University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Masters Theses 1911 - February 2014

2011

A Banded Spike Algorithm and Solver for Shared Memory Architectures

Karan Mendiratta
University of Massachusetts Amherst

Follow this and additional works at: https://scholarworks.umass.edu/theses

Mendiratta, Karan, "A Banded Spike Algorithm and Solver for Shared Memory Architectures" (2011). *Masters Theses 1911 - February 2014*. 699.
Retrieved from https://scholarworks.umass.edu/theses/699

This thesis is brought to you for free and open access by ScholarWorks@UMass Amherst. It has been accepted for inclusion in Masters Theses 1911 - February 2014 by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

A BANDED SPIKE ALGORITHM AND SOLVER FOR SHARED MEMORY ARCHITECTURES

A Thesis Presented

by

KARAN MENDIRATTA

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2011

Electrical & Computer Engineering

A BANDED SPIKE ALGORITHM AND SOLVER FOR SHARED MEMORY ARCHITECTURES

A Thesis Presented

by

KARAN MENDIRATTA

Approved as to style and content by:

____

Eric Polizzi, Chair

____

David P Schmidt, Member

____

Michael Zink, Member

____

Christopher V Hollot, Department Chair
Electrical & Computer Engineering

For Krishna Babbar, and Baldev Raj Mendiratta

// ACKNOWLEDGMENTS

I would like to thank my advisor and mentor Prof. Eric Polizzi for his guidance, patience and support. I would also like to thank Prof. David P Schmidt and Prof. Michael Zink for agreeing to be a part of my thesis committee. Last but not the least, I am grateful to my family and friends for always encouraging and supporting me in all my endeavors.

ABSTRACT

A BANDED SPIKE ALGORITHM AND SOLVER FOR SHARED MEMORY ARCHITECTURES

SEPTEMBER 2011

KARAN MENDIRATTA

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Eric Polizzi

A new parallel solver based on SPIKE-TA algorithm has been developed using OpenMP API for solving diagonally-dominant banded linear systems on shared memory architectures. The results of the numerical experiments carried out for different test cases demonstrate high-performance and scalability on current multi-core platforms and highlight the time savings that SPIKE-TA OpenMP offers in comparison to the LAPACK BLAS-threaded LU model. By exploiting algorithmic parallelism in addition to threaded implementation, we obtain greater speed-ups in contrast to the threaded versions of sequential algorithms. For non-diagonally dominant systems, we implement the SPIKE-RL scheme and a new Spike-calling-Spike (SCS) scheme using OpenMP. The timing results for solving the non-diagonally dominant systems using SPIKE-RL show extremely good scaling in comparison to LAPACK and modified banded-primitive library.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . . **iv**

ABSTRACT . . . . **v**

LIST OF TABLES . . . . **viii**

LIST OF FIGURES . . . . **ix**

CHAPTER

1. AN INTRODUCTION TO SPIKE ALGORITHM **1**

1.1 Motivation 1
1.2 Introduction 2
1.3 The Algorithm 3
1.4 SPIKE Variants 8

2. OPENMP AND MPI **12**

2.1 OpenMP 12
2.2 MPI 14
2.3 OpenMP vs MPI 15

3. SPIKE OPENMP FOR DIAGONALLY-DOMINANT SYSTEMS **17**

3.1 The TA-SPIKE algorithm 17

3.1.1 Truncated SPIKE 17
3.1.2 **LU/UL** strategy 18
3.1.3 Partitioning for TA scheme 19
3.1.4 Error Analysis 22
3.1.5 Theorems 25

3.2 SPIKE-TA OpenMP 25
3.3 Results and Discussion 26

4. SPIKE OPENMP FOR NON-DIAGONALLY DOMINANT SYSTEMS **32**

4.1 Schemes 32
4.2 Recursive SPIKE 33

4.2.1 Two partitions case 33
4.2.2 Multiple partitions case 35

4.3 Implementation 39

4.3.1 Load-balancing 41
4.3.2 Retrieval 42

4.4 Results and Discussion 42

5. CONCLUSION **49**

BIBLIOGRAPHY **51**

LIST OF TABLES

Table **Page**

2.1 OpenMP vs MPI 16

3.1 Three sets of experiments are used to illustrate the performances and scalability of MKL-LAPACK and SPIKE-OpenMP for solving real and double precision diagonally dominant systems. Both solvers are running on a Intel Nehalem node $X5550$ featuring eight cores running at $2.66Ghz$, with $48Gb$ total memory. The accuracy results on the residuals obtained by both solvers, not reported here, are comparable (accuracy machine). 28

3.2 Two sets of experiments are used to compare the performances and scalability of SPIKE-OpenMP and SPIKE-MPI, truncated implementations for solving real and double precision diagonally dominant systems. 29

4.1 SCS speed-up on 4-processors 33

4.2 Scalability comparison, LAPACK vs SPIKE-RL OpenMP: Three sets of experiments are used to illustrate the performances and scalability of MKL-LAPACK and SPIKE-RL OpenMP for solving real and double precision non-diagonally dominant systems. Both solvers are running on a Intel Nehalem node X5550 featuring eight cores running at 2.66Ghz, with 48Gb total memory. The accuracy results on the residuals obtained by both solvers are comparable of the order of $\sim 10^{-10}$, and no diagonal-boosting is observed. 43

4.3 Scalability comparison, Banded Primitive vs SPIKE-RL OpenMP: The same three sets of experiments, on a non-diagonally dominant real double-precision system matrix, carried out on the same machine, are used to illustrate their performance and scalability 45

4.4 Total(Fact + Solve) time in seconds for different partition-size ratios for N=640,000; B=200 and B=500; # rhs=1, for two, four and eight threads. 47

LIST OF FIGURES

Figure **Page**

1.1 Table showing different variants of the polyalgorithm SPIKE 10

3.1 The bottom of the spike V_j can be computed using only the bottom $m \times m$ blocks of L and U. Similarly, the top of the spike W_j may be obtained if one performs the UL-factorization. 19

3.2 Illustration of the unconventional partitioning of the linear system for the TA-SPIKE algorithm in the case of 4 processors/cores. A_1 is sent to processor 1, A_2 to processor 2 and 4, and A_3 to processor 3. 20

3.3 Illustration of the unconventional partitioning of the linear system for the truncated SPIKE algorithm in the case of 8 processors/cores. A_1 is sent to processor 1, A_5 to processor 5, and A_i for $i = 2, \ldots, 4$ to processor i and $i + 4$. 20

3.4 The bottom of the modified right hand side G_j can be computed using an entire forward sweep on L follows by and a very small fraction of backward sweep on U. 21

3.5 This figures shows the graphical representation of the speed-up of SPIKE-TA OpenMP, varying N with fixed b and #rhs 30

