PARALLELISATION TECHNIQUES FOR THE DUAL RECIPROCITY AND TIME-DEPENDENT BOUNDARY ELEMENT METHOD ALGORITHMS

TIM BASHFORD, KELVIN DONNE, ARNAUD MAROTIN & ALA AL-HUSSANY
Faculty of Architecture, Computing and Engineering, University of Wales, Trinity Saint David Swansea, Wales, United Kingdom.

ABSTRACT
The Dual Reciprocity BEM (DRBEM) and the Time-Dependent BEM (TDBEM) are considered in the context of radiative and time-dependent thermal transport, respectively. In order to achieve sensible solution times for realistic 3D problems with large meshes, a range of optimisation techniques are considered, and a number of parallelisation techniques applied: shared memory using multi-core threading, Graphics Processing Unit (GPU) acceleration using CUDA, and distributed memory on a high performance cluster using MPI. Particular consideration is given to practical methods to invert large dense matrices.
Keywords: BEM, CUDA, MPI, threading

1 INTRODUCTION

The DRBEM and TDBEM algorithms are applied to a three-dimensional radiative-thermal problem illustrated in Figure 1. The context is light-based hair removal, where the clinician requires an estimate of the time-evolving temperature distribution through the hair follicle and the surrounding tissue. A representative target follicle is considered and an unstructured tetrahedral mesh is generated to facilitate final temperature mapping onto the N_v tetrahedral elements.

This paper considers the parallelisation of both the DRBEM and TDBEM for three distinct approaches: shared memory using multi-core threading, GPU acceleration using CUDA, and distributed memory on a high performance cluster using MPI. Details of the DRBEM and TDBEM can be found in, for example, Donne *et al.* [1] and Brebbia *et al.* [2].

A number of authors, including Brebbia *et al.* [2], suggest that the number of internal poles, L, could be significantly less than the total number of volume elements, N_v (~ 29,000 in this case), required. This paper also investigates the accuracy and speed of the DRBEM as a function of internal pole count for this problem.

2 COMPUTATIONAL OPTIMISATION OF THE DRBEM

A significant optimisation was made to the model, through code analysis of the longest sections of code. Following analysis and profiling of the DRBEM code during operation, two methods were identified as major bottlenecks, both comprising three nested loops, each ranging to the total number of elements in the simulation. As such, in the case of the problem under consideration, a mesh comprising ~ 29,000 elements, requiring a total of $\sim 2.4 \times 10^{13}$ iterations for each method. While the code internal to the innermost loop was not especially computationally expensive, even a very minor increase in performance for one iteration will yield a significant improvement in time taken overall.

Through code profiling, a major inefficiency was discovered. As 2D matrices are represented as 1D arrays, column-ordered access to matrices flattened by row inevitably resulted in an extremely poor cache hit rate of ~ 48%. Cache hits for these row-flattened matrices are

ISSN: 2046-0546 (paper format), ISSN: 2046-0554 (online), http://www.witpress.com/journals
DOI: 10.2495/CMEM-V5-N3-395-403

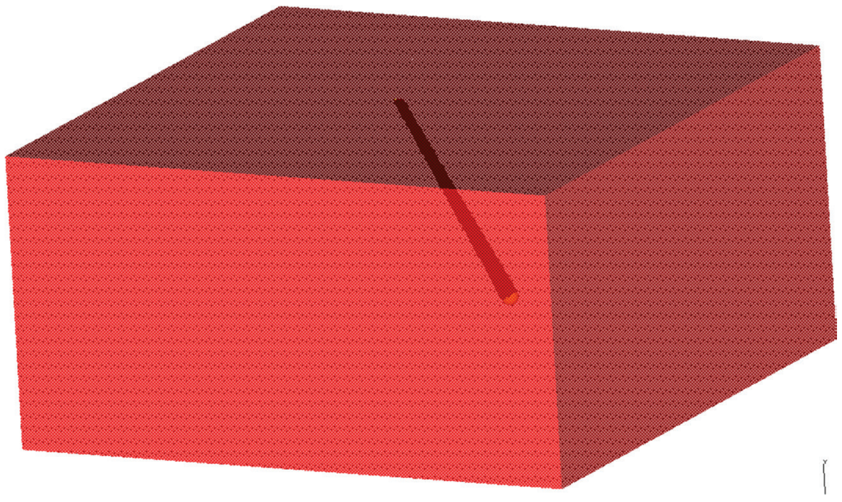


Figure 1: Mesh modelling a hair follicle buried in tissue.

