Proceedings of the International Multiconference on Computer Science and Information Technology pp. 297–302
ISBN 978-83-60810-27-9
ISSN 1896-7094

# Finite Element Approximate Inverse Preconditioning using POSIX threads on multicore systems

**G. A. Gravvanis, P. I. Matskanidis**
Department of Electrical and Computer Engineering, School of Engineering, Democritus University of Thrace, University Campus, Kimmeria, GR 67100 Xanthi, Greece
Email: ggravvan@ee.duth.gr; pascmats@ee.duth.gr

**K. M. Giannoutakis**
Centre for Research and Technology Hellas, Informatics and Telematics Institute, GR 57001, Thermi, Greece
Email: kgiannou@iti.gr

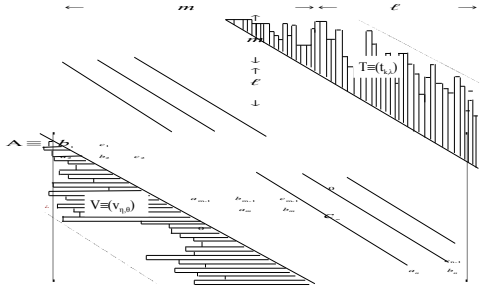
**E. A. Lipitakis**
Department of Informatics, Athens University of Economics and Business, 76 Patission street, GR 104 34 Athens, Greece
Email: eal@aueb.gr

***Abstract*—Explicit finite element approximate inverse preconditioning methods have been extensively used for solving efficiently sparse linear systems on multiprocessor and multicomputer systems. New parallel computational techniques are proposed for the parallelization of explicit preconditioned biconjugate conjugate gradient type methods, based on Portable Operating System Interface for UniX (POSIX) Threads, for multicore systems. Parallelization is achieved by assigning every loop of the parallel explicit preconditioned bi-conjugate conjugate gradient-STAB (PEPBiCG-STAB) to the desired number of threads, thus achieving for-loop parallelization. Theoretical estimates on speedups and efficiency are also presented. Finally, numerical results for the performance of the PEPBiCG-STAB method for solving characteristic two dimensional boundary value problems on multicore computer systems are presented, which are favorably compared to corresponding results from multiprocessor systems. The implementation issues of the proposed method are also discussed using POSIX Threads on a multicore system.**

## I. Introduction

LET us consider the linear system derived by the finite element (FE) method for solving boundary value problems in two dimensions, [6] ,[8], [9], [10], i.e.

$$Au = s, \tag{1}$$



(2)

where the coefficient matrix A is a non-singular large, sparse, unsymmetric, positive definite, diagonally dominant (n×n) matrix of irregular structure (where all the off-center band terms are grouped in regular bands of width $\ell$ at semi-bandwidth m), [2], while u is the FE solution at the nodal points and s is a vector, with components resulting from a combination of source terms and imposed boundary conditions.

During the last decades, explicit approximate inverse preconditioning methods have been extensively used for efficiently solving sparse linear systems on multiprocessor systems, [3], [4], [9], [10]. In recent years many researchers have derived preconditioners based on various techniques, which are difficult to be implemented on parallel systems, [7 ], [11], [12]. The effectiveness of explicit approximate inverse preconditioning schemes relies on the use of suitable preconditioners that are close approximants to the inverse of the coefficient matrix and are fast to compute in parallel, [2 ], [5].

