xdiflv* and *ydiflu* needs to access the previous and posterior positions, and so these matrices must be shared. Once the different variables have been computed, all the threads write in the result matrix *vdh2d*, which is also shared. In this code, variables *ydifv, xdifv* and *ydifu* are private, because each thread uses them independently.

4 Inclusion of auto-optimization in parallel code

With an auto-optimization methodology, a parallel code is obtained which automatically (without user intervention) adapts to the characteristics of the computational system, the parallel code and the input to solve. The code and the theoretical execution time are parameterised with some values which are decided to obtain executions close to the optimum at execution time. The parameters to decide are the number of threads to use in each loop in the code. Prior to each loop execution, the OpenMP function `omp_set_num_threads` is used to establish the number of threads to work in the loop (figure 2). So, the parallel code is not run with the maximum number of available cores (which could produce a reduction in the performance) and a different number of cores is used for each loop, depending on the computational cost inside the loop and the problem size.

The loops in the COHERENS code are classified depending on the number of flops in them. Experiments with loops with 3, 8 and 19 flops show that for different problem sizes the optimum number of cores depends on the problem size and the number of flops. Experiments have been carried out in four systems, one at the Polytechnic University of Valencia, one at the Polytechnic University of Cartagena and two at the Supercomputing Centre of Murcia. All the experiments have been performed with one node, and in three of the systems each node has a total of eight cores, and the other is a Superdome with 128 cores, but only 64 cores have been used. Figure 3 shows the evolution of the optimum number of cores for a loop with 19 flops when the problem size varies. The maximum