3.6 This figures shows the graphical representation of the speed-up of SPIKE-TA MPI, varying N with fixed b and #rhs, qualitatively similar to OpenMP 31

4.1 The spike matrix of the new reduced system for p=4 36

4.2 Figure showing the factorization step, and forming reduced systems at different levels of recursion for SPIKE-RL 40

4.3 Total time in seconds for, $N = 640,000, b = 200$, and $\#rhs = 1$, on two, four and eight-threads. Total Time for two threads = 1.551 sec. Fastest run (smallest time) on both 4 and 8-threads is at value(ratio) 3.2 45

4.4 Total time in seconds for, $N = 640,000, b = 500$, and $\#rhs = 1$, on two, four and eight-threads. Total Time for two threads = 6.054 sec. Fastest run (smallest time) on both 4 and 8-threads is at value(ratio) 3.4 . . . 46

4.5 **Solve time** in seconds for, $N = 640,000, b = 200$, and $\#rhs = 1$, on two, four and eight-threads. Solve Time for two threads = 0.181 sec. . . . 46

CHAPTER 1

AN INTRODUCTION TO SPIKE ALGORITHM

1.1 Motivation

Linear systems arise in different walks of science and engineering applications such as computational mechanics (fluids, structures, and fluid-structure interactions), computational nanoelectronics (quantum mechanical simulations), and even in the field of financial mathematics (random walks and geometric brownian motion). Certain applications of specialized types, such as the "transport problems" in nanowires and nanotubes, and "traffic flow problems" inherently requires solving banded systems. Most applications often give rise to very large sparse linear systems that can be reordered to produce either narrow banded systems or low-rank perturbations of narrow banded systems, with the systems being either dense or sparse within the band. For example, in finite-element analysis, the underlying sparse linear systems can be reordered to result in a banded system in which the width of the band is but a small fraction of the size of the overall problem. In turn, specific techniques can be used to further reduce the bandwidth and create a robust preconditioner for an iterative solver.

Traditional numerical algorithms and library packages like LAPACK are yet facing new challenges for addressing the current large-scale simulation needs for ever higher level of efficiency, accuracy, and scalability in modern parallel architectures. Thus developing robust banded parallel solvers that are efficient on both parallel high-end architectures, and low-cost clusters is of great importance. Fast and efficient banded solvers, as mentioned previously, are also required to function as preconditioners for

iterative methods for solving linear systems, and/or large eigenvalue problems.

From an algorithmic and implementation point of view for developing a scalable, fast and robust banded solver, a lot of factors come into play; in particular, the complexity of the system, the data locality, arithmetic considerations (Real vs Complex, Single vs Double precision), architectural considerations:- shared-memory(OpenMP), distributed memory(MPI), GPU, Cloud and various other high-end computing platforms.

While parallel distributed numerical packages do offer HPC users valuable tools for solving large scale problems, the growing size of the number of cores in a compute node forestalls distributed programming model (i.e. MPI) for many users. We have therefore been working on implementing a banded solver for shared-memory architectures, as we want to maintain the same scalability (in comparison to the MPI version), but at the same time enable the end-user to use the same solver for both sequential or parallel execution. We ensure the ease-of-usability, by hiding all the preprocessing steps from the user and allowing him to just specify the number of processors he wants to use for execution at run-time, unlike the MPI case where the user is responsible for distributing the system matrix on different processors before hand.

1.2 Introduction

The SPIKE algorithm is based on a divide and conquer design paradigm, involving the following steps:

(a) *Pre-processing:*

- (i) partitioning of the original system on different processors, or SMPs
- (ii) factorization of each diagonal block and extraction of a reduced system of much smaller size;

(b) *Post-processing:*

   (iii) solving the reduced system, and

   (iv) retrieving the overall solution. SPIKE has several built-in options that range from using it as a pure direct solver to using it for producing various preconditioners for any outer iterative scheme.

The primary advantage of SPIKE algorithm's divide-and-conquer design is that it makes it naturally adapted for execution in multi-processor machines, because distinct sub-problems can be executed on different processors. In addition, for shared-memory systems especially, the communication of data between processors does not need to be planned in advance. Also, such a design tends to make efficient use of memory caches, thus overall rendering the algorithm an inherent additional parallel dimension.

1.3 The Algorithm

The SPIKE algorithm is designed to solve banded systems on a parallel machine. The basic idea was introduced by Sameh and Kuck[7] who considered the tridiagonal case and Chen, Kuck, and Sameh[3] who studied the triangular case. Lawrie[12] and Sameh applied the algorithm to the symmetric positive definite systems, while Dongarra and Sameh[10] considered the strictly diagonally dominant case. Variations of the SPIKE algorithms for tridiagonal systems were introduced by Sun, Zhang, and Ni[24], who also analyzed the truncation error for tridiagonal systems which are evenly diagonally dominant. The truncation error for tridiagonal Toeplitz systems, which are also strictly diagonally dominant, as well as symmetric or skew symmetric was considered by Sun[25]. Another variation of the SPIKE algorithm for strictly diagonally dominant systems was studied by Larriba-Pey, Jorba, and Navarro[11]. Polizzi and Sameh[17] have extended the SPIKE algorithms to the general banded case, and they developed the SPIKE package.

A matrix $A = [a_{ij}]$ is diagonally dominant by rows if,

$$\sum_{i \neq j} |a_{ij}| \leq |a_{ii}| \tag{1.1}$$

for all i. If the inequality is sharp, then A is *strictly* diagonally dominant by rows.

If A is non-singular and diagonally dominant by rows, then the diagonal entries are non-zero and the dominance factor is defined as follows:

$$\epsilon = max \frac{\sum_{i \neq j} |a_{ij}|}{|a_{ii}|} \tag{1.2}$$

If $\epsilon > 0$, then the degree of diagonal dominance d is given by,

$$d = \epsilon^{-1} \tag{1.3}$$

The degree of diagonal dominance is central to the analysis of the truncated SPIKE algorithm which will be explained in detail in Chapter 3. For illustrating the underlying concepts and working of SPIKE algorithm, we consider the non-singular linear system,

$$Ax = f \tag{1.4}$$

where A is a n *by* n banded matrix, and the number of super-diagonals k is assumed to be equal to the number of sub-diagonals and that the matrix is narrow banded, i.e., $k << n$. Let p denote the number of processors. For simplicity it is assumed that p divides n. Let the system be partitioned into the block diagonal form shown below,

$$
Ax = \begin{pmatrix} A_1 & B_1 & & & \\ C_2 & A_2 & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & B_{p-1} \\ & & & C_p & A_p \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_p \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ \vdots \\ f_p \end{pmatrix} \tag{1.5}
$$

where $A_i, i = 1, 2, .., p$ is a banded matrix of order $\mu = n/p$ and bandwidth $2k+1$, B_i and C_i are matrices of size $\mu \times k$.

