Massively parallel Poisson Equation Solver for hybrid Intel Xeon – Xeon Phi HPC Systems

Peicho Petkov[a], Damyan Grancharov[a*], Stoyan Markov[a], Georgi Georgiev[a], Elena Lilkova[a], Nevena Ilieva[a], Leandar Litov[a]

[a] *National Centre for Supercomputing Application, Akad. G. Bonchev Str., 25A, Sofia 1113, Bulgaria*

Abstract

We have optimized an implementation of a massively parallel algorithm for solving the Poisson equation using a 27-stencil discretization scheme based on the stabilized biconjugate gradient method. The code is written in the C programming language with OpenMP parallelization. The main objective of this whitepaper lies in the optimization of the code for native Intel Xeon Phi execution, where we observe nearly linear scalability on the MIC architecture for the bigger problem sizes.

Introduction

The treatment of electrostatics is crucial for the modelling of biologically important interactions at atomistic and molecular level by means of molecular dynamics simulations. We investigate the possibility of doing this by discretely solving the Poisson equation.

The divergence of the electrostatic field in vacuum is specified by the differential form of Gauss' law

$$\nabla.\vec{\mathrm{E}}(\vec{r}) = \frac{\rho(\vec{r})}{\epsilon} \quad , \tag{1}$$

where $\rho(\vec{r})$ is the charge density and ϵ is the dielectric constant of vacuum.
The curl of the electrostatic field is specified by the static form of Faraday's law

$$\nabla \times \vec{E}(\vec{r}) = 0 \quad . \tag{2}$$

By Helmholtz' theorem, these two first-order vector differential relations completely determine the electrostatic field vector $\vec{E}(\vec{r})$ in any specified region of space given as a consequence of (2). The electrostatic field may be expressed as

$$\vec{E} = -\nabla\phi(\vec{r}), \tag{3}$$

where $\varphi(\vec{r})$ is the scalar potential.
Substitution of (3) into Gauss' Law (1) yields Poisson's equation

$$\nabla^2\phi = -\frac{\rho}{\epsilon} \; or \; (\phi_{xx} + \phi_{yy} + \phi_{zz}) = -\rho/\epsilon. \tag{4}$$

* Corresponding author. *E-mail address*: dgrancharov@phys.uni-sofia.bg.

In a discretized form this can be written as

$$(\phi_{xx})_{ijk} + (\phi_{yy})_{ijk} + (\phi_{zz})_{ijk} = -f_{ijk}. \quad (5)$$

Let us denote (i, j, k) three-dimensional lattice indices and $(\phi_{\alpha\alpha})_{ijk}$ is an approximation to the second partial derivative with respect to the coordinate direction $\alpha\alpha$.
Denote second-order difference operator as

$$(\phi_{\alpha\alpha})_{ijk} = \frac{1}{h_\alpha{}^2}\delta_\alpha^2\phi_{ijk}, \quad (6)$$

where h_α is the grid spacing in direction α and δ_α^2 is the second-order difference operator

$$\begin{aligned}\delta_x^2 &= \phi_{i-1,j,k} - 2\phi_{i,j,k} + \phi_{i+1,j,k},\\ \delta_y^2 &= \phi_{i,j-1,k} - 2\phi_{i,j,k} + \phi_{i,j+1,k},\\ \delta_z^2 &= \phi_{i,j,k-1} - 2\phi_{i,j,k} + \phi_{i,j,k+1}.\end{aligned}$$

Standard and high-order finite difference operators of the Poisson's equation are derived from the approximation

$$\frac{\partial^2\phi}{\partial\alpha^2}|_{\alpha=ih_\alpha} = \frac{4}{h_\alpha{}^2}\left(\sinh^{-1}\frac{\delta_\alpha}{2}\right)^2 = \frac{1}{h_\alpha{}^2}\delta_\alpha^2\left\{1 - \frac{1}{12}\delta_\alpha^2 + \frac{1}{90}\delta_\alpha^4 - \frac{1}{560}\delta_\alpha^6 \pm \ \ldots\right\}\phi_{ijk}, \quad (7)$$

where α is x, y or z.To solve the three dimensional Poisson's equation in Cartesian coordinate systems using finite difference approximations; see e.g. [1]

