ПРОГРАММИРОВАНИЕ

MSC 68U99 **DOI: 10.14529/mmp170107**

BALTIC SEA WATER DYNAMICS MODEL ACCELERATION

A.P. Bagliy, Southern Federal University, Rostov-na-Donu, Russian Federation, taccessviolation@gmail.com,
A.V. Boukhanovsky, ITMO University, St. Petersburg, Russian Federation, boukhanovsky@mail.ifmo.ru,
B.Ya. Steinberg, Southern Federal University, Rostov-na-Donu, Russian Federation, borsteinb@mail.ru,
R.B. Steinberg, Southern Federal University, Rostov-na-Donu, Russian Federation, romanofficial@yandex.ru

Industrial Baltic sea water dynamics modelling program optimization and parallelization is described. Program is based on solving the system of partial differential equations of shallow water with numerical methods. Mechanical approach to program modernization is demonstrated involving building module dependency graph and rewriting every module in specific order.

To achieve desired speed-up the program is translated into another language and several key optimization methods are used, including parallelization of most time-consuming loop nests. The theory of optimizing and parallelizing program transformations is used to achieve best performance boost with given amount of work. The list of applied program transformations is presented along with achieved speed-up for most time-consuming subroutines. Entire program speed-up results on shared memory computer system are presented.

Keywords: program transformation; program optimization; program parallelization.

Introduction

The article presents results of computer program modernization. This modernization included rewriting program code to another language and accelerating it. Speed-up was achieved by manual optimizing transformations and nested loops parallelization. Assessment of labour costs for program modernization and the resulting performance increase are presented.

Compiler transformations are not necessarily effective in all cases, so manual program transformations are still valid despite the progress in compiler optimization [1].

This is a result of group of authors performing modernization of software implementation for BSM-1 (Baltic Sea Model), which is currently running 24 hours a day 7 days a week at the St. Petersburg Flood Prevention Facility Complex [2]. BSM-1 model is based on numerical integration of partial differential equations system based on shallow water theory. The model is the evolution of a family of BSM (Baltic Sea Model) models used within CARDINAL software system [3]. This article hereafter describes the modernization process and its results, aforementioned software system is referred to as "model program".

The particularity of the presented article lies in the fact that it describes program optimization process based on optimizing transformations only, without the knowledge of

the particular field software it is based on. Using knowledge from this field allows better optimization of software than just working with the source program code [4], but it is more time-consuming.

Current tool kit of software optimizing transformations contains numerous tools to optimize nested loops, which is a reasonable focus for software based on mathematical modelling methods. Nested loops in such programs usually take the largest part of execution time. Their optimization is a priority task, which is solved by parallelizing and other transformations. Currently the software performance is determined not only by code parallelism but optimization of memory accesses [5–8] as well. Optimizations of memory accesses include alignments [9], tiling (transition to block computations) [7, 10–13] and unconventional array placements [14–16]. The best performance is demonstrated for the programs that are parallel and optimize memory access simultaneously [6, 7, 17–20]. The model software, which wasn't initially optimized in any way, had other time-consuming parts in addition to nested loops. These parts included I/O functions, data structure preparation and elementary mathematical functions.

The model program inherited CARDINAL complex main features. It was written in Object Pascal language using Delphi 6 IDE and included graphical user interface for setting the computational model and the main computational core. Only this core part was subject to optimization.

The software model was subjected to numerous optimizing transformations, many of which cannot be performed automatically by optimizing compilers [6, 7]. The task was completed thanks to authors' experience working on Optimizing Parallelizing system [21].