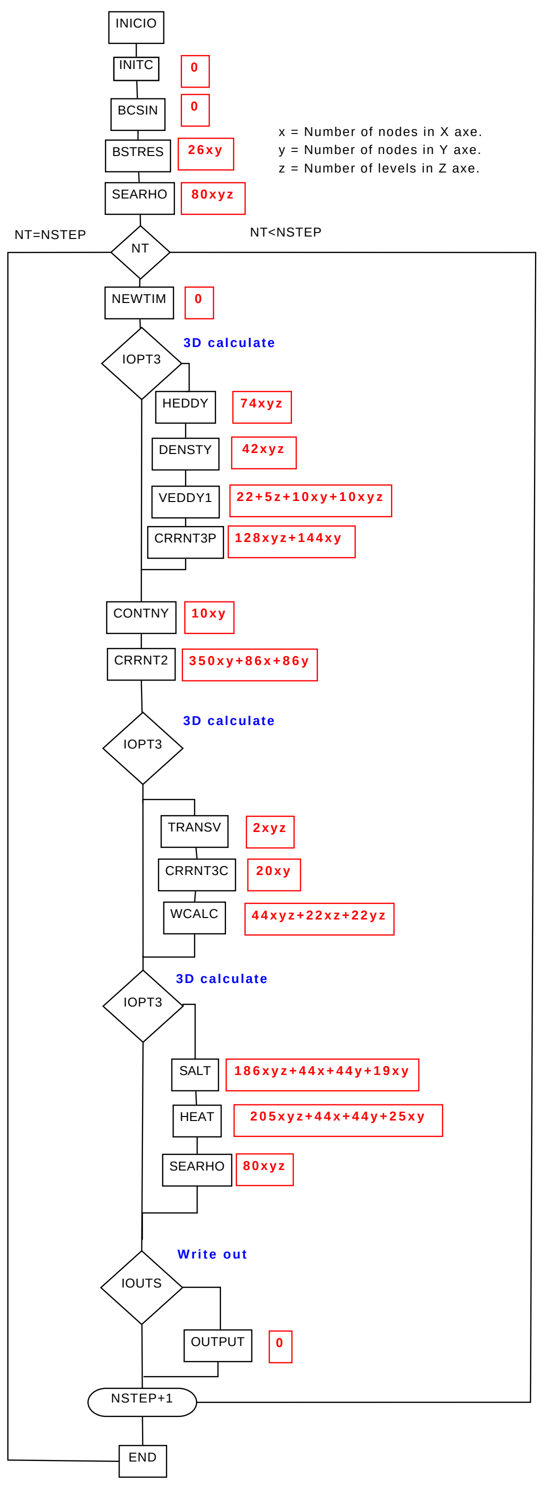

Figure 1: Computational cost of the main functions in COHERENS.

```
omp_set_num_threads()
c$omp parallel
c$omp& private (i,j,ydifv,xdifv,ydifu)
c$omp do
        do i=1,nc
        do j=2,nr
          if (npiy(j,i).eq.1) then
           ydifv = (yduflv(j,i)-ydiflv(j-1,i))/(gy2v(j)*cosphiv(j))
           xdifv = 0.5*(xdiflv(j,i+1) + xdiflv(j-1,i+1) -2
     1                    xdiflv(j,i)    - xdiflv(j-1,i))/gx2v(j,i)
           if (i.eq.1) then
            ydifu = (ydiflu(j,i+1) - ydiflu(j,i))
     1               /(0.5*gx2v(j,i+1)+1.5*gx2v(j,i))
           elseif (i.eq.nc) then
            ydifu = (ydiflu(j,i) - ydiflu(j,i-1))
     1               /(0.5*gx2v(j,i-1)+1.5*gx2v(j,i))
           else
            ydifu = (ydiflu(j,i+1) - ydiflu(j,i-1))
     1               /(0.5*(gx2v(j,i-1)+gx2v(j,i+1))+gx2v(j,i))
           endif
           vdh2d(j,i) = ydifv + xdifv + ydifu
           vdh2d(j,i) = vdh2d(j,i) + sphcurv(j)*
     1          (0.5*(xdiflu(j-1,i)+xdiflu(j,i))
     2           -2.0*sphcurv(j)*dheddyvv(j,i)*vd2(j,i)/h2atv(j,i))
          endif
         end do
        end do
c$omp end do
c$omp end parallel
```

Figure 2: Parallelization with OpenMP of the sequential loop.

number of cores is 8 or 64, but the optimum has not always the maximum possible value, and is different for each system, which means the auto-optimization should be system-specific.

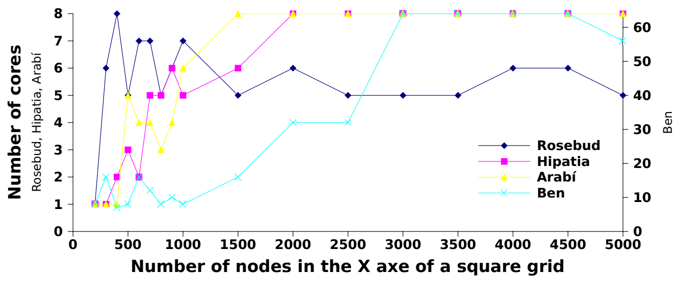

Figure 3: Optimum number of cores for a loop with 19 flops, when varying the problem size, in four systems.

The optimum number of cores is obtained experimentally by running the loop with one core, then with two, and so on, until the execution time with p cores is higher than with $p-1$ cores. Then, $p-1$ is taken as the optimum number of cores for the loop.

The selection of the values of the parameters (in this case the number of cores to use in each loop) can be made at installation time or at execution time:

- When the code is being installed in a system, some experiments can be carried out to determine the number of cores on each loop. Experiments for representative problem sizes are carried out. Some default problem sizes are provided, and the system manager (who knows about the system, the code and the typical problems to solve) could decide to experiment with these sizes or with other sizes which better represent the mean behaviour of the simulation to be made. Then, each particular problem will be run with the optimum number of cores determined at installation time for each loop and for the problem size closest to that of the problem being solved. This form of work supposes a considerable running time at installation, but it can reduce the time of successive simulations. On the other hand, the problems and inputs used in the installation may not represent the behaviour of the simulations for particular inputs well.

- Another possibility is to obtain the optimum number of cores for each loop at running time. An adaptive code is used. In the first temporal step one core is used for each loop. In the second two cores are used. The loops in which the execution time with two cores is higher than with one core finish their adaptation part. The rest of the loops carry out the third step with three cores. And the adaptation steps continue until all the loops have obtained their optimum number of cores. The problem here is the adaptation steps are carried out with a far from the optimum number of cores, and so the execution time of the initial steps can be very high and not compensate the reduction obtained with the use of a number of cores close to the optima after the adaptation has finished.