achieved essentially only by coincidence, nearly always at L3 as a result of other threads having recently transferred the block, the probability of a cache hit inevitably decreasing in inverse proportion to the mesh size, compounding the problem. Furthermore, as these cache misses will require block fetches from main memory, they will trigger block replacement especially in the L1 and L2 caches, invariably removing data which will be quickly referenced again, essentially resulting in a higher rate of cache turnover, and causing cache misses for blocks unrelated to the non-sequentially accessed arrays which may otherwise have been hits.

To remedy this, the matrices being accessed were modified to permit sequential access by swapping rows and columns around the pivot value, that is, matrix transposition. This facilitates sequential rather than non-consecutive element access, meaning that access to subsequent array elements is highly likely to result from cache hits. As such, the sampled last level cache hit rate for these functions changed from ~ 48%, the lower figures in the range for larger matrix simulations, to ~ 94%, with matrix size making little difference to the cache hit rate. Most importantly, the increase in cache hit rate resulted in a factor of 10 increase in performance for these portions of the algorithm. For the problem considered, this resulted in a real-time change from ~ 18 hours to ~ 5 hours processing time.

2.1 The inversion of large dense matrices

While the finite element method generates diagonally banded sparse matrices, the BEM is characterised by large, dense matrices, which require inversion. There is significant literature in the inversion of sparse matrices, but very little pertaining to inverting large, dense matrices.

BLAS comprises a low-level specification of linear algebra operations, implemented by a host of well-established linear algebra libraries. Tightly coupled to specific hardware principles, BLAS libraries are almost always considerably faster than bespoke implementations of the same operation.

Libraries which implement BLAS include LAPACK, ATLAS, Eigen BLAS and Intel MKL. A number of very similar libraries, including Armadillo, Eigen, MAGMA and PLASMA also exist; while not fully BLAS-compliant, these libraries tend to include a significant degree of crossover. Several of these were investigated for performance, especially with respect to parallelism. Figure 2 compares the efficiency of these BLAS implementations for varying matrix order, and indicates the benefit of choosing the Intel MKL library.

By contrast, LU decomposition was found to be extremely computationally fast. This was tested using the BLAS LU factorisation preconditioning function *dgetrf* and a second function, *dgetri* to take the preconditioned matrix and complete the inversion in serial. These methods were selected by following the LAPACK naming scheme [3] where the first character denotes the data type, the second and third the matrix type and the final 2–3 a descriptor of the operation. In this case:

$$\underbrace{d}_{\text{double floating point}} \quad \underbrace{ge}_{\text{general matrix}} \quad \underbrace{trf}_{\text{tridiagonal factorisation}}$$

$$\underbrace{d}_{\text{double floating point}} \quad \underbrace{ge}_{\text{general matrix}} \quad \underbrace{tri}_{\text{tridiagonal inversion}}$$