1. Underlying Mathematical Model

Consider shallow water equations [3] in curvilinear coordinates in two-dimensional form, that are being solved in the model software:

$$\begin{aligned} U_t + \left(\frac{U^2}{H}\right)_x + \left(\frac{UV}{H}\right)_y &= -gH\varsigma_x - \frac{gH^2}{2\rho_0}\overline{\rho}_x - \frac{H}{\rho_0}\frac{\partial P_q}{\partial x} + \\ &\quad + fV + K\Delta U + C_D\frac{\rho_q}{\rho_0}w_(x)|\overline{W}| - f_b\frac{U|\overline{V}|}{H^2}, \\ V_t + \left(\frac{UV}{H}\right)_x + \left(\frac{V^2}{H}\right)_y &= -gH\varsigma_y - \frac{gH^2}{2\rho_0}\overline{\rho}_y - \frac{H}{\rho_0}\frac{\partial P_a}{\partial y} - fU + \\ &\quad + K\Delta V + C_D\frac{\rho_a}{\rho_0}w_(y)|\overline{W}| - f_b\frac{V|\overline{V}|}{H^2}, \end{aligned}$$

$$\varsigma_t + U_x + V_y = \omega_s, \quad (\overline{c}H)_t + (U\overline{c})_x + (V\overline{c})_y = K_cH\Delta\overline{c} - \lambda\overline{c}H + \overline{c}_s\omega_s - f_s,$$

where u, v, $w(x, y, z, t)$ are velocity vector components in Cartesian coordinate system; $c(x, y, xz, t)$ is concentration; $T(x, y, z, t)$ is water temperature; $S(x, y, z, t)$ is salinity; $\varsigma(x, y, t)$ is water level; $h(x, y)$ is water depth; $H = h + \varsigma$; g is free fall acceleration; V is volume of water originating from internal sources in volume units per second; c_s is sources concentration; λ is non-conservative factor; k_c, K_c are diffusion coefficients in vertical and horizontal directions; w_0 is vertical speed sedimentation of suspended solids; K и k are turbulent viscosity coefficients in vertical and horizontal directions; P_A is atmospheric pressure; W is wind speed; ρ is water density; $f = 2\omega\sin\phi$ is Coriolis parameter; U and V

are total flows (unit discharge); ω_s is water volume coming from internal sources per unit area per second; f_b is bottom friction coefficient.

2. Software Optimization Methods

The following is done to the model program:

1. Program profiling for typical input data sets.
2. Finding program hotspots based on profiling results.
3. Experimental attempts at hotspot optimization. Goal of these experiments was to quickly identify the most efficient optimization method for particular pieces of code. Experiments are carried out for functions within source program or with the same functions but separated into individual programs for simplicity.
4. Optimization of program hotspots with chosen methods. Resulting speed-up measurement.
5. Testing the optimized program for correctness. Testing is performed by comparing modified functions with their original counterparts on the same input datasets.
6. Optimized program profiling to find new hotspots.

Operations mentioned above constitute an iterative development process which is based on the source program. The process stops when the required criteria are met.

The program optimization consists of a set of operations that reduce program execution time in typical use cases (on typical input data). Those operations may include:

1. Optimizing transformations including loop parallelization and other loop nest transformations. It makes sense to separate program transformations into groups:
   (a) Paralellizing loop transformations.
   (b) Loop optimizing transformations such as fission, unrolling, invariant extraction, etc.
   (c) Operator-level parallelization.
   (d) I/O optimization, including splitting I/O operations into separate thread.
   (e) Other optimizing transformations, such as inlining, common subexpression elimination ...
2. Rewriting parts of it or entire program into another programming language.
3. Replacing program parts with third-party library function calls.
4. Testing resulting program correctness and performance.

Rewriting entire program to another language is perhaps the most expensive way to increase performance. This, however, may improve other characteristics, such as ease of support and reuse, apart from performance. Rewriting is justified if the source program language lacks comprehensive optimizing compilers, high-performance libraries or its use is not justified for some other reasons. In case of model software it was Object Pascal used in Embarcadero Delphi 6.0 IDE. Main computational part was ported to C# language for Microsoft .NET 4.0 platform, which was done for ease of maintenance as well as performance boost.

3. Labour Costs Distribution

It's important to identify where most of the labour and time will be spent during sizeable program optimization. Working on model program showed the following main cost items:

1. Porting the program to another programming language.
2. Adding unit tests for all ported modules.
3. Profiling and finding suitable optimization methods.
4. I/O optimization.
5. Computational functions optimization including parallelization.
6. Functional testing and debugging.
7. Performance testing and performance analysis on the target computing system.

In total, the cost of model program modernization amounted to about 60 man-months. The labour costs distribution for the items listed above is shown on Fig. 1.