- It is also possible to combine experimentation at installation time with adaptation at execution time. The optimum number of cores decided at installation time is used as starting point for the adaptation process. So, if the optimum number of cores in the installation for loops of a certain computational cost and problems of a certain size is p, the first step is carried out with p cores, the second with $p + 1$, and so on until the execution time increases. After that, the adaptation continues with $p - 1$, $p - 2...$ cores while the optimum execution time decreases.

Experiments have been carried out for different problem sizes and loops with different computational costs. Figure 4 shows the quotient between the execution time obtained with the parameters provided by the proposed methodology and that with the maximum number of cores. For small sizes the methodology allows the user to obtain execution times better than those using the complete system (values lower than 1), and for bigger problems the optimum number of cores is the maximum available, and the methodology provided a number of cores close to the optimum (quotient close to 1).

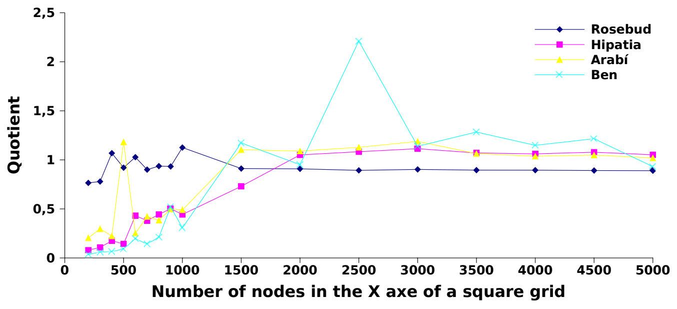

Figure 4: Quotient between the execution time obtained with the number of cores selected by the proposed methodology and that with the maximum number of cores, for a loop with 19 flops, when varying the problem size, in four systems.

5 Conclusions and future work

The paper analyses a methodology to easily develop and auto-optimize shared memory parallel codes based on OpenMP. Because OpenMP is used to program todays multicore systems, which are the basic components of clusters and supercomputers. The methodology is applicable to a wide range of systems and can be used to accelerate the simulation of physical processes.

The methodology has been analysed with COHERENS, but it can be used in other packages with a similar structure. It has not been integrated in the simulation package, and so the next work to do is to integrate the methodology with the package, analyse its application to other simulation packages and validate it with experiments in a wider number of multicore systems with different characteristics.

Acknowledgements

This work has been partially supported by the Consejería de Educación de la Región de Murcia (Fundación Séneca, 08763/PI/08), and by the Spanish Ministerio de Ciencia e Innovación (TIN2008-06570-C04-02/TIN).

The experiments have been carried out in the computers: *Rosebud*, of the Parallel Computing Group of the Polytechnic University of Valencia, *Hipatia*, of the Polytechnic University of Cartagena, and *Arabi* and *Ben* (Superdome), of the Supercomputing Centre of Murcia.

REFERENCES

[1] J. Cuenca, D. Giménez, and J. González. Architecture of an automatic tuned linear algebra library. *Parallel Computing*, 30(2):187–220, 2004.

[2] Robert M. Hunter, Jing-Ru C. Cheng, Hwai-Ping Cheng, and Tim Campbell. Parallel coupled watershed-nearshore model development. In *Computational methods in water resources*, 2008.

[3] Sonia Jerez, Juan-Pedro Montávez, and Domingo Giménez. Optimizing the execution of a parallel meteorology simulation code. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*. IEEE, May 2009.

[4] Zygmunt Kowalik and Tadepalli Satyanarayana Murty. *Numerical modeling of ocean dynamics*. World Scientific, 1993.

[5] R. V. Madala and S. A. Piacsck. A semi-implicit numercial model of baroclinic oceans. *Computational Physical*, 23:167–178, 1977.

[6] T. J. Simons. Verification of numerical models of lake Ontario. *Physical Oceanography*, 4:507–523, 1974.

[7] COHERENS web page. `http://www.mumm.ac.be/~patrick/mast/`.

[8] MM5 web page. `http://www.mmm.ucar.edu/mm5/`.

[9] Parallel Ocean Program webpage. `http://climate.lanl.gov/Models/POP/`.

[10] Brian Williams. *Hydrobiological Modelling*. Brian Williams, 2006.

[11] D. Yanhui and G. Li. A parallel-computing method for modeling large-scale groundwater flow. In *Computational methods in water resources*, 2008.