Proceedings of the World Congress on Engineering 2013 Vol II,
WCE 2013, July 3 - 5, 2013, London, U.K.

Parallel Iterative Solution of the Hermite Collocation Equations on GPUs

E. Mathioudakis, N. Vilanakis, E. Papadopoulou and Y. Saridakis

Abstract—**We consider the computationally intense problem of solving the large, sparse and non-symmetric system of equations arising from the discretization of elliptic Boundary Value Problems (BVPs) by the Collocation finite element method using Hermite bi-cubic elements. As the size of the problem directly suggests the usage of parallel iterative methods, we consider the implementation on multiprocessor shared memory parallel architectures with Graphics Processing Units of the non-stationary preconditioned Bi-Conjugate Gradient Stabilized (BiCGSTAB) iterative method. To induce scalability to our computation, we structure the Collocation matrix to a particular line red-black ordered form, leading to the development of a well-structured parallel algorithm for the iterative method. The realization of the said algorithm took place on a HP SL390 multiprocessor machine with Tesla M2070 GPUs. Execution time measurements are used to reveal the efficiency of our parallel implementation.**

Index Terms—**Collocation, BiCGSTAB, Shared Memory, OpenMP, OpenACC, GPUs.**

I. Introduction

COLLOCATION method is a high order accurate discretizer for BVPs modelling applications in several fields of science and engineering (e.g. [1]). The method approximates the solution of the problem avoiding numerical integration and making readily available the values of the solution function and its first derivatives at all grid nodes. Thus the resulting linear system is large and sparse, suggesting the usage of efficient iterative solvers [2], [3], [4]. For realistic applications, where fine discretizations are necessary, the realization of the method requires high performance computing architectures. The resources for these computing environments can include multi-core machines with Graphic Processing Units (GPUs), which can accelerate the performance. To take advantage of the increased computing power capabilities GPUs induce to computers, an efficient realization of a parallel algorithm for shared memory and massively parallel architectures is needed.

These scientific issues have attracted the interest of several researchers in the past and an important progress has been made in the area of parallel iterative solution of the collocation finite element method (e.g. [5], [6], [7], [8], [9], [10]).

The aim of this work is the development of appropriate parallel algorithms for the numerical treatment of the collocation equations on GPU computational environments. The paper is organized as follows: In Section II, the iterative solution for collocation linear system of algebraic equations arising from the application of finite element method based on a Hermite bi-cubic elements is briefly described. Section III presents the basic features for developing a parallel algorithm for multiprocessor with GPUs machines for the BiCGSTAB iterative solving procedure. Finally, in Section IV, we present the numerical results from the performance evaluation of the parallel algorithm.

Manuscript received March 23, 2013; revised April 16, 2013.

This work was supported by EU (European Social Fund ESF) and Greek funds through the operational program *Education and Lifelong Learning* of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALIS.

All authors are with the Department of Sciences, Technical University of Crete, University Campus, 73132 Chania, Crete, Greece – Corresponding author's e-mail: manolis@science.tuc.gr.

II. Red Black Collocation linear systems

Let us consider the modified Helmholtz problem

$$\begin{cases} \nabla^2 u(x,y) - \lambda u(x,y) &= f(x,y) \ , \ (x,y) \in \Omega \\ u(x,y) &= g(x,y) \ , \ (x,y) \in \partial\Omega \end{cases} \tag{1}$$

with $\lambda \geq 0$ on the rectangular domain $\Omega \equiv (0,1) \times (0,1)$ as our model problem. Assuming a uniform partition of the intervals $I^x = I^y = [0,1]$ into n_s subintervals $I^x_m = I^y_m \ , \ m = 1 \ , \ \dots \ , \ n_s$ which generates a uniform grid with spacing $h = \frac{1}{n_s}$ and nodal coordinates (x_i, y_j), where $x_i = (i-1)h$ and $y_j = (j-1)h$, $i, j = 1 \ , \ \dots \ , \ (n_s + 1)$. The Hermite Bi-Cubic finite element approximation seeks an approximate solution $\tilde{u}(x,y)$ in the form

$$u(x,y) \ \sim \ \tilde{u}(x,y) \ = \ \sum_{i=1}^{\tilde{n}} \sum_{j=1}^{\tilde{n}} \alpha_{i,j} \phi_i(x) \phi_j(y) \ , \tag{2}$$

where $\tilde{n} = 2(n_s + 1)$. The *basis functions* $\phi_i(x)$ and $\phi_j(y)$ are the known one dimensional piecewise Hermite cubic polynomials [11]. Based, now, on the basic properties of Hermite basis functions, one can easily verify that the following four unknowns