We have developed an approximation using central difference scheme to obtain a 19-point stencil and a 27-point stencil with some modification in the right hand side terms.
The grid size along the x, y and z directions is *h=0.5Å(Angstrom)*; *i= 1,2,3,…..m*; *j=1,2,3…n*; k= *1,2,3,….p*; *m = x/h*; *n = y/h*; *p = z/h*; $\rho = \frac{q}{h^3}$, where q is the point electric charge.
In the 27-point stencil finite difference approximation, for the unknowns *φ(i,j,k)* the following equation is obtained, when the finite difference operator from (6) is used:

$$\begin{aligned}144h^2\rho \ = \ -600\varphi_{i,j,k} \ + 60\left[\varphi_{i,j,k-1} + \varphi_{i,j,k+1}\right] \ + \ 60\left[\varphi_{i-1,j,k} + \ \varphi_{i+1,j,k} + \ \varphi_{i,j-1,k} + \ \varphi_{i,j+1,k}\right] \ + \\ 18\left[\varphi_{i-1,j-1,k} + \ \varphi_{i-1,j+1,k} + \varphi_{i+1,j-1,k} \ + \ \varphi_{i+1,j+1,k}\right] + 18\left[\varphi_{i-1,j,k-1} + \ \varphi_{i+1,j,k-1} + \ \varphi_{i-1,j,k+1} \ + \right. \\ \left. \varphi_{i+1,j,k+1} + \ \varphi_{i,j-1,k-1} + \ \varphi_{i,j+1,k-1} + \ \varphi_{i,j-1,k+1} + \ \ \varphi_{i,j+1,k+1}\right] \ + \ 3\left[\varphi_{i+1,j-1,k-1} + \ \varphi_{i-1,j-1,k-1} + \right. \\ \left. \varphi_{i+1,j+1,k-1} + \ \varphi_{i+1,j+1,k+1} + \ \varphi_{i-1,j-1,k+1} \ + \ \varphi_{i+1,j-1,k+1} \ + \ \varphi_{i-1,j+1,k-1} + \ \varphi_{i-1,j+1,k+1}\right] \quad (8)\end{aligned}$$

The maximum absolute error of this approximation for $\varphi(\vec{r})$ is $O(h^{-6})$.

Taken in the X-, Y- and Z-direction from eq. (8) we get a large system of linear equations (the number of equations actually depends on the size of the lattice). This system of equations can be written in a matrix form as:

$$Au(s) = \ b(s), \quad (9)$$

where

$$A = \begin{pmatrix} R & S & & & & & & \\ S & R & S & & & & & \\ & S & R & S & & & & \\ & & & & \ddots & & & \\ & & & & & S & R & S \\ & & & & & & S & R \end{pmatrix}$$

$$R = \begin{pmatrix} R_1 & R_2 & & & & & & \\ R_2 & R_1 & R_2 & & & & & \\ & R_2 & R_1 & R_2 & & & & \\ & & & & \ddots & & & \\ & & & & & R_2 & R_1 & R_2 \\ & & & & & & R_2 & R_1 \end{pmatrix}$$

$$S = \begin{pmatrix} S_1 & S_2 & & & & & & \\ S_2 & S_1 & S_2 & & & & & \\ & S_2 & S_1 & S_2 & & & & \\ & & & & \ddots & & & \\ & & & & & S_2 & S_1 & S_2 \\ & & & & & & S_2 & S_1 \end{pmatrix}$$

Matrix *A* has *p* blocks and each block is of order *mn x mn*. *R* and *S* have *n* blocks and each block is of order *m x m*.