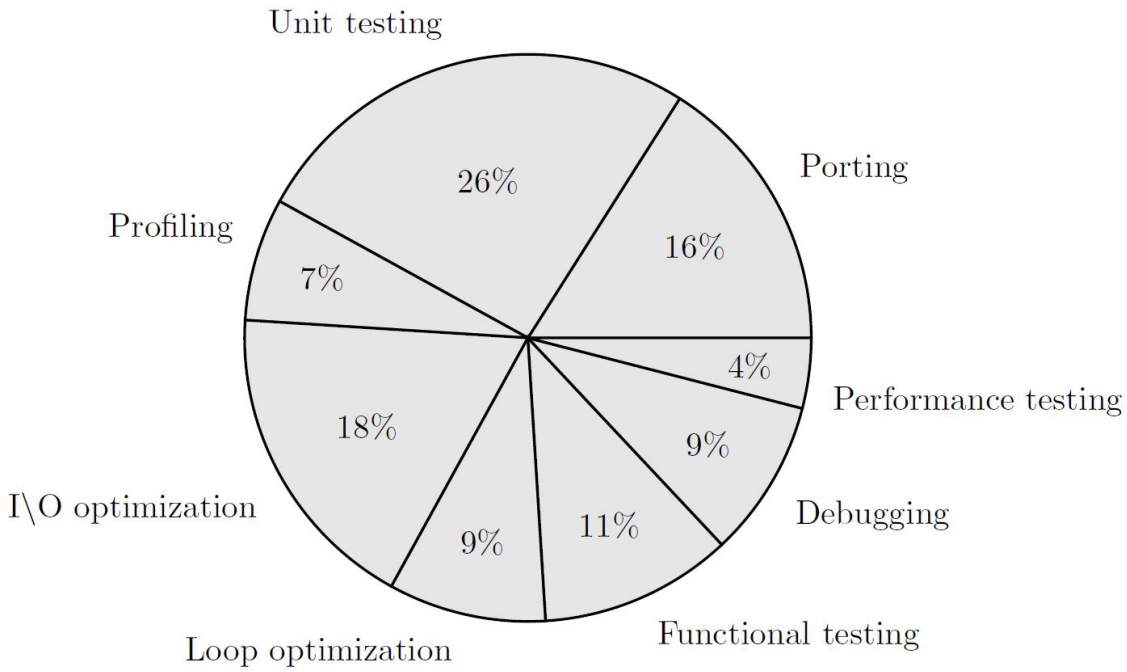

Fig. 1. Labour costs in model program modernization

4. Porting into C#. Results

Porting the model program into more modern language provided considerable performance increase. Table 1 shows comparison of execution times for main computational functions (that include nested loops), which took most of the program execution time. Performance boost achieved through porting to C# is 45%. All of this boost is due to better optimizing compiler for .NET platform. Compiler was able to perform inlining of small functions called from inside nested loops, as well as other optimizations. Apart from that, .NET standard library includes faster mathematical functions.

Table 1

Speed-up due to rewriting main computing functions, sec.

	TimeStep	GetPStar	GetQNew	QStar	PNew	DryCells
Source program	644	167	180	85	78	111
Optimized program	501	144	149	66	58	61
Speed-up	1,28	1,16	1,2	1,28	1,34	1,81

5. Program Modules Dependencies

Porting tens of thousands lines of code to a different language is quite an ambitious task taking into account necessary testing. The model program was subdivided into modules. Directed graph of modules dependencies was developed in order to determine module porting order. Unit dependencies appear because of access to global variables, functions or types. If unit A uses data types or functions declared in unit B, the graph has an edge leading from A to B. Edges determine the order of porting modules: from two modules the dependent one is ported later. This order simplifies porting and testing. Modules dependencies graph is similar to function call graph [22]. This graph for the model program is shown on Fig. 2. Typically the module hierarchy contains service units with simple auxiliary I/O functions and user input processing at the very bottom. It's reasonable to start porting the program from the lowest hierarchy level, sequentially climbing up with the completion of each level.