$$\begin{cases} a_{2i-1,2j-1} &= \tilde{u}(x_i, y_j) \\ a_{2i-1,2j} &= h \frac{\partial}{\partial y} \tilde{u}(x_i, y_j) \\ a_{2i,2j-1} &= h \frac{\partial}{\partial x} \tilde{u}(x_i, y_j) \\ a_{2i,2j} &= h^2 \frac{\partial^2}{\partial x \partial y} \tilde{u}(x_i, y_j) \end{cases} \tag{3}$$

are associated with the mesh point (x_i, y_i). With the imposition of boundary conditions $8n_s + 4$ unknowns, associated with nodes on the boundary $\partial\Omega$, can be determined beforehand. Therefore collocation equations needed for the determination of the remaining $n = 4n_s^2$ unknowns are then constructed by forcing the approximate solution $\tilde{u}(x,y)$ to satisfy the BVP in n interior collocation points. These are the four Gaussian points in each of the n_s^2 elements I_{ij}. Since there is an one-to-one correspondence between collocation points and equations, a numbering of the equations is produced when we number the collocation points while a numbering of the unknowns is readily available when we number the unknowns associated with each node. This procedure results to a linear system

$$A\boldsymbol{x} \ = \ \boldsymbol{b} \ , \tag{4}$$

ISBN: 978-988-19252-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

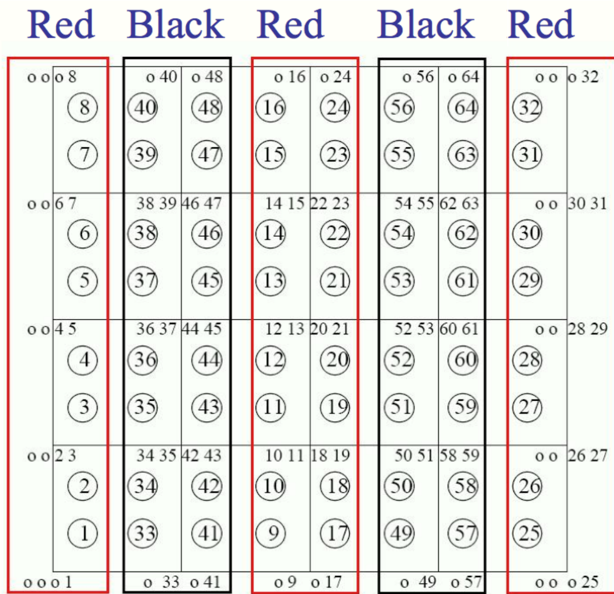


Fig. 1. Numbering of unknowns and equations for $n_s = 4$.

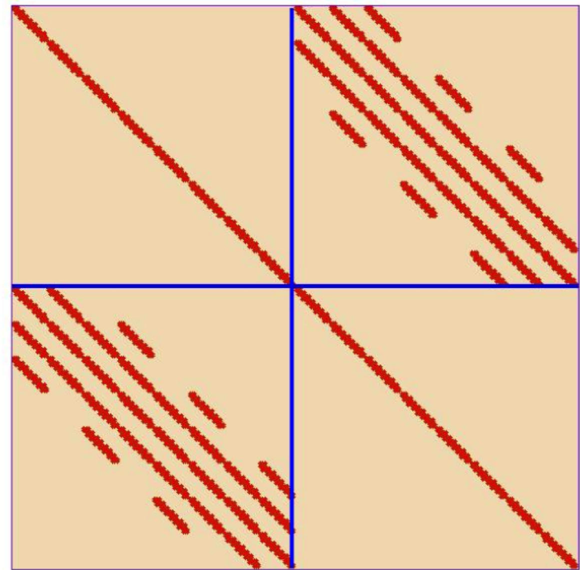


Fig. 2. Structure of the Collocation Matrix for $n_s = 4$.

where A is the $n \times n$ *Collocation coefficient matrix* and

$$\boldsymbol{x} = [x_1 \ x_2 \ \cdots \ x_n]^T \equiv [\alpha_{1,1} \ \cdots \ \alpha_{\tilde{n},\tilde{n}}]^T$$

is the unknown vector, with $n = 4n_s^2$.

To increase parallelism, we number unknowns and equations in red-black ordered fashion (cf. [7]) depicted in Fig. 1 for $n_s = 4$. Small numbers represent the numbering of unknowns per node while circled numbers represent the numbering of the equations per element. Small circles on the boundary indicate unknowns that have been evaluated from the boundary conditions. In Fig. 2 the structure of the resulted collocation matrix after the application of a similarity transformation as in [7], [8] is shown schematically. One can easily verify that the collocation matrix takes its 2-cyclic normal form

$$A = \begin{bmatrix} D_R & H_B \\ H_R & D_B \end{bmatrix}, \tag{5}$$

where D_R and D_B are non-singular block diagonal matrices. For the case of $n_s = 2p$ they have the form

$$D_R = \text{diag}[\underbrace{A_2 \ 2A_1 \ 2A_2 \ \cdots \ 2A_1 \ 2A_2 \ -A_2}_{2p-blocks}], \tag{6}$$