$$R_1 = \begin{pmatrix} -600 & 60 & & & & & & \\ 60 & -600 & 60 & & & & & \\ & 60 & -600 & 60 & & & & \\ & & & & \ddots & & & \\ & & & & & 60 & -600 & 60 \\ & & & & & & 60 & -600 \end{pmatrix}$$

$$R_2 = \begin{pmatrix} 60 & 18 & & & & & & \\ 18 & 60 & 18 & & & & & \\ & 18 & 60 & 18 & & & & \\ & & & & \ddots & & & \\ & & & & & 18 & 60 & 18 \\ & & & & & & 18 & 60 \end{pmatrix}$$

$$S_1 = \begin{pmatrix} 60 & 18 & & & & & & \\ 18 & 60 & 18 & & & & & \\ & 18 & 60 & 18 & & & & \\ & & & & \ddots & & & \\ & & & & & 18 & 60 & 18 \\ & & & & & & 18 & 60 \end{pmatrix}$$

$$S_2 = \begin{pmatrix} 18 & 3 & & & & & & \\ 3 & 18 & 3 & & & & & \\ & 3 & 18 & 3 & & & & \\ & & & & \ddots & & & \\ & & & & & 3 & 18 & 3 \\ & & & & & & 3 & 18 \end{pmatrix}$$

$$u = \begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ \dots \\ U_p \end{pmatrix} \quad b = \begin{pmatrix} B_1 \\ B_2 \\ B_3 \\ \dots \\ B_p \end{pmatrix}$$

$$U_k = (u_{1,1,k}, u_{2,1,k}, \dots, u_{m,1,k}; u_{1,2,k}, u_{2,2,k} .. u_{m,2,k} \dots u_{1,n,k}, u_{2,n,k}, u_{m,n,k})$$

$$B_k = (b_{1,1,k}, b_{2,1,k}, \dots, b_{m,1,k}; b_{1,2,k}, b_{2,2,k}, \dots, b_{m,2,k} \dots . b_{1,n,k}, b_{2,n,k}, b_{m,n,k});$$

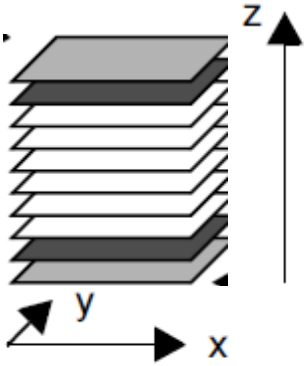
If in the point with lattice indices *t, s, v* the charge is *q*, then the charge density is $b_{t,s,v} = \rho = q/h^3$.

Methods

Biconjugate gradient stabilized method

We suggest a new parallel BiCGSTAB algorithm to find the solution of the aforementioned system of linear algebraic equations. A classical BiCGSTAB algorithm is described in [2].
The multiplications of the matrix *A* with the vectors of the solution φ and the search directions *s* are the most time consuming part of the algorithm. The number of arithmetic operations "multiplication and addition" is *27mnp* for each of the two multiplications.

Parallel BiCGSTAB solver [3]
We assume that the number of Intel Xeon Phi coprocessors is at least equal to the number of rows of matrix *A*, which is *p*. Then, the solver construction is as follows:



Each slice is calculated on one Xeon Phi. Below is given the description of the parallel algorithm:

- 1. The row A_k of matrix *A* and block b_k of vector *b* as well as the initial value of blocks x_{k-1}, x_k, x_{k+1} of vector *x* are sent to the *k-th* Xeon/Xeon Phi node.
- 2. All Xeon Phi's simultaneously calculate $\delta_{0k}=S*x_{k-1}+R*x_k+S*x_{k+1}$, by parallel treatment of *n* threads, each with length *m*. The maximum number of threads depends on the coprocessor type.
- 3. $r_{0k} = b_k - \delta_{0k}$, $r_k = r_{0k}$, $p_k = r_k$ and the constant $\rho_k = (r_k, r_{0k})$ are obtained.
- 4. Each Phi sends the vectors r_{0k}, r_k and the constant ρ_k to the corresponding Xeon. Afterwards the Xeons communicate and the vectors $r = r_0$ and $p=r$ and the constant $\rho = \sum \rho_k$ are obtained by collective communication.
- 5. The Xeons send the blocks p_{k-1}, p_k, p_{k+1} of vector *p* and r_{k-1}, r_k, r_{k+1} of vector *r* to the Xeon Phi's. *k*=1,2….*p*.
- 6. All Xeon Phi's simultaneously calculate the blocks $u_k=S*p_{k-1}+R*p_k+S*p_{k+1}$ by parallel treatment of *n* threads, each of length *m*. The constant $\gamma_k = (r_{0k}, u_k)$ is calculated.
- 7. The Xeon (master) calculates the constant $\alpha = \rho/\sum \gamma_k$ and sends it to other Xeon's and then they send it to Xeon Phi's. The latter calculate $s_k = r_k - \alpha u_k$.
- 8. s_k are sent to the Xeon (master), which constructs the vector *s* and sends the triplet s_{k-1}, s_k, s_{k+1} to the *k*-th Xeon and then they send it to Xeon Phi's. *k*=1, 2….*p*.
- 9. The same way as in step 5, $t_k = S*s_{k-1} + R*s_k + S*s_{k+1}$. The constants $\tau_k=(t_k, t_k)$ and $\sigma_k = (t_k, s_k)$ are sent to the Xeon (master), which calculates $\tau = \sum \tau_k$, $\sigma = \sum \sigma_k$ and $\omega = \sigma/\tau$.
- 10. The *k*-th Xeon Phi calculates the new x_k, r_k.Then x_k, r_k and u_k are sent to the Xeons and then to the Xeon (master).
- 11. The constants β and ρ and then the new vector *p* are calculated.
- 12. If the error *abs(r)* is greater than ε, than the cycle repeats from point 3. Else
- 13. STOP

If the number of coprocessors is at least equal to p, and the number of threads is $\leq n$, then the number of arithmetic operations is proportional to $54m$. Hence, the acceleration of the Parallel BiCGSTAB solver in comparison to the classical BiCGSTAB method is proportional to pn. If the number of Xeon Phi cards is equal to K, and the threads $n > 240$, then the acceleration is proportional to $pn/(Res+1)(L+1))$, where $Res = int(p/K)$ and $L = int(n/240)$.

Work done and results

Implementation of the BiCGSTAB solver for native execution on Xeon Phi coprocessors

We have developed a parallel implementation of the algorithm described in section "Methods" in the C programming language with the OpenMP parallelization model for Xeon Phi systems. Here we present characteristics and results with the code written for native execution on the Xeon Phi.
All necessary vectors are allocated dynamically as 3-dimensional arrays with sizes equal to the dimensions of the lattice – charges, residuals, search directions and their conjugates, as well as the potentials. Since we work with 3-dimensional arrays and use direct indexation, it is easy to account for boundary elements that do not have full amount of neighbours. The matrix of the 27-stencil approximation is represented only by the coefficients in eq. 8.
The calculations in the current implementation are dominated by a matrix vector multiplication that appears two times per iteration. The other calculations consist of four vector dot multiplications and some multiplications of vectors with scalars. Therefore, it is decisive to have a scalable and efficient matrix vector multiplication code for Xeon Phi. The matrix vector multiplication is done in a subroutine that takes the vector as an argument without storing the matrix explicitly. It should be emphasized, that there are no loops over the matrix, only the coefficients in eq. 8 are used.
Our general idea for the parallelization of the code consists of splitting the 3-dimensional lattice in slices along the z axis. Then we could divide the work between multiple Xeon Phi coprocessors by giving a certain amount of slices to each of them. Further calculations are divided between threads in the current slice. We have written and tested an MPI communication scheme for halo exchange previously and the combination of these two seems a promising alternative. However, here we focus on the optimization of the Xeon Phi part of the code.
The code consists of around 760 lines and contains 6 functions. Finally, the potentials are returned to the main function after reaching convergence with a threshold of 10^{-4} or user specified. In this way the code may also be used as a library, attached to another application.