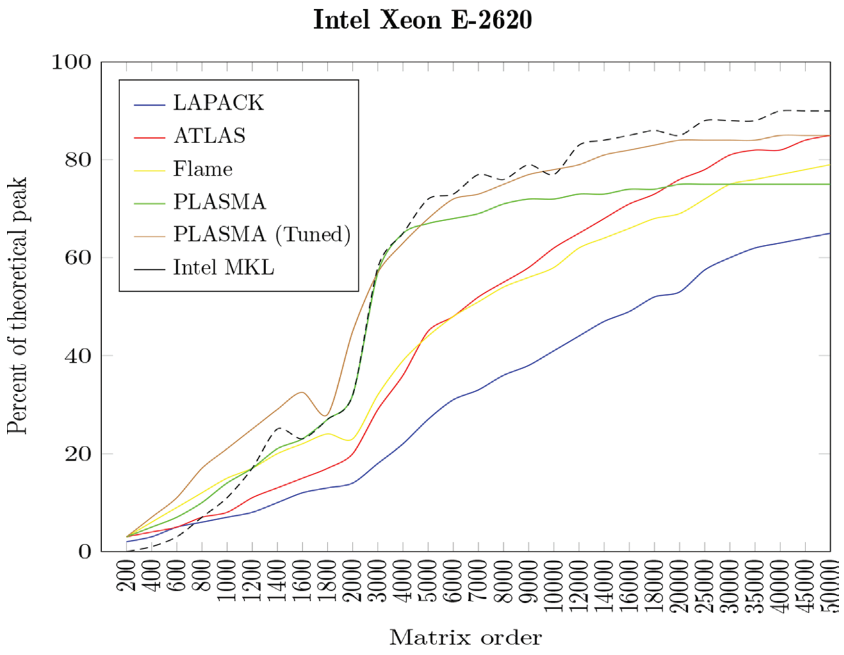
2.2 Parallelisation strategy

An important consideration of any parallel implementation is the model through which memory is shared. One of the greatest restricting factors for an implementation is often that memory may not be accessed outside of a process which allocates it, requiring a form of inter-process communication. Inter-process communication methods often require an undesirable replication of data to permit each process to access identical memory across process boundaries. In nearly all parallel implementations, access to shared memory is highly desirable as it negates copy overheads and memory duplication requirements.

Utilisation of shared and/or distributed memory is dictated by the mode of parallelism being implemented and the hardware available. While some approaches, such as threading, can trivially make use of shared memory, others such as cluster processing through the Message Passing Interface (MPI) provide numerous methods for interacting with distributed memory, requiring specialised hardware to access shared memory. Distributed memory does not always necessarily cross physical system boundaries; in the case of GPGPU implementations data must be transferred between system and graphics memory across the system bus, resulting in a form of distributed memory. Recent development in approaches are working towards rationalising this, with Nvidia implementing algorithms which simulate shared memory in CUDA 6.0 called ‘unified memory’ [5, 6].

2.2.1 Threading

A thread is a stream of executable code, which may be scheduled, sometimes considered and previously termed a ‘lightweight process’. Threads are very similar to processes and, while having less overhead, they cannot run autonomously. All threads are attached to a process and all processes have a main thread of execution, including those intended



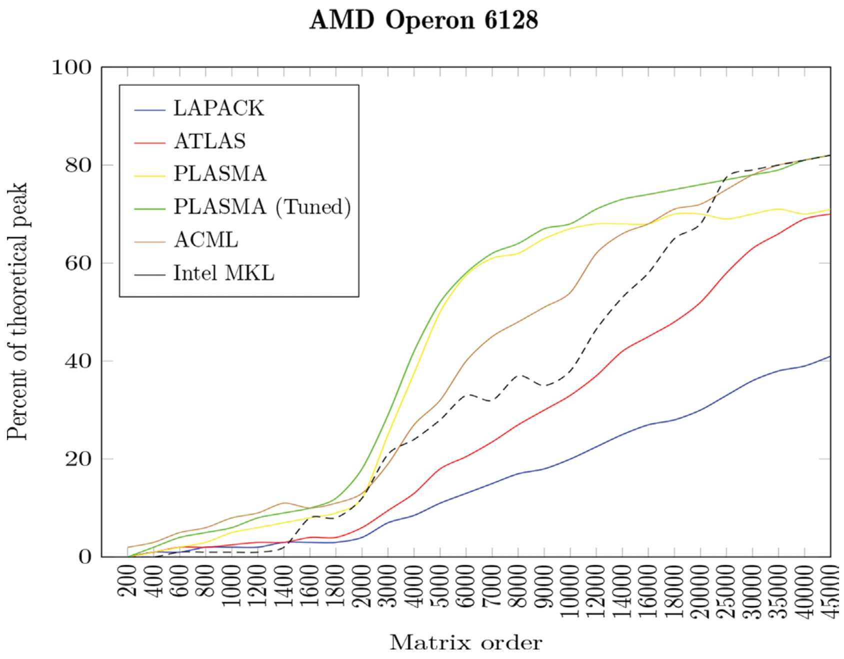


Figure 2: LU decomposition efficiency for different parallel BLAS libraries, generated by Hasan and Whaley [4].

for sequential processers. The development of multiple cores on a single die meant that multithreading became an important method of improving program performance. Unlike parallelism methods across multiple systems, or even multiple processes, threading has a major advantage in that every thread may have access to the same memory, meaning that complex inter-process communication, such as TCP, semaphore, signals and MPI are usually unnecessary.