$$D_B = 2\,\text{diag}[\underbrace{A_1 \ A_2 \ \cdots \ A_1 \ A_2}_{2p-blocks}] \tag{7}$$

$$H_R = \begin{bmatrix} R_1 & R_2 & & & & \\ R_3 & R_1 & R_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & R_3 & R_1 & R_2 \\ & & & & R_3 & \hat{R}_1 \end{bmatrix} \tag{8}$$

$$H_B = \begin{bmatrix} B_1 & B_2 & & & & \\ B_3 & B_1 & B_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & B_3 & B_1 & B_2 \\ & & & & B_3 & B_1 \end{bmatrix} \tag{9}$$

where

$$R_1 = \begin{bmatrix} A_4 & A_3 \\ -A_4 & A_3 \end{bmatrix}, \ \hat{R}_1 = \begin{bmatrix} A_4 & -A_4 \\ -A_4 & -A_4 \end{bmatrix},$$

$$R_2 = -\begin{bmatrix} A_4 & 0 \\ A_4 & 0 \end{bmatrix}, \ R_3 = \begin{bmatrix} 0 & A_3 \\ 0 & -A_3 \end{bmatrix},$$

and

$$B_1 = \begin{bmatrix} A_3 & -A_4 \\ A_3 & A_4 \end{bmatrix},$$

$$B_2 = \begin{bmatrix} 0 & 0 \\ A_3 & -A_4 \end{bmatrix}, \ B_3 = -\begin{bmatrix} A_3 & A_4 \\ 0 & 0 \end{bmatrix}.$$

The block form of the above matrices involve four $2n_s \times 2n_s$ pentadiagonal basic real matrices A_i for $i = 1, \ldots, 4$ [10].

In our earlier work [8], [9], [12], [10] we have successfully solved the red-black collocation linear system using the SOR and preconditioned Krylov subspace iterative methods based in the following splitting of the matrix

$$A = D_A - L_A - U_A, \tag{10}$$

where

$$D_A = \begin{bmatrix} D_R & O \\ O & D_B \end{bmatrix}, \ L_A = \begin{bmatrix} O & O \\ -H_R & O \end{bmatrix},$$

$$U_A = \begin{bmatrix} O & -H_B \\ O & O \end{bmatrix}, \tag{11}$$

and for the other members of the collocation linear system we have assumed the conformal partitioning of the vectors $\boldsymbol{x}$ and $\boldsymbol{b}$ into

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_R \\ \boldsymbol{x}_B \end{bmatrix} \text{ and } \boldsymbol{b} = \begin{bmatrix} \boldsymbol{b}_R \\ \boldsymbol{b}_B \end{bmatrix}. \tag{12}$$

In these works, as well as in [6], the BiCGSTAB iterative method, preconditioned by either the Symmetric Gauss-Seidel (SGS) or the Gauss-Seidel (GS) schemes, proved to

ISBN: 978-988-19252-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

converge faster than SOR and any other Krylov subspace method.

To increase the scalability, as well as to reduce the execution time, of the preconditioned BiCGSTAB method, we introduce the following two sided preconditioning

$$M_1^{-1}\, A\, M_2^{-1}\, M_2 \boldsymbol{x} \;=\; M_1^{-1} \boldsymbol{b} \;\;, \tag{13}$$

where M_1 is the Gauss Seidel's iteration matrix based on the matrix splitting in (10)

$$M_1 \;=\; D_A - L_A \;=\; D_A(I - D_A^{-1} L_A) \tag{14}$$

and

$$M_2 \;=\; I - D_A^{-1} U_A \;\;. \tag{15}$$

The collocation linear system takes the following form

$$\begin{bmatrix} I & O \\ O & S \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_R + D_R^{-1} H_R \boldsymbol{x}_B \\ \boldsymbol{x}_B \end{bmatrix} = \begin{bmatrix} \hat{\boldsymbol{b}}_R \\ \hat{\boldsymbol{b}}_B \end{bmatrix} \;, \tag{16}$$

where

$$S = D_B - H_R D_R^{-1} H_B \tag{17}$$

is the Schur complement of the collocation matrix with respect of D_B and

$$\hat{\boldsymbol{b}}_R \;=\; D_R^{-1} \boldsymbol{b}_R \quad \text{and} \quad \hat{\boldsymbol{b}}_B \;=\; \boldsymbol{b}_B - H_R \hat{\boldsymbol{b}}_R \;\;. \tag{18}$$

The detailed computation involved in the equations above can be described by means of the following algorithm :