The approximate inverse $M_r^{\delta l}$ represents a class of generalized approximate inverses that includes various families of approximate inverses according to the requirements of accuracy, storage and computational work, as can be seen by the following diagrammatic relation:

$$\underline{\text{class I}} \qquad \underline{\text{class II}} \quad \underline{\text{class III}}$$
$$A^{-1} \equiv M \leftarrow M_{r=m-1}^{\delta l} \leftarrow M_r^{\delta l} \leftarrow M_i \tag{3}$$

where M is the exact inverse resulting in a direct method, i.e. r=m-1 and δl=n with the disadvantage of high memory requirements and computational work for large order systems. The entries of the class I inverse have been retained after the computation of the exact inverse ( $r$=m-1, δl=n) by retaining only δl and δl-1 elements in the lower and upper part of the exact inverse. The entries of the class II inverse have been computed and retained during the computational procedure of the (approximate) inverse ( $r$≤m-1, δl<n), while the entries of the class III of the generalized approximate inverse retains only the diagonal elements, i.e. δl=1 while requires only the the diagonal entries of the sparse lower matrix $L_r$, [6], [10], resulting in a fast inverse algorithm.

Furthermore, we present the **E**xplicit **P**reconditioned **BI**-conjugate **C**onjugate **G**radient-**STAB** (**EPBI-CGSTAB**) method, which can be expressed by the following compact scheme:

Let $u_0$ be an arbitrary initial approximation to the solution vector u. Then,

compute $$r_0 = s - Au_0, \quad (4)$$

set $$r_0^{'} = r_0, \ \rho_0 = \alpha = \omega_0 = 1 \text{ and } v_0 = p_0 = 0 \quad (5)$$

Then, for i=1,….,n (until convergence) compute the vectors $u_i$, $r_i$ and the scalar quantities α, β, $\omega_i$ as follows:

calculate $$\rho_i = \left(r_0^{'}, r_{i-1}\right) \text{ and } \beta = \frac{\left(\rho_i / \rho_{i-1}\right)}{\left(\alpha / \omega_{i-1}\right)} \quad (6)$$

compute $$p_i = r_{i-1} + \beta\left(p_{i-1} - \omega_{i-1} v_{i-1}\right), \quad (7)$$

form $$y_i = \mathbf{M}_r^{\delta l} p_i \text{ and } v_i = Ay_i, \quad (8)$$

compute $$\alpha = \rho_i / \left(r_0^{'}, v_i\right) \text{ and } x_i = r_{i-1} - \alpha v_i, \quad (9)$$

form $$z_i = M_r^{\delta l} x_i \text{ and } t_i = Az_i, \quad (10)$$

set $$\omega_i = \left(\mathbf{M}_r^{\delta l} t_i, \mathbf{M}_r^{\delta l} x_i\right) / \left(\mathbf{M}_r^{\delta l} t_i, \mathbf{M}_r^{\delta l} t_i\right) \quad (11)$$

compute $$u_i = u_{i-1} + \alpha y_i + \omega_i z_i \text{ and } r_i = x_i - \omega_i t_i \quad (12)$$

Assuming that the approximate inverse $M_r^{\delta l}$ can be compactly stored in n×(2δl-1) diagonal vectors, then the computational complexity of the **EPBICG-STAB** method is ≈O[(6δl+4 $\mathcal{C}$ +16)n mults + 6n adds]*v* operations, where ν is the number of iterations required for convergence to a certain level of accuracy, [6], [10].

In this article, new parallel computational techniques are proposed for the parallelization of explicit preconditioned conjugate gradient type methods, based on Portable Operating System Interface for UniX (POSIX) Threads, for multicore systems. The excessive overhead produced by the template-based parallel implementations was avoided by using POSIX Threads, maximizing the overall performance of the parallel implementation of the **PEPBICG-STAB** method and throttling it close to the corresponding theoretical estimate.

Finally, numerical results for the performance of the **PEPBICG-STAB** method for solving characteristic two dimensional boundary value problems on multicore computer systems are presented, which are favorably compared to corresponding results from multiprocessor systems. The implementation issues of the proposed method are also discussed using POSIX Threads on a multicore computer system.

## II. Parallel Biconjugate Conjugate Gradient-Type Method

In this section, we demonstrate the parallel implementation of the **EPBICG-STAB** method designed for multicore computer systems using POSIX threads, [1], [13].