Threading was applied to the matrix calculation steps of the DRBEM algorithm by utilising the shared memory architecture such that no additional allocation of memory was required. This resulted in a near-linear improvement in speed, scaling nearly linearly with available core count.

2.2.2 MPI

A computer cluster (from here on, 'cluster') is a term often applied broadly and ambiguously, but is specifically used to describe an 'ensemble of independently operational elements integrated by some medium for coordinated and cooperative behaviour' [7]. Cluster computing exists in the domain of high-performance computing (HPC) alongside grid and distributed computing [8]. In more general terms, a cluster is a group of distinct physical computers, capable of separate operation, which can cooperate as one via a network to distribute and perform computing tasks. Clusters come in a variety of formats, from purpose-built systems forming some of the world's most powerful supercomputers, to small collections of desktop computers.

Clusters can vastly surpass the capabilities of individual servers and are highly extensible, with it usually being possible to add additional processing power to a cluster by increasing the number of processing machines (blades). Clusters have long been utilised as research tools for calculations which require substantial processing power to complete in a timely fashion, however, are increasingly being used in commerce.

Considering the non-trivial memory requirements of the algorithm, any distributed memory implementation must consider the risk of encountering page file. With the possible exception of compute clusters specifically designed for heavy paging, generally utilising high-speed hardware for swapping (for example node-level solid state drives), page file is ubiquitously undesirable. The DRBEM requires simultaneous read access to large matrices for all four stages of the calculation, and read and write access for the matrix inversion stages. As such, a core affinity implementation would effectively enforce the required memory for each node to at least memory (per simulation instance) × core count.

Given the relatively large matrices which are desirable for accuracy, this is an unacceptable limitation. While it would be possible to reduce some of the matrices, this would still result in a large amount of overlap of data causing additional memory requirements, and necessitating a significant amount of network traffic to facilitate this secondary distribution of data. As such, this method is sub-optimal. Instead, an approach was utilised whereby the application was designed for node affinity, with each node maintaining a monolithic copy of the data, but processing only a subset of it. By utilising node affinity, MPI parallelism occurs at the inter-node level, but intra-node parallelism occurs through threading. This reduces the impact of a cluster's distributed memory such that each node requires no more physical memory than the serial implementation, and also greatly reduces network traffic.

It should be noted that, while an MPI wrapper exists for the *dgetrf* function, the timings generated were found not to exceed the performance of a single highly parallel compute node due to the overheads involved. Therefore, this was not implemented, and instead the threaded implementation utilised.

2.2.3 CUDA

Unlike a CPU, GPUs have been heavily optimised for graphical display, which at a computational level means large numbers of floating point calculations for every frame; indeed a

GPU is in essence a specialised numeric computing engine. At a hardware level, GPUs are hierarchical, comprising of multiple streaming multiprocessors, each with their own streaming multiprocessors. GPU design revolves around maximising numerical throughput, and the predominant way this is achieved in the hardware industry is through parallelism on a substantial scale. Modern GPUs include a very large number of Algorithmic Logic Units (ALUs) and run a large number of threads, each with concurrent memory access to perform the same operation on different parts of the data simultaneously, being classified as single instruction multiple data (SIMD) within Flynn's taxonomy [9].

An NVIDIA CUDA implementation was finally generated, using the threaded implementation as an initial code base due to the comparable shared memory technique of the two approaches. While CUDA 6.0 supports shared memory between system and device for cross-kernel access, the efficiency of this is poor for non-monolithic data storage. As such, a block-copy implementation was selected instead, with each kernel being given exactly the data it needs to calculate a memory-limited subset of the results, with as many kernels being launched as necessary for processing to complete.