Algorithm for Schur complement collocation equations

S1: Solve $D_R \hat{\boldsymbol{b}}_R = \boldsymbol{b}_R$
S2: Evaluate $\hat{\boldsymbol{b}}_B \;=\; \boldsymbol{b}_B - H_R \hat{\boldsymbol{b}}_R$
S3: Solve with BiCGSTAB $S\, \boldsymbol{x}_B \;=\; \hat{\boldsymbol{b}}_B$
S4: Evaluate $\hat{\boldsymbol{x}}_B \;=\; H_B \boldsymbol{x}_B$
S5: Solve $D_R \hat{\boldsymbol{x}}_R = \hat{\boldsymbol{x}}_B$
S6: Evaluate $\boldsymbol{x}_R \;=\; \hat{\boldsymbol{b}}_R - \hat{\boldsymbol{x}}_R$

Obviously, the dominant operations of the algorithm are the matrix vector multiplication with the block diagonal matrices D_R, D_B, H_B and H_R, followed by the direct solution of linear systems involving matrices D_R and D_B. Said operations, in these *red* and *black* computation cycles, require the design of efficient algorithms for shared memory architectures.

III. Parallel Algorithm for Collocation

The architecture of the parallel system available, along with the number of processors, are the most important factors affecting the ordering and the partition of the whole computation of a parallel algorithm. Our model is a shared memory system, consisting of a few powerful processor cores for the host computer and a few hundreds for the graphics processing unit. GPU cores can perform only basic arithmetic operations and they have their own memory. As such, data and computation partitioning must take into consideration the particular architecture at hand and, at the same time, keep all cores busy during the whole computation. This *no idle core* model requires well balanced computational and memory communication loads. An efficient way to carry out the demanding task of assigning threads to cores is by considering at first a virtual architecture with unlimited number of cores. If now one takes also into consideration the requirement that these threads must be data independent with minimized memory communication, and that the number of subintervals $n_s \;=\; 2p$ of the discretization in both x and y directions is even, it can be easily observed that the appropriate allocation scheme requires the assignment of one core thread for each one of the $2p+1$ vertical grid lines. This is shown schematically in Fig.3 for $n_s = 4$.

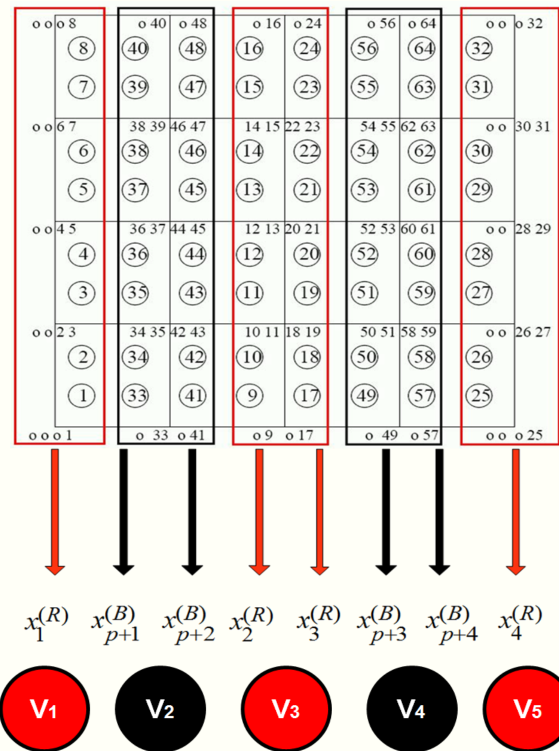


Fig. 3. Assigning collocation unknowns into threads for $n_s = 4$.

Moreover, *odd* core threads represent *red* grid lines while *even* threads represent *black* grid lines and due to the $2p$-block partitioning of all vectors participating in the computation, it becomes clear that each one of the odd threads V_{2i-1} , $i = 1, \ldots, p+1$, has being assigned with the task of determining the solution subvectors $\boldsymbol{t}_{2i-2}$ and $\boldsymbol{t}_{2i-1}$ while each one of the even threads V_{2i} , $i = 1, \ldots, p$, has being assigned with the task of determining the solution subvectors $\boldsymbol{t}_{2p+2i-1}$ and $\boldsymbol{t}_{2p+2i}$. The irregularity appeared in threads V_1 and V_{2p+1} is due to the boundary conditions.

In the parallel algorithm all basic linear algebra vector operations such as inner products, vector additions and multiplications with scalars, can be executed in parallel based on the above $2p$-block partitioning of all vectors. These operations are performed in parallel since there is no data dependence from one thread to the other.

According to the above scheme of data mapping into threads, the parallel procedures have no data dependency for all matrix vector multiplications, and the direct linear system solutions for the red and black cycles of computation. The following parallel algorithms integrate all the above properties for the evaluation of the arbitrary vector $\boldsymbol{t}$ of length $4{n_s}^2$.