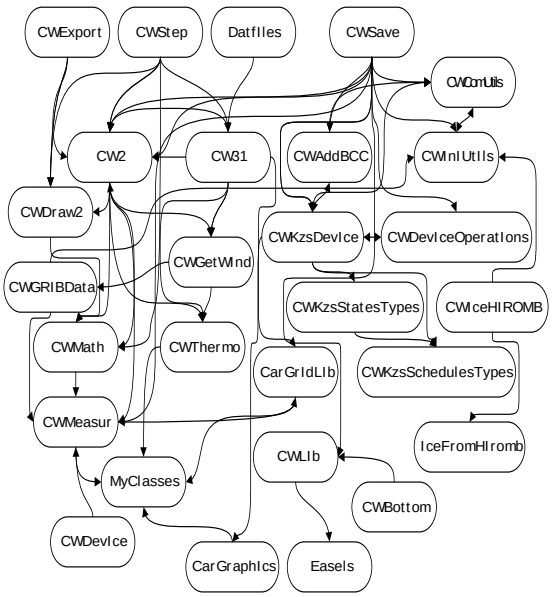

Fig. 2. Some source code modules dependencies within model software computational kernel

Porting order may be formally obtained as follows:

1. Construct the module dependency graph.
2. Construct a strongly connected components factor-graph for the module dependency graph.

3. The resulting factor-graph is acyclic, therefore there is a tiered form for it. Find this form. It has no edges between nodes within single tier and any edge goes downwards.

4. Modules porting order is given by tiers from bottom to top.

6. Program Transformations Used

The model program transformations used are listed in Table 2.

Table 2

Program transformations used

Transformation name	Program parts being transformed	Properties
Inlining	Nested loops inside hotspots, I/O, data preparation	Done automatically by the compiler for simple cases. Done manually for some hotspots
Loop invariant extraction	Nested loops inside hotspots	Allows to parallelize some loops due to data dependencies elimination
Loop interchange	Nested loops inside hotspots	Changing the order of matrix traversal which often benefits performance
Loop unrolling	Nested loops inside hotspots	Decrease the added cost of loops
Instruction extraction into a new function	Data preparation functions	Improves code readability and decreases its volume
Common sub expressions elimination	Nested loops inside hotspots	Common sub expressions could be loop invariants that are possible to extract
Loop parallelization	Nested loops inside hotspots	Done for loops without data dependences that prevent parallel execution
Asynchronous function calls	I/O	Allows to extract I/O into separate threads
Dead code removal	Data preparation functions	
Replacing global variables by local ones	Nested loops inside hotspots	Eliminates dependencies allowing parallelization
Extension of scalars (replacing a scalar variable with an array)	Nested loops inside hotspots	Eliminates dependencies allowing parallelization

Some transformations listed should be performed in specific order to obtain the highest speed-up. For example:

1. Common sub expressions elimination.

2. Loop invariants extraction.

3. Loop parallelization.

Finding an optimal manual transformation order is easy in some basic cases, while modern optimizing compilers lack such a mechanism. Optimizing transformations interrelation resulting in final speed-up is a complex and less studied process.

7. Parallelization

Program hotspots usually contain nested loops that perform calculations over data arrays. Parallelizing nested loops may provide the highest benefit compared to other program optimizations, but not all the loops could be parallelized due to data dependencies between iterations.

While working on the model software it was impossible to apply automatic tools. Manual data dependency analysis did not show any dependencies that could prevent parallelization. The most time consuming loops were transformed into parallel form with application of TPL (.NET Task Parallel Library). TPL is the main parallel execution tool for Microsoft .NET 4.0 platform. TPL use is comparable with OpenMP in many cases, keeping the code readability and maintainability.

Matrix elements are being read and written in parallelized two-dimensional loops. Matrix traversal is performed through even rows or columns in each iteration. Loop parallelization is performed as shown in Figure 3 for rows. Arrows demonstrate the order of matrix elements being written. Points without arrows going to them are not overwritten.