Memory overheads for this code were largely avoided as memory was copied from the device to host directly into storage arrays in blocks, avoiding allocation of temporary host storage. Despite this, processing overheads for these copy operations were significant. The speed of these three methods relative to one another for a variety of mesh sizes is discussed in section 2.4.

Similar to the MPI implementation, while a GPGPU-compliant implementation of the *dgetrf* function exists in the MAGMA libraries, the performance was found to be significantly worse than that of the CPU threaded implementation. As such, the matrix inversion routine was not offloaded to the GPU.

2.3 Internal pole count

The DRBEM does not require the number of internal poles to be the same as the number of volume elements where the solution is desired. Therefore, calculation time can be significantly reduced if the number of internal poles is reduced and the poles are selected from the regions of interest where there is a graduated fine mesh. This was easily achieved using a random number generator, where the selection of internal poles is automatically biased towards the fine mesh region of interest. Given that meshes are generated to be dense around the regions of interest, selection of entirely random elements will result in selection of a good range of elements, assuming sufficient elements are selected.

This means that the ideal outcome is one which makes at worst a very minor sacrifice for accuracy, but in doing so gains a significant increase in speed. Table 1 illustrates the dramatic reduction in calculation time for varying pole count. Due to the DRBEM being of time complexity $O(N^3)$ for much of the simulation, a relatively small decrease in the number of input elements results in a significant decrease in time taken. 9 values to test were selected based on previous experience with the algorithm, along with timing taken for each simulation. Experiments on 1,000 and 1,200 internal poles were re-run a series of times to test for statistical variance.

For each experiment where $L < N_v$, the mean square error relative to the $L = N_v$ experiment is calculated and displayed in Table 2.

Simulations at 1,500 internal poles and lower suffer from the greatest degree of error, with simulations less than 5,000 poles making only a moderate improvement to accuracy.

Table 1: Values selected for pole-limited DRBEM testing.

Total elements	Total non-follicle elements	Time taken (mins)
1,000	159	0.4
1,200	359	0.5
1,500	659	0.6
2,000	1,159	0.9
5,000	4,159	8.1
10,000	9,159	33
12,000	11,159	94
15,000	14,159	869
20,000	19,159	145
28,761	27,920	296

Table 2: Mean squared error (MSE) for each selected internal pole value.

Total elements	MSE
1,000	0.000552
1,200	0.000508
1,500	0.000461
2,000	0.000294
5,000	0.000128
10,000	0.000037
12,000	0.000023
15,000	0.000012
20,000	0.000002

The optimal choice of internal pole count will be considered in a future paper, including the approach through which an appropriate value may be selected.

2.4 Computational results

Figure 3 outlines the time taken for each parallelism technique, for a range of internal pole counts. Prior to any optimisation or parallelisation, the DRBEM algorithm required ~ 78 days to complete processing for 29,000 elements. Optimised and parallelised, the model can be run for 10,000 internal poles in 33 min on a threaded CPU, 28 min on a GPU, or 9 min on a cluster.