$$
B_i = \begin{pmatrix} 0 & 0 \\ b_i & 0 \end{pmatrix}, \; C_{i+1} = \begin{pmatrix} 0 & c_{i+1} \\ 0 & 0 \end{pmatrix}, \; i = 1, 2, ..., p-1 \tag{1.6}
$$

where b_i and $c_i + 1$ are triangular matrices of size $k \times k$. Let D denote the main block diagonal, i.e.,

$$
D = diag\{A_1, A_2, .., A_p\} \tag{1.7}
$$

If both sides of (1.5) are Pre-multiplied by D^{-1}, we obtain a reduced linear system of the form $Sx = g$, as shown below:

$$
\begin{pmatrix} I_\mu & V_1 & & & \\ W_2 & I_\mu & V_2 & & \\ & \ddots & \ddots & \ddots & \\ & & W_{p-1} & I_\mu & V_{p-1} \\ & & & W_p & I_\mu \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{p-1} \\ x_p \end{pmatrix} = \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_{p-1} \\ g_p \end{pmatrix} \tag{1.8}
$$

in which V_i denotes the right spikes and W_i denotes the left spikes, with k columns given by,

$$V_i = A_i^{-1} \begin{pmatrix} 0 \\ B_i \end{pmatrix}, \text{ and } W_i = A_i^{-1} \begin{pmatrix} C_i \\ 0 \end{pmatrix} \tag{1.9}$$

and V_i and W_i are partitioned as,

$$V_i = \begin{pmatrix} V_i^{(t)} \\ V_i^{(m)} \\ V_i^{(b)} \end{pmatrix}, \text{ and } W_i = \begin{pmatrix} W_i^{(t)} \\ W_i^{(m)} \\ W_i^{(b)} \end{pmatrix} \tag{1.10}$$

where $V_i^{(t)}$, $V_i^{(b)}$, $W_i^{(t)}$, and $W_i^{(b)}$ are the tips of the spike of size $k \times k$ The superscripts *t,m,* and b are abbreviations of the words top, middle, and bottom, respectively. Also, x_i and g_i are partitioned conformally as,

$$x_i = \begin{pmatrix} x_i^{(t)} \\ x_i^{(m)} \\ x_i^{(b)} \end{pmatrix}, \text{ and } g_i = \begin{pmatrix} g_i^{(t)} \\ g_i^{(m)} \\ g_i^{(b)} \end{pmatrix} \tag{1.11}$$

As an illustration, the system above for $p = 3$ is shown below,

$$\begin{pmatrix} I_k & & & V_1^{(t)} & & & & & \\ & I_v & & V_1^{(m)} & & & & & \\ & & I_k & V_1^{(b)} & & & & & \\ & & W_2^{(t)} & I_k & & & V_2^{(t)} & & \\ & & W_2^{(m)} & & I_v & & V_2^{(m)} & & \\ & & W_2^{(b)} & & & I_k & V_2^{(b)} & & \\ & & & & & W_3^{(t)} & I_k & & \\ & & & & & W_3^{(m)} & & I_v & \\ & & & & & W_3^{(t)} & & & I_k \end{pmatrix} \begin{pmatrix} x_1^{(t)} \\ x_1^{(m)} \\ x_1^{(b)} \\ x_2^{(t)} \\ x_2^{(m)} \\ x_2^{(b)} \\ x_3^{(t)} \\ x_3^{(m)} \\ x_3^{(b)} \end{pmatrix} = \begin{pmatrix} f_1^{(t)} \\ f_1^{(m)} \\ f_1^{(b)} \\ f_2^{(t)} \\ f_2^{(m)} \\ f_2^{(b)} \\ f_3^{(t)} \\ f_3^{(m)} \\ f_3^{(b)} \end{pmatrix} \tag{1.12}$$

where $\upsilon = \mu - 2k$.

It is from these narrow block columns or spikes extending from the main diagonal that the algorithm has derived its name. The matrix S is referred to the SPIKE matrix.

Observe that the union of the k equations above and the k equations below the $p-1$ partition lines forms an independent subsystem of order $2k(p-1)$. This is referred to as the "reduced" system and written as, $Rx_r = g_r$. It has the form

$$Ax = \begin{pmatrix} E_1 & F_1 & & & \\ G_2 & E_2 & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & F_{p-2} \\ & & & G_{p-1} & E_{p-1} \end{pmatrix} \begin{pmatrix} x_{r,1} \\ x_{r,2} \\ \vdots \\ x_{r,p-1} \end{pmatrix} = \begin{pmatrix} g_{r,1} \\ g_{r,2} \\ \vdots \\ g_{r,p-1} \end{pmatrix} \quad (1.13)$$

where,

$$E_i = \begin{pmatrix} I_k & V_i^{(b)} \\ W_{i+1}^{(t)} & I_k \end{pmatrix}, \quad F_i = \begin{pmatrix} 0 & 0 \\ V_{i+1}^{(t)} & 0 \end{pmatrix}, \quad \text{and } G_{i+1} = \begin{pmatrix} 0 & W_i^{(b)} \\ 0 & 0 \end{pmatrix}, \quad (1.14)$$

and,

$$x_{r,i} = \begin{pmatrix} x_i^{(b)} \\ x_{i+1}^{(t)} \end{pmatrix}, \quad \text{and } g_{r,i} = \begin{pmatrix} g_i^{(b)} \\ g_{i+1}^{(t)} \end{pmatrix} \quad (1.15)$$

The subscript r is an abbreviation of the word "reduced" and $i = 1, \ldots, p-1$ Once the reduced system has been solved to obtain the partial solution, and as the as the system is largely decoupled, the retrieval can be carried out in perfect parallelism using:

$$x_1 = g_1 - V_1 x_2^{(t)} \quad (1.16)$$

$$x_i = g_i - W_i x_{i-1}^{(b)} - V_i x_{i+1}^{(t)}, \qquad i = 2, \ldots, p-1 \tag{1.17}$$

$$x_p = g_p - W_p x_{p-1}^{(b)} \tag{1.18}$$

It should also be noted that x_0, x_{p+1}, W_1 , and V_p are undefined and are assumed zero. If the calculations are carried out using exact arithmetic, then the above retrieval gives the exact solution of $Ax = f$.