Specifically, kernel-level threads of the POSIX 1003.1c standard are used in a thread pool pattern, [1], originally designed by Sun Microsystems for Solaris Operating System, [13], and modified for the purposes of our computations. A thread pool manages a certain number of threads that perform tasks based on a job queue. Every time a thread finishes its task it switches to idle state, waiting for another task from the queue to be assigned to it. In this way, fork and join of threads is limited to just the active threads in the pool, rather than creating and detaching every thread that performs a task. Hence, due to the reduction of latencies an improvement is expected on the parallel performance of the proposed implementation.

This parallel implementation is based on the standard for-loop parallelization model. Let *nthreads* denote the number of threads used and *thread_id* be the "id" number of each thread (from 0 to *nthreads*). Then, every thread, denoted as *localthr,* deals only with the number of allocated elements, where *localthr*=n/*nthreads*.

Hence, the **P**arallel **E**xplicit **P**reconditioned **BI**conjugate **C**onjugate **G**radient-**STAB** (**PEPBICG-STAB**) method can be expressed by the following algorithmic scheme:

Let $u_0$ be an arbitrary initial approximation to the solution vector u. Then,

**thread_pool_queue**
**for** j=(1+*thread_id***localthr*) **to** (*thread_id*+1)**localthr*

$$\left(r_0\right)_j = s_j - A\left(u_0\right)_j \quad (13)$$

**end for**
**thread_pool_wait**
**thread_pool_queue**
**for** j=(1+*thread_id***localthr*) **to** (*thread_id*+1)**localthr*

$$\left(r_0^{'}\right)_j = \left(r_0\right)_j \quad (14)$$

$$\left(p_i\right)_j = 0.0 \quad (15)$$

$$\left(v_i\right)_j = 0.0 \quad (16)$$

**end for**
**thread_pool_wait**

$$\rho_0 = \alpha = \omega_0 = 1.0 \quad (17)$$

$$v_0 = p_0 = 0.0 \quad (18)$$

Then, **for i=1, ..., (until convergence)** compute in parallel the vectors $u_i, r_i$ and the scalar quantities $\alpha, \beta, \omega_i$ as follows:

**thread_pool_queue**
**for** j=(1+*thread_id***localthr*) **to** (*thread_id*+1)**localthr*
**do reduction +** $\rho_i$

$$\rho_i = \left(r_0^{'}\right)_j \cdot \left(r_{i-1}\right)_j \quad (19)$$

**end for**
**thread_pool_wait**

$$\beta = \left(\rho_i / \rho_{i-1}\right) / \left(\alpha / \omega_{i-1}\right) \quad (20)$$

**thread_pool_queue**

**for** j=(1+*thread_id***localthr*) **to** (*thread_id*+1)**localthr*

$$(p_i)_j = (r_{i-1})_j + \beta\left((p_{i-1})_j - \omega_{i-1}(v_{i-1})_j\right) \quad (21)$$

**end for**
**thread_pool_wait**
**thread_pool_queue**
**for** j=(1+*thread_id***localthr*) **to** (*thread_id*+1)**localthr*

$$(y_i)_j = \left(\sum_{k=\max(1,j-\delta l+1)}^{\min(n,j+\delta l-1)} \mu_{j,k}(p_i)_k\right) \quad (22)$$

**end for**
**thread_pool_wait**
**thread_pool_queue**
**for** j=(1+*thread_id***localthr*) **to** (*thread_id*+1)**localthr*

$$(v_i)_j = A(y_i)_j \quad (23)$$

**end for**
**thread_pool_wait**
**thread_pool_queue**
**for** j=(1+*thread_id***localthr*) **to** (*thread_id*+1)**localthr*
**do reduction +** $d_i$

$$d_i = (r_0')_j \cdot (v_i)_j \quad (24)$$

**end for**
**thread_pool_wait**

$$\alpha = \rho_i / d_i \quad (25)$$

**thread_pool_queue**
**for** j=(1+*thread_id***localthr*) **to** (*thread_id*+1)**localthr*

$$(x_i)_j = (r_{i-1})_j - \alpha(v_i)_j \quad (26)$$

**end for**
**thread_pool_wait**
**thread_pool_queue**
**for** j=(1+*thread_id***localthr*) **to** (*thread_id*+1)**localthr*

$$(z_i)_j = \left(\sum_{k=\max(1,j-\delta l+1)}^{\min(n,j+\delta l-1)} \mu_{j,k}(x_i)_k\right) \quad (27)$$