Optimization for MIC architecture

The most important characteristics of an application in order to perform well on the Xeon Phi are the vectorization and scaling capabilities of the code [4]. The vectorization of loops is embedded in the compiler and is done automatically with a certain level of optimization. But helping the compiler to understand which loops are vectorizable is far from trivial. We have invested a lot of effort in vectorization of the most time-consuming loops in the code. The main instruments turned out to be the well-structured data and some pragma directives. Loops that contain several statements in their body or statements that are not dependent only on indexes and constants are less likely to be classified as vectorizable by the compiler. It is also not recommended to call functions inside the body of the loop because this confuses the compiler as well. In this context we have defined and allocated a supplementary array that we used only to supply the loop with better structured data. It was a key to vectorization.
Last but not least, it is not recommended to try to vectorize an OpenMP parallel for-loop. In our implementation we have discovered that a combination of an outer openmp parallel loop and vectorized inner loop works best.
In addition, we encountered problems with data allocation and memset operations. As written in [4], the code runs better if the data is aligned at 64 bytes, according to the MIC architecture, which can be achieved by the _mm_malloc call. Nevertheless, our observation is that data allocation is relatively slow. It is the same situation with the memset call: it is slow and scales poorly.

Results

We have tested our code on different problem sizes on various amounts of threads on Intel Xeon Phi cluster Eurora at Cineca. It is important to emphasize that for small-scale problems - 30x30x30 and 60x60x60 grid points, the speedup curve flattens, because of thread saturation – a sign of an under-load situation. For larger-scale problems - 120x120x120 and 240x240x240 grid points, we observe good scaling. The time per iteration drops with the increase of the total amount of threads and speedup is nearly linear as shown in fig. 1 and fig. 2.

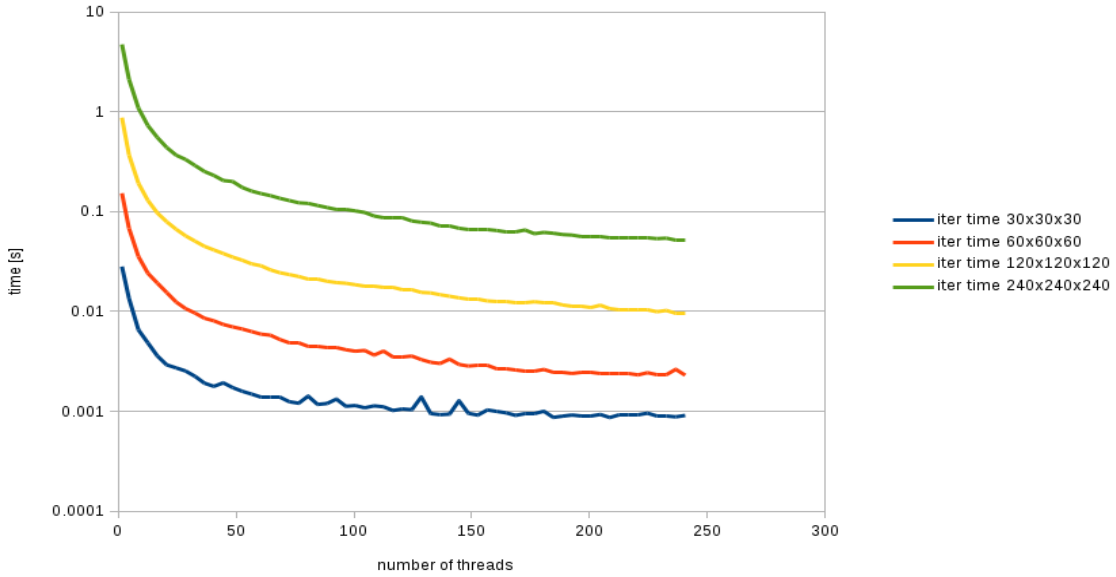


Figure 2. Execution time as a function of the number of the Intel Xeon Phi threads for one iteration for different problem size (dimensions of the grid) 30x30x30 (blue line), 60x60x60 (red line), 120x120x120(yellow line) and 240x240x240 (green line).