Thus to summarize, the SPIKE algorithm can be broken down into the following steps:

1. Factorize independently the diagonal blocks of A. This is much more efficient than factorizating A directly.

2. Compute the spikes using the factorization obtained in the previous step and compute the modified right hand side.

3. Form and solve the reduced system to obtain the partial solution above and below each partition.

4. Retrieve the rest of solution of x.

1.4 SPIKE Variants

The original SPIKE algorithm explained in the previous has many variants. These variants target systems of equations with certain properties in order to reduce the amount of computation performed. They also increase the amount of parallelism available during different stages of the algorithm.

SPIKE presents different computation options depending on the properties and type of the matrix and the platform architectures.

1. SPIKES can be computed:

   (a) Explicitly

   (b) On-the-fly

   (c) Approximately

2. The diagonal blocks can be solved:

   (a) Directly (PLU,LU,UL etc.)

   (b) Iteratively

3. The reduced system can be solved

   (a) Directly (Recursive SPIKE)

   (b) Iteratively along with a preconditioning scheme

   (c) Approximately (Truncated SPIKE)

The figure 1.1 below illustrates and summarizes different versions of SPIKE algorithms under different factorization schemes.

My focus for implementation and analysis for this thesis is on the variant that use a truncated scheme to solve the reduced system. The truncated scheme is useful for systems that are diagonally dominant. In diagonally dominant systems, the values in the spikes far from the diagonal are likely to be very close to zero and therefore contribute little to the solution. Consequently, the truncated scheme treats these values as zero and only computes the $k \times k$ portion of the spikes close to the diagonal, specifically, $V^{(b)}$ and $W^{(t)}$. This is accomplished by either using the LU or UL factorization computed for the blocks of the diagonal. Since the matrix is diagonally dominant, the LU-factorization of each block A_j can be performed without pivoting, using for example the modified LAPACK routine 'X'DBTRF proposed in ScaLAPACK. The truncated approach also consists of performing a UL-factorization without pivoting. Similar to the LU-factorization, this allows obtaining the top block of W_j involving

Algorithm / *Factorization*	**E** **Explicit**	**R** **Recursive**	**T** **Truncated**	**F** **On-the-fly**
P : LU w/ pivoting	Explicit generation of spikes, reduced system is solved iteratively with a pre-conditioner.	Explicit generation of spikes, reduced system is solved directly using recursive SPIKE	--	Implicit generation of reduced system which is solved on-the-fly using an iterative method.
L : LU w/o pivoting	Explicit generation of spikes, reduced system is solved iteratively with a pre-conditioner.	Explicit generation of spikes, reduced system is solved directly using recursive SPIKE	Truncated generation of spike tips: V_b is exact, W^t is *approx.* and reduced system is solved directly.	Implicit generation of reduced system which is solved on-the-fly using an iterative method.
U : LU and UL w/o pivoting	–	--	Truncated generation of spike tips: V_b , W^t are *exact* and reduced system is solved directly.	Implicit generation of reduced system which is solved on-the-fly using an iterative method with *pre-conditioner.*
A: Alternate LU/UL	Explicit generation of spikes using *new partitioning,* reduced system is solved iteratively with a pre-conditioner.	--	Truncated generation of spikes using *new partitioning*, reduced system is solved directly.	--

Figure 1.1. Table showing different variants of the polyalgorithm SPIKE

only the top $k \times k$ blocks of the new U and L. The numerical experiments indicate that the time consumed by this LU/UL strategy, is much less than that taken by performing only one LU factorization per diagonal block and generating the entire left spikes. Using a permutation of the rows and columns of each diagonal block, we can use the same LAPACK routines to obtain the UL-factorization. Also, an alternative to performing a UL-factorization of the block A_j could be to approximate the top of the left spike,$W_i^{(t)}$,of order k, by inverting only an $l \times l$ $(l > m)$ top diagonal block (left top corner) of the banded block A_i. Typically, we choose $l = 2k$ to get a suitable approximation of the $k \times k$ left top corner of the inverse of A_i. The quality of this approximation depends on the degree of diagonal dominance.

LU factorization of A_i is used to solve the bottom tips, V_i , of the spikes and the UL factorization of A_i is used to solve for the top tips, W_i , of the spikes. Polizzi and Sameh [17] found experimentally that it is faster to extract the truncated reduced system using LU/UL combinations on the machines were arithmetic operations require much less time than memory references. The LU/UL strategy also has a greater data locality and computing LU/UL factorizations is a BLAS-3 operation, which is highly optimized.

CHAPTER 2

OPENMP AND MPI

2.1 OpenMP

OpenMP is a new API for multi-platform shared-memory programming on UNIX and Microsoft Windows NT platforms. OpenMP provides comment-line directives, embedded in C/C++ or Fortran source code, for

1. scoping data
2. specifying work load
3. synchronization of threads

OpenMP provides function calls for obtaining information about threads.
e.g., $omp_num_threads(), omp_get_thread_num()$ The main idea behind OpenMP is using a new form of programming formalism that is built under the blanket of an :-

1. An existing sequential language modified to handle parallelism like C/C++ and Fortran
2. A parallelizing compiler
3. Library routines/compiler directives compatible with the sequential language

Under OpenMP Shared Memory Parallelization, all processors can access all the memory in the parallel system (one address space). The time to access the memory may not be equal for all processors implying that the memory might not necessarily be flat. Also, Parallelizing on a SMP(symmetric multiprocessing) does not reduce CPU

time , it rather it reduces wall clock time. Parallel execution is achieved by generating multiple threads which execute in parallel. Number of threads (in principle) is independent of the number of processors. In the following sections, we are going to elaborate on the basics of SMP parallelization, threads and variable scoping in an OpenMP environment.

SMP Parallelization

Symmetric Multiprocessing or SMP involves a multiprocessor computer hardware architecture where two or more identical processors are connected to a single shared main memory and are controlled by a single OS instance. Most common multiprocessor systems today use an SMP architecture. For multi-core processors, the SMP architecture applies to the cores, treating them as separate processors, which may be interconnected using buses, crossbar switches or on-chip mesh networks. SMP finds many uses in science, industry, and business which often use custom-programmed software for multi-threaded processing. The basic concepts behind SMP parallelism will be discussed in detail in the following sections.

Threads

The threads under SMP constructs are not full UNIX processes. They are lightweight, independent "collections of instructions" that execute within a UNIX process.

1. All threads created by the same process share the same address space.It can be considered both as a blessing and a curse due to the fact that "inter-thread" communication is efficient, but it is easy to stomp on memory and create race conditions.

2. Because they are lightweight, they are (relatively) inexpensive to create and destroy exploiting the fact that creation of a thread can take three orders of magnitude less time than process creation.

3. Threads can be created and assigned to multiple processors.

SMP Parallelism and OpenMP Threads

Using a parallelizing compiler and its directives, we can generate pthreads using industry-standard directives (e.g. !$OMP etc.)