**end for**
**thread_pool_wait**
**thread_pool_queue**
**for** j=(1+*thread_id***localthr*) **to** (*thread_id*+1)**localthr*

$$(t_i)_j = A(z_i)_j \quad (28)$$

**end for**
**thread_pool_wait**

$$a_1 = 0.0 \quad (29)$$

$$a_2 = 0.0 \quad (30)$$

**thread_pool_queue**
**for** j=(1+*thread_id***localthr*) **to** (*thread_id*+1)**localthr*
**do reduction +** $a_1$ , $a_2$

$$(tt_i)_j = \left(\sum_{k=\max(1,j-\delta l+1)}^{\min(n,j+\delta l-1)} \mu_{j,k}(t_i)_k\right) \quad (31)$$

$$a_1 = (tt_i)_j \cdot (z_i)_j \quad (32)$$

$$a_2 = (tt_i)_j \cdot (tt_i)_j \quad (33)$$

**end for**
**thread_pool_wait**

$$\omega_i = a_1 / a_2 \quad (34)$$

**thread_pool_queue**
**for** j=(1+*thread_id***localthr*) **to** (*thread_id*+1)**localthr*

$$(u_i)_j = (u_{i-1})_j + \alpha(y_i)_j + \omega_i(z_i)_j \quad (35)$$

$$(r_i)_j = (x_i)_j - \omega_i(t_i)_j \quad (36)$$

**end for**
**thread_pool_wait**

The computational complexity of the **PEPBICG-STAB** method is ≈O[(6δl+4 $\ell$ +16)*localthr* mults + 6*localthr* adds] ν operations, where ν denotes the number of iterations required for convergence to a certain level of accuracy and *localthr* is the number of rows of the matrices distributed onto each thread, [2], [3], [5].

Thus, the theoretical speedup and efficiency of the **PEPBICG-STAB** methods are respectively:

$$S_p^{\delta l} = \frac{1}{\frac{1}{\text{nthreads}} + \frac{12t_l}{O(6\delta l+4\ell+16)n \cdot t_m}} \quad (37)$$

and

$$E_p^{\delta l} = \frac{1}{1 + \frac{12t_l \cdot \text{nthreads}}{O(6\delta l+4\ell+16)n \cdot t_m}} \quad (38)$$

where $t_m$ denotes the computational time of a multiplication, while $t_l$ is the threads assignment latencies. It is obvious that for $\delta l \to \infty$ , then $S_p^{\delta l} \to$nthreads and $E_p^{\delta l} \to 1$, (37)-(38).

The effectiveness of the parallel explicit preconditioned conjugate gradient type method is related to the fact that the approximate inverse × vector and vector × vector can be efficiently implemented on parallel systems, [1], [2], [5], [13].

## III. Numerical Results

In this section we examine the effectiveness and applicability of the new proposed parallel schemes for solving characteristic two dimensional boundary value problems, on multicore computer systems, using POSIX threads, [1], [13]. The numerical test runs were performed on a Dual 2x Intel Xeon server at 2.0 GHz with 12MB cache memory, 8GB RAM and 1333MHz bus, running Debian GNU/Linux (University College at Cork, Ireland).

