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:

\[
U_t + \left( \frac{U^2}{H} \right)_x + \left( \frac{UV}{H} \right)_y = -gH \zeta_x - \frac{gH^2}{2\rho_0} \frac{\rho_x}{\rho_0} - \frac{H}{\rho_0} \frac{\partial P}{\partial x} + fV + K \Delta U + C_D \frac{\rho_0}{\rho_0} w(x) |\nabla W| - f_b \frac{U |V|}{H^2},
\]

\[
V_t + \left( \frac{UV}{H} \right)_x + \left( \frac{V^2}{H} \right)_y = -gH \zeta_y - \frac{gH^2}{2\rho_0} \frac{\rho_y}{\rho_0} - \frac{H}{\rho_0} \frac{\partial P}{\partial y} - fU + K \Delta V + C_D \frac{\rho_0}{\rho_0} w(y) |\nabla W| - f_b \frac{V |V|}{H^2},
\]

\[
\zeta_t + U_x + V_y = \omega_s, \quad (cH)_t + (Uc)_x + (Vc)_y = K_c H \Delta c - \lambda \Delta c + \zeta \omega_s - f_s,
\]

where \( u, v, w(x, y, z, t) \) are velocity vector components in Cartesian coordinate system; \( c(x, y, z, t) \) is concentration; \( T(x, y, z, t) \) is water temperature; \( S(x, y, z, t) \) is salinity; \( \zeta(x, y, t) \) is water level; \( h(x, y) \) is water depth; \( H = h + \zeta \); \( g \) is free fall acceleration; \( V \) is volume of water originating from internal sources in volume units per second; \( c_s \) is sources concentration; \( \lambda \) 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; \( K \) and \( k \) are turbulent viscosity coefficients in vertical and horizontal directions; \( P_A \) is atmospheric pressure; \( W \) is wind speed; \( \rho \) is water density; \( f = 2\omega \sin \phi \) is Coriolis parameter; \( U \) and \( V \)
are total flows (unit discharge); $\omega_2$ 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) Parallelizing 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.

![Labour costs distribution](image)

**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

<table>
<thead>
<tr>
<th>Source program</th>
<th>TimeStep</th>
<th>GetPStar</th>
<th>GetQNew</th>
<th>QStar</th>
<th>PNew</th>
<th>DryCells</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>644</td>
<td>167</td>
<td>180</td>
<td>85</td>
<td>78</td>
<td>111</td>
</tr>
<tr>
<td>Optimized program</td>
<td>501</td>
<td>144</td>
<td>149</td>
<td>66</td>
<td>58</td>
<td>61</td>
</tr>
<tr>
<td>Speed-up</td>
<td>1,28</td>
<td>1,16</td>
<td>1,2</td>
<td>1,28</td>
<td>1,34</td>
<td>1,81</td>
</tr>
</tbody>
</table>

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.

![Diagram of modules dependencies](image)

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

<table>
<thead>
<tr>
<th>Source loop</th>
<th>Parallel loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i;</td>
<td>int i;</td>
</tr>
<tr>
<td>i = g.Lbe + 1;</td>
<td>i = g.Lbe + 1;</td>
</tr>
<tr>
<td>while (i &lt;= g.Len - 1) {</td>
<td>int startI = (g.Lbe + 1) / 2;</td>
</tr>
<tr>
<td>int j = g.Nbe;</td>
<td>int finishI = (g.Len - 1) / 2 + 1;</td>
</tr>
<tr>
<td>...</td>
<td>Parallel.For(startI, finishI, index =&gt; {</td>
</tr>
<tr>
<td>while (j &lt;= g.Nen) {</td>
<td>int i = index * 2;</td>
</tr>
<tr>
<td>...</td>
<td>int j = g.Nbe;</td>
</tr>
<tr>
<td>}</td>
<td>...</td>
</tr>
<tr>
<td>}</td>
<td>while (j &lt;= g.Nen) {</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
| } | }

Table 4

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

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

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](image)

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

References


Received May 8, 2016
УСКОРЕНИЕ МОДЕЛЕЙ ДИНАМИКИ ВОДНЫХ МАСС БАЛТИЙСКОГО МОРЯ

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

Описываются 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 механизм перехода к модернизации программы, включающей построение графика зависимости модулей и запись каждого модуля в отдельном порядке. Для достижения желаемого ускорения работы программы используется теория оптимизирующих и распараллеливающих преобразований. Оптимизация и распараллеливание программы гарантирует достижение уменьшения производительности при заданном объеме работы. Представлен ряд преобразований программ с полученными результатами по улучшению 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

12. Корж, А.А. Результаты масштабирования бемка NPВ UA на тысячи ядер супер-
компьютера Blue Gene/P с помощью расширения OpenMP / А.А. Корж // Вычисли-
13. Лихоед, Н.А. Обобщенный тайлинг / Н.А. Лихоед // Доклады НАН Беларуси. –
15. Gustavson, F.G. Cache Blocking for Linear Algebra Algorithms / F.G Gustavson,
R. Wyrzykowski // Parallel Processing and Applied Mathematics 2011, Part I, Lectures
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.
18. Mycroft, A. Programming Language Design and Analysis Motivated by Hardware Evolution
(Invited Presentation) / A. Mycroft // The 14th International Static Analysis Symposium.
19. Галушкин, А.И. Стратегия развития современных суперкомпьютеров на пути к экзакто-
20. Арьков, С.Б. Система асинхронного параллельного программирования «Аспект» / С.Б.
Арьков, В.Э. Мальшкин // Вычислительные методы и программирование. –
22. Ryder, B.G. Constructing the Call Graph of a Program / B.G. Ryder // Software
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 вы-
копроизводительных вычислений, Санкт-Петербургский национальный исследователь-
ский университет информационных технологий, механики и оптики, институт науково-
емких технологий (г. Санкт-Петербург, Российская Федерация), bukhanovsky@mail.ifino.ru.

Борис Яковлевич Штейнберг, доктор технических наук, профессор, Южный федераль-
ный 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.