ISBN: 978-988-19252-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Red Computation Cycle

C$ OMP PARALLEL DO DEFAULT(SHARED)
<u>do</u> $i = 0$ <u>to</u> p
  V_{2i+1} <u>computes</u> $\boldsymbol{t}_{2i}, \ \boldsymbol{t}_{2i+1}$
<u>enddo</u>
C$ OMP END PARALLEL DO

Black Computation Cycle

C$ OMP PARALLEL DO DEFAULT(SHARED)
<u>do</u> $i = 1$ <u>to</u> p
  V_{2i} <u>computes</u> $\boldsymbol{t}_{2p+2i}, \boldsymbol{t}_{2p+2i-1}$
<u>enddo</u>
C$ OMP END PARALLEL DO

In the above algorithms, the CPU thread implementation can be realized by using efficient procedures from existing numerical libraries [13]. For example, in the case of the forward and backward substitutions, during the block direct solution of the linear systems with matrices D_R and D_B, the appropriate procedure from Lapack library is chosen, while for the matrix vector multiplication, involving the matrices H_B and H_R, the procedure from BLAS library can be used. However, if one wants to exploit the special structure of the matrices involved in the computations to optimize the GPU thread implementation, it is necessary to design efficient algorithms for the massively parallel model.

The serial nature of the computations involved during the forward and backward substitutions of linear system solutions, combined with the fact of GPU memory limitations, directly imply that the solution of the linear systems during the red and black cycles, with coefficient matrices D_R and D_B respectively, have to be performed from the computer host threads.

On the other hand, the independence of the data involved during the matrix-vector basic linear algebra operations, combined with the fact that the hundreds of GPU cores are organized in computational groups and are able to perform simultaneously - via the SIMD programming model - basic arithmetic operations extremely faster than the computer host processors, directly imply that the matrix-vector multiplication subroutines during the red and black cycles involving the H_R and H_B block matrices are suitable for GPU implementations. Further improvement is achieved by taking advantage of H_R's and H_B's block structure, from relations (8) and (9) respectively, which is based on two only pentadiagonal matrices A_3 and A_4 of order $2n_s$.

The above are implemented in the parallel algorithm that follows and describes in detail the black cycle for $\boldsymbol{t} = H_B\boldsymbol{z}$ GPU matrix vector multiplication.

Black GPU Computation Cycle

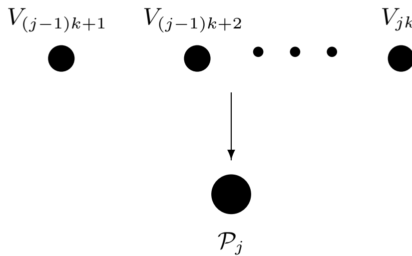
!$ACC DATA COPYIN($\boldsymbol{z}$) CREATE($\boldsymbol{temp}$) COPYOUT($\boldsymbol{t}$)
!$ACC KERNELS
!$ACC LOOP INDEPENDENT
<u>do</u> $k = 1$ <u>to</u> $n_s - 3$ <u>with step</u> 2
  $k_1 = (k-1)2n_s \ , \ k_2 = k2n_s$
  $k_3 = (k+1)2n_s \ , \ k_4 = (k+2)2n_s$
!$ACC LOOP INDEPENDENT
  <u>do</u> $i = 1$ <u>to</u> $2n_s$
    $t(k_2+i) = A_3(3,i)z(k_1+i) + A_4(3,i)z(k_2+i)$
  <u>enddo</u>
!$ACC LOOP INDEPENDENT
  <u>do</u> $i = 2$ <u>to</u> $2n_s$
    $t(k_2+i-1) = t(k_2+i-1)+$
      $A_3(2,i)z(k_1+i-1) + A_4(2,i)z(k_2+i)$
  <u>enddo</u>
!$ACC LOOP INDEPENDENT
  <u>do</u> $i = 1$ <u>to</u> $2n_s - 1$
    $t(k_2+i+1) = t(k_2+i+1)+$
      $A_3(4,i)z(k_1+i) + A_4(4,i)z(k_2+i)$
  <u>enddo</u>
!$ACC LOOP INDEPENDENT
  <u>do</u> $i = 1$ <u>to</u> $2n_s - 2$
    $t(k_2+i) = t(k_2+i)+$
      $A_3(1,i+2)z(k_1+i+2)+$
      $A_4(1,i+2)z(k_2+i+2)$
  <u>enddo</u>
!$ACC LOOP INDEPENDENT
  <u>do</u> $i = 1$ <u>to</u> $2n_s - 2$
    $t(k_2+i+2) = t(k_2+i+2)+$
      $A_3(5,i)z(k_1+i) + A_4(5,i)z(k_2+i)$
  <u>enddo</u>
!$ACC LOOP INDEPENDENT
  <u>do</u> $i = 1$ <u>to</u> $2n_s$
    $t(k_3+i) = A_3(3,i)z(k_3+i) - A_4(3,i)z(k_4+i)$
  <u>enddo</u>
!$ACC LOOP INDEPENDENT
  <u>do</u> $i = 2$ <u>to</u> $2n_s$
    $t(k_3+i-1) = t(k_3+i-1)+$
      $A_3(2,i)z(k_3+i-1) - A_4(2,i)z(k_4+i)$
  <u>enddo</u>
!$ACC LOOP INDEPENDENT
  <u>do</u> $i = 1$ <u>to</u> $2n_s - 1$
    $t(k_3+i+1) = t(k_3+i+1)+$
      $A_3(4,i)z(k_3+i) - A_4(4,i)z(k_4+i)$
  <u>enddo</u>
!$ACC LOOP INDEPENDENT
  <u>do</u> $i = 1$ <u>to</u> $2n_s - 2$
    $t(k_3+i) = t(k_3+i)+$
      $A_3(1,i+2)z(k_3+i+2)-$
      $A_4(1,i+2)z(k_4+i+2)$
  <u>enddo</u>
!$ACC LOOP INDEPENDENT
  <u>do</u> $i = 1$ <u>to</u> $2n_s - 2$
    $t(k_3+i+2) = t(k_3+i+2)+$
      $A_3(5,i)z(k_3+i) - A_4(5,i)z(k_4+i)$
  <u>enddo</u>
!$ACC LOOP INDEPENDENT
  <u>do</u> $i = 1$ <u>to</u> $2n_s$
    $temp(i) = t(k_3+i)$
  <u>enddo</u>
!$ACC LOOP INDEPENDENT
  <u>do</u> $i = 1$ <u>to</u> $2n_s$
    $t(k_3+i) = t(k_3+i) - t(k_2+i)$
  <u>enddo</u>
!$ACC LOOP INDEPENDENT
  <u>do</u> $i = 1$ <u>to</u> $2n_s$
    $t(k_2+i) = t(k_2+i) + temp(i)$
  <u>enddo</u>
<u>enddo</u>
!$ACC END KERNELS
!$ACC END DATA