Let us consider the following 2D-boundary value problem:

$$\Delta u(x,y) + u(x,y) = f(x,y), \quad (x,y) \in R \quad (39)$$

$$u(x,y) = 0, \quad (x,y) \in \partial R, \quad (39.a)$$

TABLE I

THE CONVERGENCE BEHAVIOR OF THE **PEPBICG-STAB** METHOD FOR SEVERAL VALUES OF n, m, AND δl WITH NUMBER OF THREADS=1.

| n | m | Number of iterations of PEPBICG-STAB method | | | | |
|---|---|---|---|---|---|---|
| | | "Retention" parameter δl | | | | |
| | | δl=1 | δl=m/2 | δl=m | δl=2m | δl=4m |
| 62500 | 251 | 13 | 11 | 8 | 5 | 5 |
| 122500 | 351 | 13 | 11 | 8 | 5 | 5 |
| 160000 | 401 | 13 | 11 | 8 | 5 | 5 |
| 202500 | 451 | 14 | 11 | 9 | 5 | 5 |

TABLE II

THE PERFORMANCE OF THE **PEPBICG-STAB** METHOD FOR SEVERAL VALUES OF THE NUMBER OF THREADS, n AND δl.

| No of Threads | n | m | δl=1 | δl=m/2 | δl=m | δl=2m | δl=4m |
|---|---|---|---|---|---|---|---|
| **1** | 62500 | 251 | 0.601 | 18.043 | 23.056 | 30.183 | 61.502 |
| **2** | | | 0.332 | 9.246 | 11.726 | 15.328 | 31.090 |
| **4** | | | 0.189 | 4.669 | 5.963 | 7.787 | 15.820 |
| **8** | | | 0.157 | 2.426 | 3.043 | 3.919 | 7.968 |
| **1** | 122500 | 351 | 1.184 | 42.828 | 55.145 | 69.803 | 141.597 |
| **2** | | | 0.677 | 21.798 | 27.914 | 35.332 | 71.496 |
| **4** | | | 0.333 | 11.042 | 14.164 | 17.915 | 36.294 |
| **8** | | | 0.297 | 5.598 | 7.173 | 9.048 | 18.332 |
| **1** | 160000 | 401 | 1.554 | 57.764 | 77.395 | 97.981 | 198.367 |
| **2** | | | 0.856 | 29.332 | 39.193 | 49.396 | 99.805 |
| **4** | | | 0.434 | 14.852 | 19.859 | 25.119 | 50.762 |
| **8** | | | 0.384 | 7.531 | 10.057 | 12.694 | 25.682 |
| **1** | 202500 | 451 | 2.139 | 79.866 | 118.654 | 132.285 | 267.750 |
| **2** | | | 1.000 | 40.398 | 53.243 | 66.560 | 134.712 |
| **4** | | | 0.596 | 20.479 | 27.013 | 33.855 | 68.517 |
| **8** | | | 0.488 | 10.357 | 13.662 | 17.117 | 34.647 |

TABLE III

SPEEDUPS OF THE **PEPBICG-STAB** METHOD FOR SEVERAL VALUES OF THE NUMBER OF THREADS, n, m AND δl.

| No of Threads | n | m | δl=1 | δl=m/2 | δl=m | δl=2m | δl=4m |
|---|---|---|---|---|---|---|---|
| **2** | 62500 | 251 | 1.810 | 1.951 | 1.966 | 1.969 | 1.978 |
| **4** | | | 3.180 | 3.864 | 3.867 | 3.876 | 3.888 |
| **8** | | | 3.828 | 7.437 | 7.577 | 7.702 | 7.719 |
| **2** | 122500 | 351 | 1.883 | 1.965 | 1.976 | 1.976 | 1.980 |
| **4** | | | 3.282 | 3.879 | 3.893 | 3.896 | 3.901 |
| **8** | | | 3.987 | 7.651 | 7.688 | 7.715 | 7.724 |
| **2** | 160000 | 401 | 1.955 | 1.969 | 1.975 | 1.984 | 1.988 |
| **4** | | | 3.305 | 3.889 | 3.897 | 3.901 | 3.908 |
| **8** | | | 4.047 | 7.670 | 7.696 | 7.719 | 7.724 |
| **2** | 202500 | 451 | 1.986 | 1.977 | 1.981 | 1.987 | 1.988 |
| **4** | | | 3.333 | 3.900 | 3.904 | 3.907 | 3.908 |
| **8** | | | 4.070 | 7.711 | 7.720 | 7.728 | 7.728 |