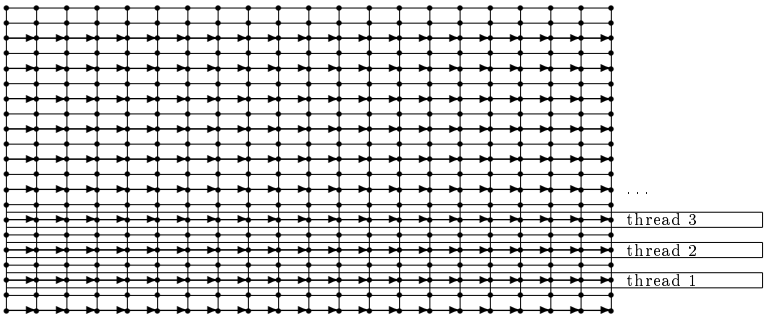

Fig. 3. Matrix element traversal order and its distribution into threads

Writing each row or column is separated into a parallel thread. This means outer loop parallelization for a two-dimensional nested loop. Table 3 demonstrates one loop nest before and after parallelization. Main computational functions speed-up as the result of parallelization is shown in Table 4. Substantial performance boost was obtained. But the performance increases disproportionately to the thread number increase due to low memory access speed compared to the speed of calculations inside the loops.

Table 3

Loop parallelization example

Source loop	Parallel loop

`int i;`
`i = g.Lbe + 1;`
`while (i <= g.Len - 1)`
`{`
`  int j = g.Nbe;`
`  ...`
`  while (j <= g.Nen)`
`  {`
`  ...`
`  }`
`}`
	`int i;`
	`i = g.Lbe + 1;`
	`int startI = (g.Lbe + 1) / 2;`
	`int finishI = (g.Len - 1) / 2 + 1;`
	`Parallel.For(startI, finishI, index =>`
	`{`
	`  int i = index * 2;`
	`  int j = g.Nbe;`
	`  ...`
	`  while (j <= g.Nen)`
	`  {`
	`  ...`
	`  }`
	`}`

Table 4

Main functions speed-up due to parallelization on 4-core processor, sec.

Function name	Sequentially	In parallel on 4 cores, 8 threads	Speed-up
GetPStar	75,8	22	3,4
GetQNew	81,8	25	3,3
QStar	32,9	8,4	3,9
PNew	23	6,3	3,6
WaterLevelStar	4,8	1,7	2,8
WaterLevelNew	4,8	1,6	3,0
DryCells	33	11,2	2,9

Conclusion

Main results of this work are:

1. New implementation of BSM-1 was created. It performs 3-7 times faster in typical use cases.
2. Structured approach to old software modernization is described.
3. Set of practical software optimization methods is listed.

The performance boost obtained is shown on Fig. 4. Execution time shown is for 7 different data sets while using different numbers of threads. Performance was measured on 6-core Intel Core i7-3930K processor with 8GB DDR3-1600 memory in single channel

mode. This computer system provided best results for optimized program. One can see that performance boost was largely not achieved with parallelization: optimized program operates sequentially 4 or more times faster than the original. While testing the optimized program on several computing systems discovered performance differences obviously were dependent on the following system features:

1. Memory Bandwidth.
2. The number of CPU cores.
3. Memory hierarchy, including different levels of cache.
4. CPU performance.

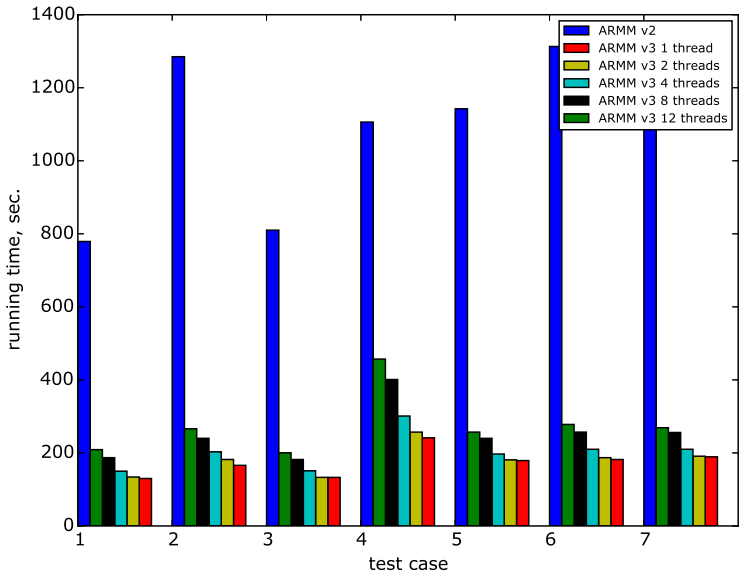

Fig. 4. BSM-1 speed-up for the set of 7 typical tasks

One needs to take these differences into account when optimizing software for high-performance computing system (server, cluster or supercomputer).