ISBN: 978-988-19252-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

We point out that, due to limited space, the part of the algorithm corresponding to the first and last block rows of the H_B matrix is not included. The GPU realization of the red computational cycle can be described by an analogous algorithm.

MAPPING ONTO A FIXED SIZE ARCHITECTURE

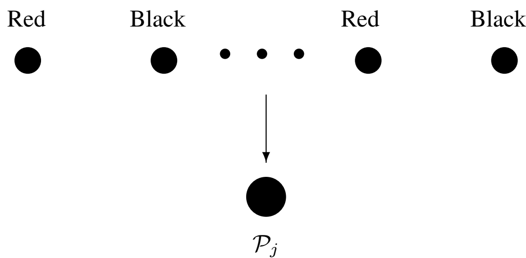
When implementing the algorithm on a fixed core parallel system, consisting of P cores, groups of threads have to be mapped onto the GPU and host processor cores. In this section we describe the mapping process in the case of $n_s = kP$, since simple adjustments can be made to cover all other cases. In this particular case, however, the computational cost is uniform for all cores. The mapping mechanism we shall follow is that of associating k consecutive threads to each one of the $\mathcal{P}_j$, $(j = 1, \ldots, P)$ cores of the fixed size architecture and is shown schematically bellow.



It becomes obvious, following the above mapping, that with each $\mathcal{P}_j$ core we associate the k core threads $V_{(j-1)k+1}, \ldots, V_{jk}$.

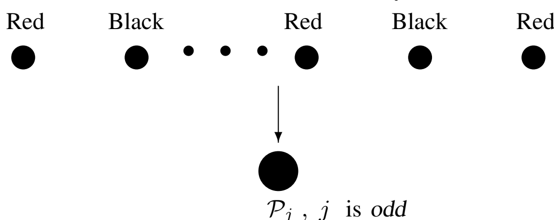
Observe that :

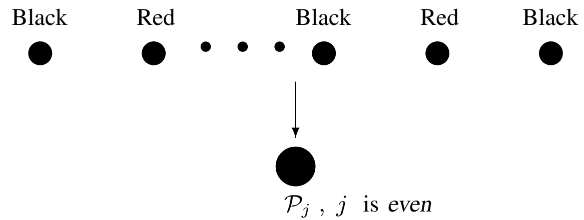
- Whenever k is *even* the indices $(j-1)k+1$ and jk satisfy $(j-1)k+1$ is *odd* while jk is *even*. Hence the core threads $V_{(j-1)k+1}$ and V_{jk} are respectively *red (odd)* and *black (even)* threads and therefore schematically we have:



  Thus, with core $\mathcal{P}_j$ we associate the k *red* vectors $\boldsymbol{t}_l$, $l = (j-1)k, \ldots, jk-1$ and the k *black* vectors $\boldsymbol{t}_{2p+l}$, $l = (j-1)k+1, \ldots, jk$.