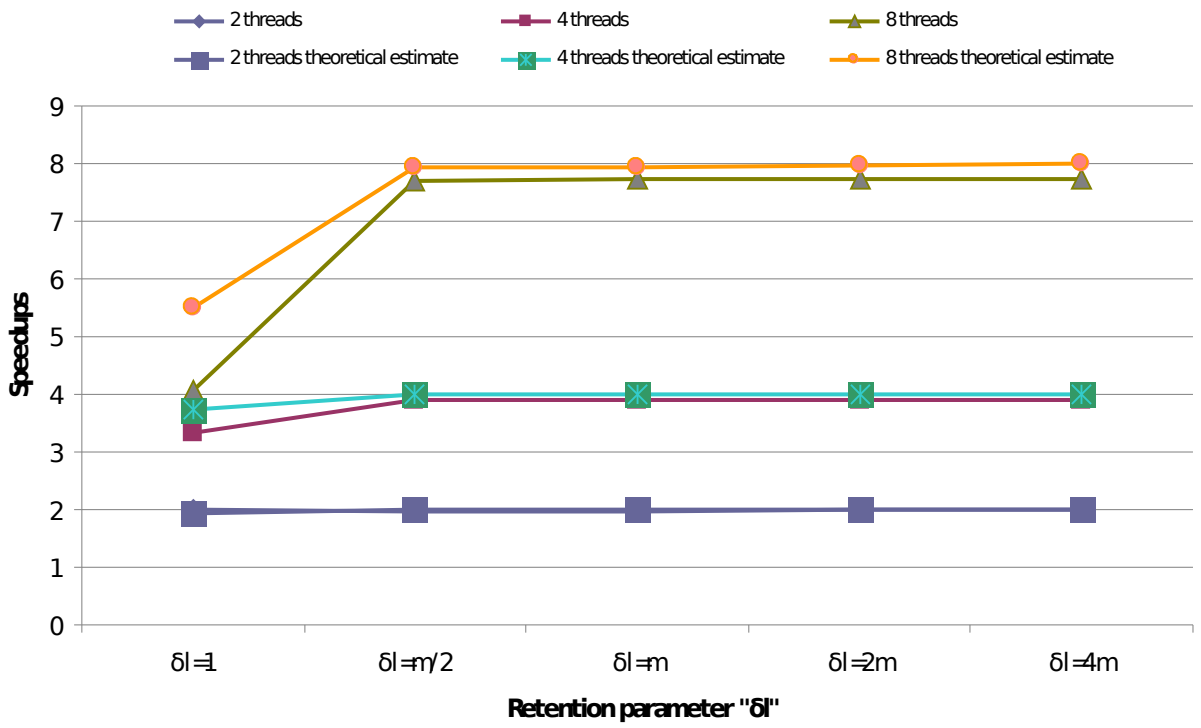



Fig 1. Speedups versus the retention parameter δl for the **PEPBICG-STAB** method, along with theoretical upper bounds, for n=202500.

where Δ is the Laplace operator, R is the unit square and ∂R denotes the boundary of R. The domain R∪∂R was covered by a non-overlapping triangular network resulting in a hexagonal mesh. The right hand side vector of the sparse linear system (1) was computed as the product of the coefficient matrix A by the solution vector, with its components equal to unity. The "width" parameter was set to $\ell=3$ and the "fill-in" parameter to r=2. The iterative process was terminated when $\|u_{i+1}-u_i\|_\infty<10^{-5}$, [6], [8], [10].

The convergence behavior and the performance, given in "seconds.hundreds", of the **PEPBICG-STAB** method for several values of the order n, semi-bandwidth m and "retention" parameter δl is presented in Table I and Table II respectively. The speedups and number of threads allocated of the **PEPBICG-STAB** method for several values of the order n, semi-bandwidth m and "retention" parameter δl are given in Table III.