References

1. Metcalf M. *Fortran Optimization*. Academic Press, 1985. 264 p. DOI: 10.1002/spe.4380170208
2. Boukhanovsky A.V., Zhitnikov A.N., Petrosyan S.G., Sloot P. [High-Performance Technologies of Urgent Computing for Flood Hazard Prevention]. *Journal of Instrument Engineering*, 2011, vol. 54, no. 10, pp. 14–20. (in Russian)
3. Klevannyy K.A., Smirnova E.V. Using of Modeling System Cardinal for Solving Hydraulic Problems. *Vestnik Gosudarstvennogo Universiteta Morskogo i Rechnogo Flota Imeni Admirala S.O. Makarova*, 2009, no. 1, pp. 152–161. (in Russian)
4. Kovalchuk S.V., Ivanov S.V., Kolykhmatov I.I., Boukhanovsky A.V. [Special Characteristics of High Performance Software for Complex Systems Simulations]. *Information and Control Systems*, 2008, no. 3, pp. 10–18. (in Russian)

5. Gervich L.R., Kravchenko E.N., Steinberg B.Ya., Yurushkin M.V. Automatic Program Parallelization with Block Data Distribution. *Siberian Journal of Numerical Mathematics*, 2015, vol. 18, no. 1, pp. 41–53. (in Russian)
6. GCC Compiler Suite (2016). Available at: http://gcc.gnu.org/
7. Muchnik S. *Advanced Compiler Design and Implementation.* San Francisco, Morgan Kaufmann, 1997.
8. Kowarschik M., Weiß C. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. *Algorithms for Memory Hierarchies: Advanced Lectures.* Berlin, Heidelberg, Springer, 2003, pp. 213–232.
9. Kasperskiy C. *Tekhnika Optimizatsii Programm. Effektivnoe Ispol'zovanie Pamyati* [Program Optimization Techniques. Efficient Use of Memory]. St. Peterburg, BHV-Petersburg, 2003.
10. Gervich L.R., Steinberg B.Ya., Yurushkin M.V. [Exaflop Systems Programming]. *Open Systems. DBMS*, 2013, vol. 8, pp. 26–29. (in Russian)
11. Abu-khalil Zh.M., Morylev R.I., Steinberg B.Ya. Parallel Global Alignment Algorithm with the Optimal Use of Memory. *Digital Scientific Magazine "Modern Problems of Science and Education"*, 2013, no. 1, 6 p. Available at: http://www.science-education.ru/ru/article/view?id=8139
12. Korzh A.A. NPB UA Benchmark Scaling to Thousands of Blue Gene/P Cores Using PGAS-like OpenMP Extension. *Numerical Methods and Programming*, 2010, vol. 11, pp. 31–41.
13. Likhoded N.A. Generalized Tiling. *Doklady of the National Academy of Sciences of Belarus*, 2011, vol. 55, no. 1, pp. 16–21. (in Russian)
14. Denning P.J. The Locality Principle. *Communications of the Association for Computing Machinery*, 2005, vol. 48, no. 7, pp. 19–24. DOI: 10.1145/1070838.1070856
15. Gustavson F.G., Wyrzykowski R. Cache Blocking for Linear Algebra Algorithms. *Parallel Processing and Applied Mathematics 2011, Part I, Lecture Notes in Computer Science*, 2012, vol. 7203, pp. 122–132. DOI: 10.1007/978-3-642-31464-3_13
16. Lam M.S., Rothberg E.E., Wolf M.E. The Cache Performance and Optimizations of Blocked Algorithms. *Proceeding ASPLOS IV Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* Palo Alto, pp. 63–74.
17. Goto K., van de Geijn R.A. Anatomy of High-Performance Matrix Multiplication. *ACM Transaction on Mathematical Software*, 2008, vol. 34, no. 3, pp. 1–25. DOI: 10.1145/1356052.1356053
18. Mycroft A. Programming Language Design and Analysis Motivated by Hardwere Evolution (Invited Presentation). *The 14th International Static Analysis Symposium,* 2007, vol. 3634, pp. 18–33. Available at: http://www.cl.cam.ac.uk/am21/papers/sas07final.pdf
19. Galushkin A.I. [The Development Strategy of Modern Supercomputers on the Path to Ekzaflopsnym Computing]. *Prilozhenie k zhurnalu "Informacionnye tehnologii"*, 2012, no. 2. 32 p. (in Russian)
20. Arykov S.B., Malyshkin V.E. Asynchronous Parallel Programming System "Aspect". *Numerical Methods and Programming*, 2008, vol. 9, no. 1, pp. 48–52. (in Russian)
21. *Optimizing Parallelizing System* (2016). Available at: www.ops.rsu.ru
22. Ryder B.G. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, 1979, vol. SE-5, no. 3, pp. 216–226. DOI: 10.1109/TSE.1979.234183