1. All OpenMP programs begin as a single process: the master thread
2. FORK: the master thread then creates a team of parallel threads.
3. Parallel region statements executed in parallel among the various team threads.
4. JOIN: threads synchronize and terminate, leaving only the master thread.

Variable Scoping

The most difficult part of shared-memory parallelization is to ascertain the following:

1. The memory that is going to be shared.
2. The memory that would be private (i.e. each processor will have its own copy)
3. And how the private memory is going to be treated vis-â-vis the global address space.

Variables are shared by default, except for loop index in parallel *do loop*. It also must mesh with the Fortran view of memory. The variables broadly speaking can be classified as:

• Global: shared by all routines

• Local: local to a given routine and saved vs. non-saved variables.

2.2 MPI

Message Passing Interface (MPI) is an API specification that allows processes to communicate with one another by sending and receiving messages. It is typically used

to implement parallel programs that are intended for execution on high performance clusters and supercomputers, where the cost of accessing non-local memory is high. MPI controls its own internal data structures i.e. supports heterogeneity.

A MPI communicator is a handle representing a group of processes that can communicate with each other. All MPI communication calls have a communicator argument like MPI_COMM_WORLD which is defined when an MPI_INIT call is made. The calling sequence to create a parallel program in MPI is described in the next section.

Using MPI

1. Initializing MPI:

   MPI_INIT must be the first MPI routine called (only once)

2. Process Rank:

   It is used to identify the source and destination of message

   (a) There is a Process ID number within the communicator that starts with zero and goes to $n-1$, where n is the number of processes requested

3. Exiting MPI:

   No calls to MPI routines after finalization. For example the Fortran call, CALL *MPI_FINALIZE* tells the compiler to exit the distributed environment. The number of processes running after this routine is called is undefined.

2.3 OpenMP vs MPI

The table belows summarizes the key differences between OpenMP and MPI:

OpenMP	MPI
1. Platform specific, i.e. Only for shared memory-architectures	1. Platform independent, i.e. Portable to all platforms
2. Easy to incrementally parallelize but more difficult to write highly scalable programs	2. Highly distributed, i.e. everything needs to be parallelized or nothing
3. Small API based on compiler directives and limited library routines	3. API based on vast collection of library routines
4. Same program can be used for sequential and parallel execution	4. Possible but difficult to use same program for both serial and parallel execution
5. Global and local scoping i.e. Shared and private variables	5. Grossly local scoping as variables are local to each processor

Table 2.1. OpenMP vs MPI

CHAPTER 3

SPIKE OPENMP FOR DIAGONALLY-DOMINANT SYSTEMS

3.1 The TA-SPIKE algorithm

The TA-SPIKE algorithm described in [19] combines both the advantages of the **LU/UL** strategy of the truncated SPIKE scheme, and a new unconventional partitioning scheme to achieve an effective load balancing between processors/cores.

3.1.1 Truncated SPIKE

The truncated SPIKE scheme introduced in [16] is an optimized version of the SPIKE algorithm with enhanced use of parallelism for handling diagonally dominant systems. These systems may arise from several science and engineering applications, and are defined if the degree of diagonally dominance, d, of the matrix A is greater than one; where d at stated previously, is given by:

$$d = \min \frac{|A_{i,i}|}{\sum_{j \neq i} |A_{i,j}|}. \tag{3.1}$$

If this property is satisfied, one can show that the magnitude of the elements of the right spikes V_j would decay from bottom to top, while the elements of the left spikes W_j would decay in magnitude from top to bottom [21, 22]. Since the size of the diagonal blocks A_j is assumed much larger than the size m of the blocks B_j and C_j, the bottom blocks of the left spikes $W_j^{(b)}$ and the top blocks of the right spikes $V_j^{(t)}$ can be approximately set equal to zero. It follows that the resulting "truncated"

reduced system is simply block diagonal composed by $p-1$ independent $2m \times 2m$ block systems (P represents the number of partitions) of this form:

$$\begin{bmatrix} I_m & V_j^{(b)} \\ W_{j+1}^{(t)} & I_m \end{bmatrix} \begin{bmatrix} X_j^{(b)} \\ X_{j+1}^{(t)} \end{bmatrix} = \begin{bmatrix} G_j^{(b)} \\ G_{j+1}^{(t)} \end{bmatrix}, \quad j = 1, \ldots, p-1, \tag{3.2}$$

where X_j and G_j denote the jth partition of the solution X and the modified right hand side G. Therefore the block $G_j^{(b)}$ (resp. $X_j^{(b)}$) is associated to the bottom tip of G_j (resp. X_j), while the block $G_{j+1}^{(t)}$ (resp. $X_{j+1}^{(t)}$) is associated to the top tip of G_{j+1} (resp. X_{j+1}). The reduced linear systems are then decoupled and can be solved in parallel.

Within the framework of the truncated scheme, two other major contributions have also been proposed for improving computing performance and scalability of the factorization and solve stages:

(i) a **LU/UL** strategy, and

(ii) a new unconventional partitioning scheme.

3.1.2 LU/UL strategy

The **LU/UL** strategy can be used to avoid computing (generating) the entire spikes in order to obtain the tips V_j^b and W_{j+1}^t ($j = 1, \ldots, p-1$). As illustrated in Figure 3.1, computational solve times to obtain the bottom tip V_j^b can be drastically reduced by using the LU factorization without pivoting on each diagonal block $A_{j=1,\ldots,p-1}$. In turn, a UL factorization without pivoting on each diagonal block $A_{j=2,\ldots,P}$ can be used to obtain the tips W_j^t involving only the top $m \times m$ blocks of the new U and L matrices.

Finally, it should be noted that the two-partitions case can take advantage of a single LU or UL factorization without pivoting respectively for A_1 and A_2. The

(exact) resulting single block reduced system can then be obtained and solved with minimal efforts. As a result, the number of arithmetic operations of SPIKE using the **LU/UL** strategy for the two-partitions case and running on two processors/cores, is essentially divided by two as compared to the sequential LAPACK in the factorization and solve stages.

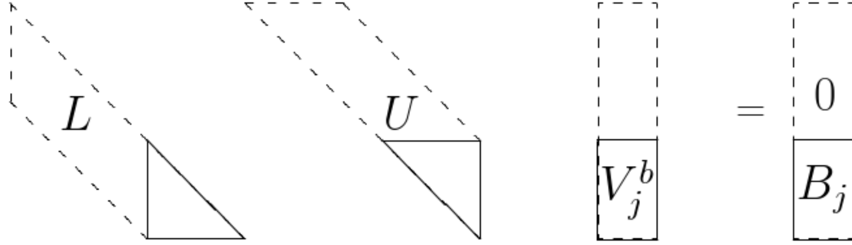

Figure 3.1. The bottom of the spike V_j can be computed using only the bottom $m \times m$ blocks of L and U. Similarly, the top of the spike W_j may be obtained if one performs the UL-factorization.