In Figure 1 the speedups and number of threads allocated for several values of δl along with theoretical estimates, are presented for the **PEPBICG-STAB** method.

It should be mentioned that the convergence behavior of the explicit preconditioned biconjugate conjugate gradient method was found to be in qualitative agreement with similar theoretical estimates obtained, [6], [10]. Additionally, when the "retention" parameter δl=1 the speedups and the efficiency are significantly improved, while for larger values of the "retention" parameter, i.e. multiples of the semi-bandwidth m, the speedups and the efficiency are slightly improved in comparison with results obtained on a multiprocessor system using OpenMP, [2].

Finally, it is pointed out that for large values of the "retention" parameter δl the speedups and the efficiency tend to the upper theoretical bound, for the parallel explicit preconditioned biconjugate conjugate gradient method, since the coarse granularity amortizes the parallelization overheads.

## IV. Acknowledgment

The author would like to express his thanks to Dr. John Morrison of the Department of Computer Science, University College of Cork, for the provision of suitable computational facilities and support through the WebCom-G project funded by Science Foundation Ireland. The author wishes to acknowledge the SFI/HEA Boole Centre for Research in Informatics and the Centre for Unified Computing for the provision of computational facilities and support. Part of this work is partially funded by Science Foundation Ireland and the Higher Education Authority under the Cosmogrid project.

## References

[1] D. R. Butenhof, *Programming with POSIX® Threads*, Addison-Wesley, 1997.

[2] K. M. Giannoutakis, and G. A. Gravvanis, "High performance finite element approximate inverse preconditioning," *Applied Mathematics and Computation*, vol. 201, pp. 293–304, 2008.

[3] G. A. Gravvanis, "High Performance Inverse Preconditioning," *Archives of Computational Methods in Engineering*, vol. 16, no. 1, pp. 77-108, 2009.

[4] G. A. Gravvanis, "Explicit Approximate Inverse Preconditioning Techniques," *Archives of Computational Methods in Engineering*, vol. 9, no. 4, pp. 371-402, 2002.

[5] G. A. Gravvanis, and K. M. Giannoutakis, "Fast parallel finite element approximate inverses," *Computer Modeling in Engineering and Sciences*, vol. 32, no. 1, pp. 35-44, 2008.

[6] G. A. Gravvanis, and E. A. Lipitakis, "An explicit sparse unsymmetric finite element solver," *Commun. Numer. Meth. in Engin.*, vol. 12, pp. 21-29, 1996.

[7] M. J. Grote, and T. Huckle, "Parallel preconditioning with sparse approximate inverses," *SIAM J. Sci. Computing*, vol. 18, pp. 838-853, 1997.

[8] E. A. Lipitakis, "Generalized extended to the limit sparse factorization techniques for solving large unsymmetric finite elements systems," *Computing*, vol. 32, pp. 255-270, 1984.

[9] E. A. Lipitakis, and D. J. Evans, "Explicit semi-direct methods based on approximate inverse matrix techniques for solving boundary-value problems on parallel processors," *Math. and Computers in Simulation*, vol. 29, pp. 1-17, 1987.

[10] E. A. Lipitakis, and G. A. Gravvanis, "Explicit preconditioned iterative methods for solving large unsymmetric FE systems," *Computing*, vol. 54, pp. 167-183, 1995.

[11] Y. Saad, and H. A. van der Vorst, "Iterative solution of linear systems in the 20$^{th}$ century," *J. Comp. Applied Math*., vol. 123, pp. 1-33, 2000.

[12] Y. Saad, *Iterative methods for sparse linear systems*, PWS Publishing, 1996.

[13] Sun Microsystems, "Multithreaded Programming Guide" (http://dlc.sun.com/pdf/816-5137/816-5137.pdf), 2008.