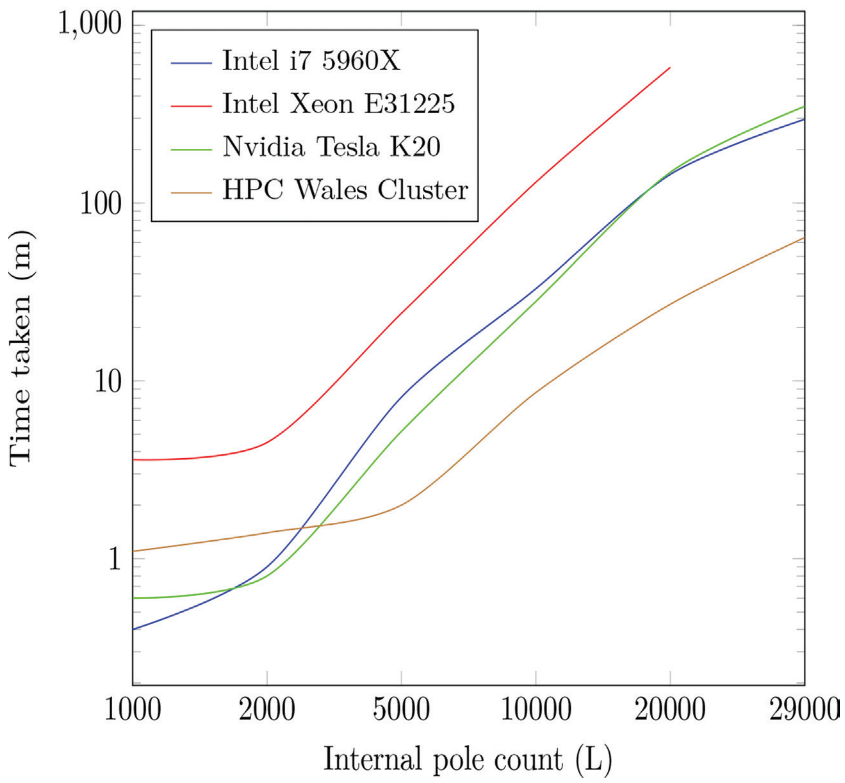


Figure 3: Time taken to run the dual reciprocity boundary element method implementation for high-end CPUs, GPUs and Clusters.

Similar optimisation of the TDBEM has been partially undertaken. Initial results are extremely promising, using the same techniques applied in this paper. Without modifying the internal pole count, an improvement in processing speed occurred, from ~ 6 h to ~ 7 min, on the same hardware, using threading. An investigation into the improvement in speed which can be achieved through modifying the internal pole count, and the resultant impact on accuracy, is the subject of future work.

3 CONCLUSIONS

Optimisation of the DRBEM has yielded a significant improvement in computational performance. Prior to optimisation, the DRBEM algorithm required ~ 78 days to complete calculation for the problem considered. By making a very minor sacrifice to accuracy, and through the application of other optimisation and parallelisation techniques, this calculation may be completed through the optimised model in ~ 28 min for the same hardware.

Similar success has been indicated for the TDBEM, however an investigation into the impact of modifying the internal pole count has not yet been conducted. Nonetheless, through optimisation and parallelism, a ~ 6 h calculation has been reduced to ~ 7 min. Consideration of the selection of an appropriate internal pole count value without the need for a parametric study has not been discussed here; these investigations will form the basis of future work.

REFERENCE

[1] Donne, K.E., Marotin, A. & Al-Hussany, A., Modified dual reciprocity boundary element modeling of collimated light fluence distribution in normal and cancerous prostate tissue during photodynamic therapy. In *34th International Conference on Boundary Elements and Other Mesh Reduction Methods,* 2012.

[2] Brebbia, C.A., Telles, J.C.F. & Wrobel, L.C., *Boundary Element Techniques - Theory and Applications in Engineering,* ed. C.A. Brebbia, Springer, 1984.

[3] Susan Blackford. *LAPack Naming Scheme*, Online. Oct. 1999, available at: URL: http://www.netlib.org/lapack/lug/node24.html.

[4] Hasan, M.R. & Whaley, R.C., Effectively exploiting parallel scale for all problem sizes in LU factorization. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 1039–1048, 2014.
http://dx.doi.org/10.1109/ipdps.2014.109

[5] Landaverde, R., Zhang, T., Coskun, A.K. & Herbordt, M., An investigation of unified memory access performance in CUDA. In *High Performance Extreme Computing Conference (HPEC), IEEE,* pp. 1–6, 2014.
http://dx.doi.org/10.1109/hpec.2014.7040988

[6] Asaduzzaman, A., Gummadi, D. & Yip, C.M., A talented CPU-to-GPU memory mapping technique. In *Southeastcon, IEEE,* pp. 1–6, 2014.
http://dx.doi.org/10.1109/secon.2014.6950676

[7] Sterling, T.L., *Beowulf Cluster Computing With Linux,* Mit Press, 2002.

[8] Yang, L.T. & Guo, M., *High-Performance Computing: Paradigm and Infrastructure,* Wiley-Interscience, 2006.

[9] Tibayrenc, M., *Genetics and Evolution of Infectious Diseases,* Elsevier Science, 2010.