- Whenever k is *odd* the indices $(j-1)k+1$ and jk satisfy $(j-1)k+1$ and jk are *odd* when j is *odd* while $(j-1)k+1$ and jk are *even* when j is *even*. Hence the threads $V_{(j-1)k+1}$ and V_{jk} are both *red (odd)* when j is *odd* while they are both *black (even)* when j is *even* and therefore schematically we have:





  Thus, when j is *odd*, with core $\mathcal{P}_j$ we associate the $k+1$ *red* vectors $\boldsymbol{t}_l$, $l = (j-1)k, \ldots, jk$ and the $k-1$ *black* vectors $\boldsymbol{t}_{2p+l}$, $l = (j-1)k+1, \ldots, jk-1$, while, when j is *even*, with core $\mathcal{P}_j$ we associate the $k-1$ *red* vectors $\boldsymbol{t}_l$, $l = (j-1)k+1, \ldots, jk-1$ and the $k-1$ *black* vectors $\boldsymbol{t}_{2p+l}$, $l = (j-1)k, \ldots, jk$.

All the above are used to carry out each step of the following Schur complement algorithm:

Parallel Algorithm for Schur comp. collocation equations

S1: *Solve in parallel on host* $D_R \hat{\boldsymbol{b}}_R = \boldsymbol{b}_R$

S2: *Send matrices* A_3 *and* A_4 *to GPU*

S3: *Evaluate in parallel on GPU* $\hat{\boldsymbol{b}}_B = \boldsymbol{b}_B - H_R \hat{\boldsymbol{b}}_R$

S4: *Solve in parallel with BiCGSTAB* $S\, \boldsymbol{x}_B = \hat{\boldsymbol{b}}_B$

S5: *Evaluate in parallel on GPU* $\hat{\boldsymbol{x}}_B = H_B \boldsymbol{x}_B$

S6: *Solve in parallel on host* $D_R \hat{\boldsymbol{x}}_R = \hat{\boldsymbol{x}}_B$

S7: *Evaluate in parallel on host* $\boldsymbol{x}_R = \hat{\boldsymbol{b}}_R - \hat{\boldsymbol{x}}_R$

The parallel algorithm's computations in step S4 for the BiCGSTAB method are performed on the host except for the two matrix vector multiplications at each iteration step involving the Schur complement matrix S. More specifically, the computation in step S4, involving the matrices H_R and H_B as multipliers, is executed on the GPU as follows:

Evaluation of $\boldsymbol{t} = S\boldsymbol{p}$

S1: *Send* $\boldsymbol{p}$ *from host to GPU*

S2: *Evaluate in parallel on GPU* $\boldsymbol{t} = H_B\, \boldsymbol{p}$

S3: *Send* $\boldsymbol{t}$ *from GPU to host*

S4: *Solve in parallel on host* $D_R \boldsymbol{s} = \boldsymbol{t}$

S5: *Send* $\boldsymbol{s}$ *from host to GPU*

S6: *Evaluate in parallel on GPU* $\boldsymbol{q} = H_R\, \boldsymbol{s}$

S7: *Send* $\boldsymbol{q}$ *from GPU to host*

S8: *Evaluate in parallel on host* $\boldsymbol{t} = D_B\, \boldsymbol{p} - \boldsymbol{q}$

The communication cost for data movement between host and GPU memory is the cost for transferring two vectors of size $2n_s^2$ in each direction. Thus, the communication cost for every BiCGSTAB iteration step, is the cost of transferring eight vector of size $2n_s^2$ in each direction, since matrices A_3 and A_4 are moved and stored in the GPU memory once at the beginning of the solution process.

IV. REALIZATION ON A GPU SHARED-MEMORY PARALLEL COMPUTER

HP's SL390s G7 is a shared memory architecture machine, consisting of a 6-core Xeon X5660@2.8GHz type processor with 12 MB Level 3 cache memory. The total memory is 24 GB and the operating system is Oracle Linux version 6.2. This machine has also a Fermi edition Tesla M2070

GPU [14] connected via a PCI-e gen2 slot. The GPU has 6GB of memory and 448 cores on 14 multiprocessors. The application is developed in double precision Fortran code using OpenMP [15], [16] and OpenACC [17] standards with PGI's compilers version 12.9 [18]. For the basic linear algebra operations subroutines from scientific libraries BLAS and LAPACK [19] are considered, as they are utilized for this specific platform.

For the implementation of the above parallel algorithm the test Dirichlet Helmholtz problem, which accepts the following exact solution

$$u(x, y) = 10\ \phi(x)\ \phi(y)\ ,\ \ \phi(x) = e^{-100(x-0.1)^2}(x^2 - x),$$

with $\lambda = 1$ was solved. The following Table I presents the behaviour of the method regarding convergence iteration steps and linear system error L2-norm for discretization sizes up to 2048 finite elements in each direction.