3.1.3 Partitioning for TA scheme

Most often, parallel algorithms which aim at achieving linear scalability on large number of processors/cores, inherit extensive preprocessing stages with increased memory references or arithmetic operations, leading to performance degradation on small number of processors/cores. For example, using the **LU/UL** strategy described above with a number of partitions greater than two, each middle partition $j = 2, \ldots, p-1$ has now to perform both LU and UL factorizations. In order to decrease the number of arithmetic operations, a new parallel distribution of the system matrix is here considered, which involves using less partitions than number of processors/cores k (i.e. $p < k$). In practice, the new number of partitions will be equal to $p = (k+2)/2$ (where k is an even number of processors/cores). Figures 3.2 and 3.3 illustrate the new partitioning of the matrix right hand side and solution, respectively for the cases $k = 4,\ p = 3$ and $k = 8,\ p = 5$. Within this new partitioning, the new block matrices A_j, $j = 1, \ldots, p$, are associated to the first p processors/cores, while

the rest of them going from $p+1$ to k hold another copy of the block matrix A_j, $j = 2, \ldots, p-1$.

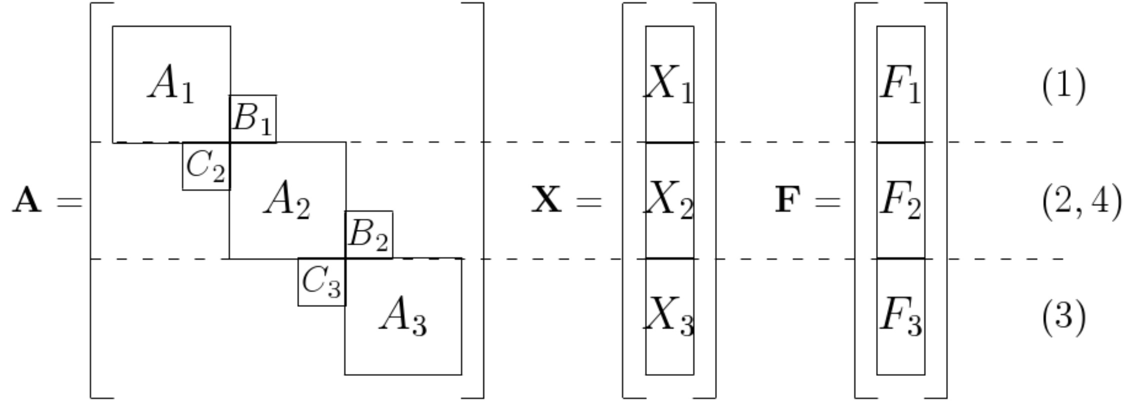

Figure 3.2. Illustration of the unconventional partitioning of the linear system for the TA-SPIKE algorithm in the case of 4 processors/cores. A_1 is sent to processor 1, A_2 to processor 2 and 4, and A_3 to processor 3.

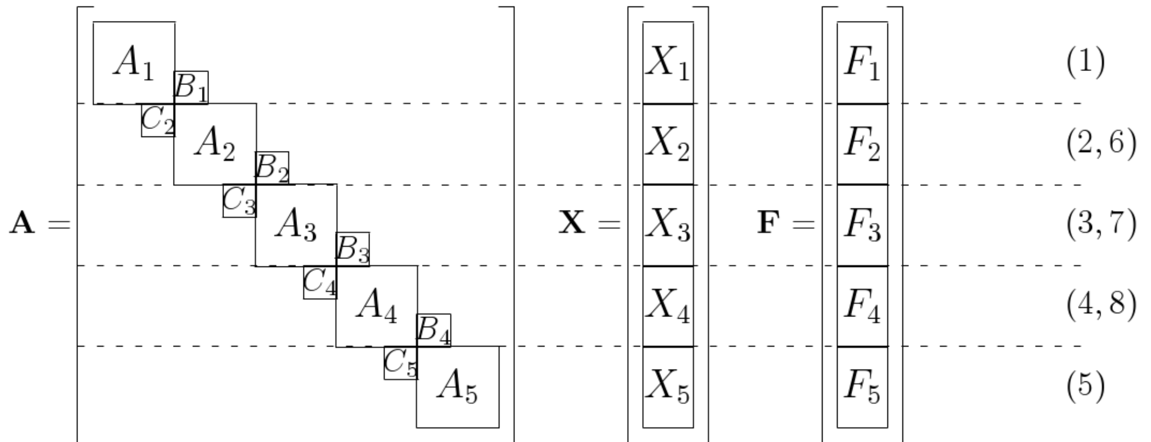

Figure 3.3. Illustration of the unconventional partitioning of the linear system for the truncated SPIKE algorithm in the case of 8 processors/cores. A_1 is sent to processor 1, A_5 to processor 5, and A_i for $i = 2, \ldots, 4$ to processor i and $i+4$.

As described above using the **LU/UL** strategy, the $V_j^{(b)}$ $(j = 1, \ldots, p-1)$ can be obtained here with minimal computational efforts via the LU solve step on processors/cores 1 to $p-1$, while the $W_j^{(t)}$ $(j = 2, \ldots, p)$ can be obtained in the similar way by performing a UL solve steps on processors/cores p to k. In the example of $k = 4$, we perform now independently the factorizations $L_jU_j \leftarrow A_j$ for partitions $j = 1, 2$,

and $\hat{U}_j\hat{L}_j \leftarrow A_j$ for partitions $j = 2, 3$. Using this new partitioning scheme the size of the partitions does increase but the number of arithmetic operations by partition decreases along with the size of the truncated reduced system. As compared to a sequential **LU** algorithm, the speed-up for the factorization stage of the TA-SPIKE scheme is then expected ideally equal to the new number of partitions, i.e. $2\times$ on two processors/cores, $3\times$ on four, $5\times$ on eight, etc.

Finally, it should be noted that the same speed-up performance is expected for the solve stage as well. It is indeed possible to compute the final solution using essentially only one forward sweep and one backward sweep by partitions (the costs for communications and for the solve of the truncated reduced system, are considered minimal). Figure 3.4 illustrates, in particular, that the G_j^b $j = 1, \ldots, p-1$ blocks which appears in the reduced system (3.2), can be generated mainly using only one forward sweep of the LU factorization on partition j.