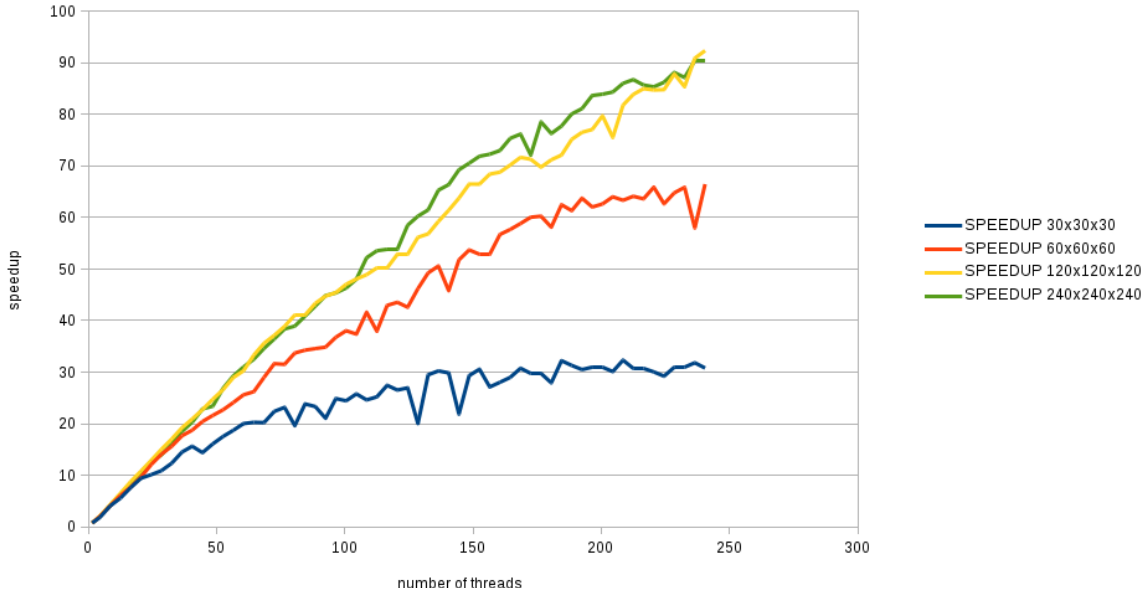


Figure 1. Speedup as a function of the number of the Intel Xeon Phi threads for one iteration for different problem size (dimensions of the grid) 30x30x30 (blue line), 60x60x60 (red line), 120x120x120(yellow line) and 240x240x240 (green line).

The same code was compiled also for native Intel Xeon execution and results show very good scaling up to 16 threads. Still, the performance of the code is 7 to 10 times better on the Xeon, than on the Xeon Phi. A possible explanation of this behaviour might be the slightly lower L1 cache bandwidth, as well as the less bandwidth per peak floating point operations for the Xeon Phi compared to the Xeon. The Xeon Phi has no L3 cache, so the total amount of cache is lower and since the most time consuming subroutine uses elements of the arrays, that are not only adjacent or close to each other, there is a lot of data transfer and possibly increased amount of cache misses.

Conclusions

In conclusion, we would like to stress out that vectorization is crucial to performance on the Xeon Phi coprocessor but it is not a free gift. One must pay attention to the way one stores and accesses data. Often it is necessary to allocate extra variables only to maintain clean data access. The vectorization can be enforced by

explicit directives in the code, but one should be careful, because it might change the meaning of the code and affect the results, as advised in [4].
In addition, we would like to emphasize that scaling is only one half of the goal and it comes second after performance. We have achieved good scaling, following all the MIC programming recommendations, yet our code still lacks performance, when compared to the more standard Xeon architecture.
The library source can be found at http://phys.uni-sofia.bg/~peicho/poisson-solver-mic-1.0.tgz .

References

[1] W. F. Spotz and G. F. Carey, Formulation for the 3D Poisson Equation," Numerical Methods for Partial Differential Equations, Vol. 12, No. 2, 1996, pp. 235-243
[2] Laurence Tianruo Yangyz, Richard P. Brent, Improved BiCGStab Method for Large and Sparse Unsymmetric Linear Systems on Parallel Distributed Memory Architectures (http://maths-people.anu.edu.au/~brent/pd/rpb206.pdf)
[3] Van der Vorst, H. A. (1992). "Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems". SIAM J. Sci. and Stat. Comput. 13 (2): 631–644.
[4] Intel Xeon Phi Coprocessor High-Performance Programming, Jim Jeffers, James Reinders; Morgan Kaufmann (an Elsevier imprint), 2013

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-261557. The project was realized using the PRACE Research Infrastructure resources at Cineca, Italy.