Table I: Iterations and error L2-norm

n_s	Iterations	$\|\|b - Ax^{(m)}\|\|_2$
256	294	6.06e-11
512	589	2.85e-11
1024	1161	1.39e-11
2048	3726	9.59e-12

Focusing now, on the performance of our parallel algorithm and its implementation on the given parallel environment, we have collected time measurements using several execution parameters, such as the number of host cores and GPU enabling. Table II below summarize these execution time measurements in seconds each of one for different problem size starting from $n_s = 256$ up to $n_s = 2048$ discretizations.

Table II: Execution time for Host/Accelerator devices

$n_s = 256$	CPU	GPU + CPU
CPU cores	Total Time	Total Time
1	12.24	11.18
2	8.12	7.53
4	4.87	5.63

$n_s = 512$	CPU	GPU + CPU
CPU cores	Total Time	Total Time
1	88.83	71.25
2	52.94	46.59
4	34.25	32.63

$n_s = 1024$	CPU	GPU + CPU
CPU cores	Total Time	Total Time
1	750.35	549.82
2	448.76	352.95
4	283.14	250.64

$n_s = 2048$	CPU	GPU + CPU
CPU cores	Total Time	Total Time
1	9176	6770
2	5001	4387
4	2999	2839

In the following last Table III the execution time in more detail for the cases involving the GPU can be found. The time for data transferring from CPU to GPU and vice versa for all discretizations is presented. The computation time for every available CPU core number is also measured.

Table III: Multi-core execution time

n_s	GPU - CPU	Computation Time		
	Comm. Time	1 Core	2 Cores	4 Cores
256	1.52	9.66	6.01	4.11
512	6.81	64.44	39.78	25.82
1024	44.3	505.5	308.6	206.3
2048	541	6229	3846	2298

We have to mention that the communication time between CPU and GPU is independent of the number of the CPU cores, because in our algorithm the transferring phase is performed by one thread due to the computation load balancing. This avoids the data bottleneck movements between multiple CPU threads over the PCI slot of the machine.

V. Conclusions

A new parallel algorithm for implementing the BiCGSTAB iterative method for solving the Hermite Collocation equations, arising from elliptic PDEs, has been developed and realized on multi-core machines with GPUs. The performance of the algorithm is affected by the size of the problem and the number of CPU cores. A performance acceleration of up to almost 30% is observed.

Acknowledgment

The present research work has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program *Education and Lifelong Learning* of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALIS. Investing in knowledge society through the European Social Fund.

References

[1] C. E. Houstis, E. N. Houstis, and J. Rice, "Pde computations: Methods and performance evaluation," *Par. Comp.*, vol. 5, pp. 141–163, 1997.
[2] R. Varga, *Matrix Iterative Analysis*. New York: Springer Verlag, 2000.
[3] H. der Vorst, "Bi-cgstab : A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems," *SIAM J. Sci.Stat.Comp.*, vol. 13, pp. 631–644, 1992.
[4] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.
[5] C.C.Christara, "Parallel solvers for spline collocation equations," *Advances in Engineering Software*, vol. 27, pp. 71–89, 1996.
[6] S.H.Brill and G.F.Pinder, "Parallel implementation of the bi-cgstab method with block red-black gauss-seidel preconditioner applied to the hermite collocation discretization of partial differetial equations," *Parallel Computing*, vol. 28, pp. 399–414, 2002.
[7] E. Mathioudakis, E. Papadopoulou, and Y. Saridakis, "Iterative solution of elliptic collocation systems on a cognitive parallel computer," *Computers and Maths with Appl.*, vol. 48, pp. 951–970, 2004.
[8] ——, "Mapping parallel iterative algorithms for pde computations on a distributed memory computers," *Parallel Algorithms and Applications*, vol. 8, pp. 141–154, 1996.
[9] ——, "Bi-cgstab for collocation equations on distributed memory parallel computers," *Numerical Mathematics and advanced applications - ENUMATH 2001, Springer*, pp. 957–966, 2003.
[10] E. Mathioudakis and E. Papadopoulou, "Grid computing for the bi-cgstab applied to the solution of the modified helmholtz equation," *Int. J. App. Maths and comp. sciences*, vol. (4),3, pp. 179–184, 2007.
[11] T. Papatheodorou, "Block aor iteration for nonsymmetric matrices," *Math. Comp.*, vol. (41),164, pp. 511–525, 1983.
[12] E. Mathioudakis, E. Papadopoulou, and Y. Saridakis, "Preconditioning for solving hermite collocation by the bi-cgstab," *WSEAS Trans. on Mathematics*, vol. (5),7, pp. 811–816, 2006.
[13] J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst, *Numerical Linear Algebra for high-performance computers*. Phil.: SIAM, 1998.
[14] Nvidia, *http://www.nvidia.com/object/tesla-servers.html*.
[15] C. Rohit, *Parallel programming with OpenMP*. M. K., 2001.
[16] OpenMP, *http://www.openmp.org*.
[17] OpenACC, *http://www.openacc.org*.
[18] PGI, *http://www.pgroup.com*.
[19] NetLib, *http://www.netlib.org*.