Received May 8, 2016

УДК 519.688 **DOI: 10.14529/mmp170107**

УСКОРЕНИЕ МОДЕЛИ ДИНАМИКИ ВОДНЫХ МАСС БАЛТИЙСКОГО МОРЯ

А.П. Баглий, А.В. Бухановский, Б.Я. Штейнберг, Р.Б. Штейнберг

Описывается опыт оптимизации и распараллеливания промышленной программы моделирования динамики водных масс Балтийского моря, в основе которой лежат численные алгоритмы решения системы дифференциальных уравнений в частных производных теории мелкой воды. Демонстрируется механический подход к модернизации программы, включающий построение графика зависимости модулей и запись каждого модуля в определенном порядке. Для достижения желаемого ускорения работы программы используется теория оптимизирующих и распараллеливающих преобразований программ. Оптимизация и распараллеливание программы гарантирует достижение увеличения производительности при заданном объеме работы. Представлен ряд преобразований программы с полученными результатами по уменьшению скорости работы наиболее трудоемких процедур. Кроме того, приводятся результаты по ускорению работы программы в целом на вычислительной системе с общей памятью.

Ключевые слова: преобразования программ; оптимизация программ; распараллеливание программ.

Литература

1. Меткалф, М. Оптимизация в Фортране / М. Меткалф. – М.: Мир, 1985. – 264 с.
2. Бухановский, А. Высокопроизводительные технологии экстренных вычислений для предотвращения угрозы наводнений / А. Бухановский, А. Житников, С. Петросян, П. Слоот // Известия вузов. Приборостроение. – 2011. – № 10. – С. 14–20.
3. Клеванный, К. Использование программного комплекса CARDINAL / К. Клеванный, Е. Смирнова // Вестник Государственного университета морского и речного транспорта имени адмирала С.О. Макарова. – 2009. – № 3. – С. 153–162.
4. Ковальчук, С.В. Особенности проектирования высокопроизводительных программных комплексов для моделирования сложных систем / С.В. Ковальчук // Информационно-управляющие системы. – 2008. – № 3. – С. 10–18.
5. Гервич, Л.Р. Автоматизация распараллеливания программ с блочным размещением данных / Л.Р. Гервич, Е.Н. Кравченко, Б.Я. Штейнберг, М.В. Юрушкин // Сибирский журнал вычислительной математики. – 2015. – Т. 18, № 1. – С. 41–53.
6. Компиляторы GCC [электронный ресурс]. – 2016. – URL: http://gcc.gnu.org/
7. Muchnik, S. Advanced Compiler Design and Implementation / S. Muchnik. – Morgan-Kaufmann, 1997.
8. Kowarschik, M. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms / M. Kowarschik, C. Weiß // Algorithms for Memory Hierarchies: Advanced Lectures. – Berlin; Heidelberg: Springer, 2003. – P. 213–232.
9. Касперский, К. Техника оптимизации программ. Эффективное использование памяти / К. Касперский. – СПб.: БХВ-Петербург, 2003. – 300 с.
10. Гервич, Л. Программирование экзафлопсных систем / Л. Гервич, Б. Штейнберг, М. Юрушкин // Открытые системы. СУБД. – 2013. – № 8. – С. 26–29.
11. Абу-Халил, Ж.М. Параллельный алгоритм глобального выравнивания с оптимальным использованием памяти / Ж.М. Абу-Халил, Р.И. Морылев, Б.Я. Штейнберг // Современные проблемы науки и образования. – 2013. – № 1. – 6 с. – URL: http://www.science-education.ru/ru/article/view?id=8139