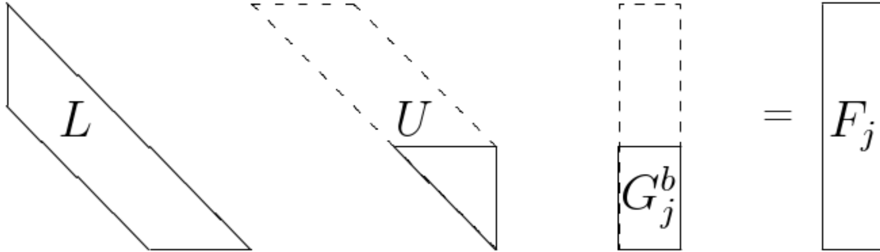

Figure 3.4. The bottom of the modified right hand side G_j can be computed using an entire forward sweep on L follows by and a very small fraction of backward sweep on U.

Similarly, the generation of G_{j+1}^t $j = 1, \ldots, p-1$ can be obtained independently using only one entire backward sweep of the UL factorization on partition $j+1$. Once the reduced system solved (with minimal costs), the entire solution can be retrieved as follows:

$$A_j\ X_j = F_j - \begin{bmatrix} 0 \\ I_m \end{bmatrix} B_j X_{j+1}^{(t)} - \begin{bmatrix} I_m \\ 0 \end{bmatrix} C_j X_{j-1}^{(b)}, \quad j = 2, \ldots, p-1, \tag{3.3}$$

where the third term (resp. the second term) of the right hand side is absent for the case $j = 1$ (resp. $j = p$). Since both LU and UL factorizations are available for A_j i.e. $A_j = L_j U_j$ and $A_j = \hat{U}_j \hat{L}_j$ (within the TA-scheme these factorizations have been assigned to different processors/cores), one can rewrite the above equation as follows:

$$X_j = U_j^{-1} \left\{ L_j^{-1} F_j - L_j^{-1} \begin{bmatrix} 0 \\ I_m \end{bmatrix} B_j X_{j+1}^{(t)} \right\} - \hat{L}_j^{-1} \left\{ \hat{U}_j^{-1} \begin{bmatrix} I_m \\ 0 \end{bmatrix} C_j X_{j-1}^{(b)} \right\}. \tag{3.4}$$

As a result, the first term of the right hand side mainly involves performing only one entire backward sweep on processors $1, \ldots, p-1$. Indeed, the forward sweep for obtaining $L_j^{-1} F_j$ has already been performed as a necessary step for generating the tips of the modified right side G_j (see Figure 3.4), while the second term in the expression would involve only a very small fraction of forward sweep. Similarly, the second term of the right hand side involves mainly performing only one entire forward sweep on processors $p, \ldots, k$ (only a very small fraction of backward sweep is also needed here).

3.1.4 Error Analysis

In this section a theoretical error analysis of the truncated SPIKE algorithm, carried out by Carl Mikkelsen and Murat Manguoglu [22] is summarized.

Let u denote the unit roundoff error on the machine, and following Higham [14], the authors define,

$$\gamma_j = ju/1 - ju \tag{3.5}$$

when $ju < 1$.

if the unit roundoff error is sufficiently small, they define

$$\alpha = \gamma_{3k+2} (\frac{d+1}{d-1})^2 < 1 \tag{3.6}$$

1. Stage 1:

   Each matrix A_i has dimension μ and is strictly diagonally dominant by rows. The computed LU factorization satisfies:

$$A + \triangle A_j = \hat{L}_i \hat{U}_i, |\triangle A_i| \leq \gamma_{3k+2} |\hat{L}_i||\hat{U}_i|, \tag{3.7}$$

   where,

$$\| \, |\hat{L}_i||\hat{U}_i| \, \|_\infty \leq \frac{d+1}{d-1} \| A \|_\infty \tag{3.8}$$

   when the unit roundoff error u is sufficiently small. The same type of estimate is obtained for the computed UL factorization.

2. Stage 2:

   In the truncated SPIKE algorithm, the entire SPIKE matrix is not computed but substituting is stopped as soon as the truncated reduced system matrix has been computed. However, in order to estimate the error, it is convenient to consider the computation of the entire SPIKE matrix S.

   Since $\hat{T} - T$ is a sub-matrix of $\hat{S} - S$, following accuracy bound is obtained:

$$\| \hat{T} - T \|_\infty \leq \| \hat{S} - S \|_\infty \leq \frac{2\alpha}{1-\alpha} \| S - I \|_\infty \leq \frac{2\alpha}{1-\alpha} \tag{3.9}$$

3. Stage 3:

   By Theorem 2, the truncated reduced system is a good approximation of the reduced system if d is not too close to 1 and if the partitions are not too small. By Theorem 1 ,the truncated reduced system is strictly diagonally dominant by rows with a degree no less than the original system. It consists of $p-1$ independent systems which are of dimension 2k. It follows that if Gaussian elimination runs to completion, then the computed solution $\hat{x_{tr}}$ of the computed truncated

reduced system $\hat{T}x_{tr} = \hat{g}_r$ satises

$$(\hat{T} + \triangle\hat{T})\hat{x_{tr}} = \hat{g}_r and |\triangle\hat{T}| \leq \gamma_{6k}|\hat{L}_t||\hat{U}_t| \tag{3.10}$$

where $\hat{L}_t\hat{U}_t$ is the computed LU factorization of the computed truncated reduced system matrix $\hat{T}$. It follows that,

$$\parallel \hat{x_{tr}} - x_{tr} \parallel_\infty \leq \frac{\beta}{1-\beta} \parallel x_{tr} \parallel_\infty, \parallel \hat{T}\hat{x_{tr}} - \hat{g}_r \parallel_\infty \leq \frac{\beta}{1-\beta} \parallel g_r \parallel_\infty \tag{3.11}$$

provided the unit round off error is small enough such that $\beta < 1$

4. Stage 4:

   Adjusting the original right-hand side, i.e., computing

   $$h_i = f_i - C_i x_{i-1}^b - B_i x_{i+1}^t \tag{3.12}$$

   introduces a small forward error. C_i affects only the top of f_i and B_i affects only the bottom f_i The component wise relative forward error is no more than,

   $$\hat{h_i} - h_i \leq \gamma_{k+1}(|f_i| - +C_i||x_{i-1}^b| + |B_i||x_{i+1}^t|) \tag{3.13}$$

   regardless of the order in which the scalar products are evaluated. Both the norm-wise relative residual as well as the norm-wise relative forward error are at most $\frac{\alpha}{1-\alpha}$

In a nutshell, if d is not too close to 1 and if the partitions are not too small, then the errors at every stage of the algorithm are small. The simplest way to evaluate the overall error was to calculate the residual and estimate,

$$\parallel x - y \parallel_\infty \leq \parallel A^{-1} \parallel_\infty \leq \parallel f - Ay \parallel_\infty \leq \frac{1}{1-d^{-1}} \parallel f - Ay \parallel_\infty \tag{3.14}$$

which turns out to be fairly effective as long as d is not too close to 1.