12. Корж, А.А. Результаты масштабирования бенчмарка NPB UA на тысячи ядер суперкомпьютера Blue Gene/P с помощью расширения OpenMP / А.А. Корж // Вычислительные методы и программирование. – 2010. – Т. 11. – С. 31–41.
13. Лиходед, Н.А. Обобщенный тайлинг / Н.А. Лиходед // Доклады НАН Беларуси. – 2011. – Т. 55, № 1. – С. 16–21.
14. Denning, P.J. The Locality Principle / P.J. Denning // Communications of the ACM. – 2005. – V. 48, № 7. – P. 19–24.
15. Gustavson, F.G. Cache Blocking for Linear Algebra Algorithms / F.G Gustavson, R. Wyrzykowski // Parallel Processing and Applied Mathematics 2011, Part I, Lectures Notes in Computer Science 7203. – 2012. – P. 122–132.
16. Lam, M.S. The Cache Performance and Optimizations of Blocked Algorithms / M.S. Lam, E.E. Rothberg, M.E. Wolf // Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. – 1991. – P. 63–74.
17. Goto, K. Anatomy of High-Performance Matrix Multiplication / K. Goto, R. van de Geijn // ACM Transactions on Mathematical Software. – 2008. – V. 34, № 3. – P. 1–25.
18. Mycroft, A. Programming Language Design and Analysis Motivated by Hardwere Evolution (Invited Presentation) / A. Mycroft // The 14th International Static Analysis Symposium. – 2013. – V. 3634. – P. 18–33. – URL: http://www.cl.cam.ac.uk/am21/papers/sas07final.pdf
19. Галушкин, А.И. Стратегия развития современных суперкомпьютеров на пути к экзафлопсным вычислениям / А.И. Галушкин // Приложение к журналу «Информационные технологии». – 2012. – № 2. – 32 с.
20. Арыков, С.Б. Система асинхронного параллельного программирования «Аспект» / С.Б. Арыков, В.Э. Малышкин // Вычислительные методы и программирование. – 2008. – Т. 9, № 1. – С. 48–52.
21. Оптимизирующая распараллеливающая система. – 2016. – URL: www.ops.rsu.ru.
22. Ryder, B.G. Constructing the Call Graph of a Program / B.G. Ryder // Software Engineering, IEEE Transactions on Software Engineering. – 1979. – V. SE-5, № 3. – P. 216–226.

Антон Павлович Баглий, научный сотрудник, Южный федеральный университет, институт математики, механики и компьютерных наук им. И.И. Воровича (г. Ростов-на-Дону, Российская Федерация), taccessviolation@gmail.com.

Александр Валерьевич Бухановский, доктор технических наук, заведующий кафедрой высокопроизводительных вычислений, Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики, институт наукоемких технологий (г. Санкт-Петербург, Российская Федерация), boukhanovsky@mail.ifmo.ru.

Борис Яковлевич Штейнберг, доктор технических наук, профессор, Южный федеральный университет, институт математики, механики и компьютерных наук им. И.И. Воровича (г. Ростов-на-Дону, Российская Федерация), borsteinb@mail.ru.

Роман Борисович Штейнберг, кандидат физико-математических наук, доцент, Южный федеральный университет, институт математики, механики и компьютерных наук им. И.И. Воровича (г. Ростов-на-Дону, Российская Федерация), romanofficial@yandex.ru.

Поступила в редакцию 8 мая 2016 г.