3.1.5 Theorems

Theorem 1 1. *Let A be strictly diagonally dominant by rows with degree* $d > 1$. *Then the matrices S, R, and T are strictly diagonally dominant by rows with degree no less than* d, *specifically*

$$d \leq d(S) \leq d(R) \leq d(T),$$

with equality possible. The condition numbers are upper bounded by $\frac{d+1}{d-1}$ with equality possible.

Theorem 2 1. *Let A be an* n by n *narrow banded matrix with upper and lower bandwidth* k, *and strictly diagonally dominant by rows with degree d. Then the truncation error satisfies*

$$\parallel R - T \parallel_{\infty} \leq max_{i=1,\ldots,p} d^{-q_i} \tag{3.15}$$

where $q_i = \lfloor \mu_i / k \rfloor$ and μ_i, *and is the size of the ith partition.*

3.2 SPIKE-TA OpenMP

As mentioned in the previous section, the theoretical speed-up of SPIKE-TA (or scaling factor) in comparison to LAPACK sequential is expected to be equal to the number of partitions p, given by $k/2 + 1$, where k represents the number of threads (k even number). It should be noted that in contrast to its counterpart MPI-implementation, the system matrix is not supposed to be distributed privately on the different threads by the user before entering the factorization stage. Since the entire system matrix is shared in memory, an additional copy of the matrix is also needed for $k > 2$ in order to perform both LU and UL factorizations independently on each thread. The partitioning stage is naturally hidden from the user in the SPIKE

OpenMP version, while on the other hand, it is unlikely that the unconventional partitioning of the TA-scheme can be handled with ease via MPI-programming from the user perspective. Although the extra copy which becomes a work array for the UL factorization, can be done in perfect parallel manner using all the available threads, one can expect a degradation of performance as compared to the ideal speed-up. It also makes SPIKE-OpenMP more expensive in memory as compared to LAPACK (we note, however, that since factorizations are performed without pivoting no extra-storage is necessary for the banded LAPACK format).

Both the routines 'XSPIKE_TRF' and 'XSPIKE_TRS' respectively for factorization and solve stages, have been created to mimic the functionalities of the LAPACK routines 'XGBTRF' and 'XGBTRS' (or the auxiliary routines 'XDBTRF' and 'XDBTRS' contained in the ScaLAPACK package [4] when applied to diagonally dominant systems). The SPIKE implementation makes uses of our in-house sequential (BLAS-3) banded primitives:

(i) 'XGBALU' and 'XGBAUL' respectively for approximate LU and UL banded factorizations (approximate stands for 'no pivoting and diagonal boosting if necessary') which, in turn, are exact for diagonally dominant systems;

(ii) 'XTBSM' for banded triangular solve which involves multiple right hand sides. The banded primitives library named 'libbprim' can currently be obtained from the public FEAST eigenvalue solver package [23].

3.3 Results and Discussion

The numerical experiments were carried out on an Intel Nehalem $X5550$ node featuring eight cores, running at $2.66Ghz$, with RAM: 48GB (12 x 4GB) DDR3-1333 Registered ECC (2 Modules/channel: 1066MHz max speed), 8MB Cache and, 6.4GT/s QPI.

The BLAS, BLAS-threaded and LAPACK were obtained from the Intel MKL version 10.1.0.015. The shell variable OMP_NUM_THREADS=k defines the number of threads for SPIKE OpenMP, while the shell variable MKL_NUM_THREADS=k controls the number of threads used by the MKL-LAPACK routines. The MKL-LAPACK results obtained with $k = 1$ are used as references for the sequential times, and the parallel runs are performed for the cases $k = 2$, $k = 4$ and $k = 8$. Denoting N the size of the system matrix (which is taken symmetric here), b its bandwidth, and $\#rhs$ the number of right hand sides, we propose to perform three sets of experiments reported in Table 3.3. We use *toeplitz* matrix (diagonal-constant matrix i.e. diagonals elements don't change), as our system matrix(A) in these experiments. For diagonally-dominant systems, $A = [0.1 \ldots 1.0 \ldots 6.0 \ldots 1.0]$ (6.0 on the diagonal, 1.0 on the two off-diagonals), and the constant vector 1.0 as the RHS.

At first, the scalability performances of MKL-LAPACK and SPIKE-OpenMP are compared for different size matrix N while a fixed bandwidth of $b = 200$ and only one right hand side are considered (Figure 3.5). MKL-LAPACK exhibits either some slight performance degradations mainly on two cores or very limited speed-up performances on four and eight cores (from $\sim 0.9\times$ to $\sim 1.3\times$). In contrast, a speed-up of $2\times$ is consistently obtained for all cases using SPIKE-OpenMP on two cores, while for large N/b ratio, the speed-up performances improves to $\sim 2.7\times$ on four cores and $\sim 3.9\times$ on eight cores. It should be noted that if the timing of the extra-copy of the matrix is not accounted for (i.e. if an additional copy is provided by the user for example), one obtains instead the (expected) speed-up of $3\times$ and $4.8\times$ respectively on four and eight cores. For smaller N/b ratio and in particular for the cases of $N = 20,000$, $N = 40,000$ and $N = 80,000$ a degradation of performance is observed while going from four to eight cores. The computational time becoming very small for four cores (example: $T = 0.1179/2.391 = 0.049s$ for $N = 20,000$), the cost of communications are expected to exceed the cost of computations and these performance degradations

	Time(s)	LAPACK speed-up			SPIKE-TA OMP speed-up		
# threads	1	2	4	8	2	4	8
b=200 #rhs=1							
N=20,000	0.1179	0.887	0.906	1.078	1.822	2.391	1.384
N=40,000	0.2051	0.987	1.297	1.229	1.853	2.540	1.769
N=80,000	0.3915	1.033	1.102	1.180	2.014	2.643	2.204
N=160,000	0.7537	1.019	1.088	1.173	2.137	2.697	3.011
N=320,000	1.4962	0.879	1.008	1.162	2.178	2.689	3.482
N=640,000	2.9736	1.127	1.137	1.332	2.192	2.730	3.765
N=1,280,000	5.9268	0.921	1.330	1.244	2.195	2.719	3.940
N=1,280,000 #rhs=1							
b=125	4.1719	1.005	1.007	1.008	2.161	2.763	4.033
b=250	8.7000	1.048	1.222	1.445	2.157	2.780	4.065
b=500	23.450	1.448	1.552	2.130	2.116	2.771	4.343
b=1000	77.223	1.369	2.085	2.948	2.079	2.799	4.177
N=1,280,000 b=1000							
#rhs=10	84.589	1.672	2.230	2.970	2.163	2.931	4.406
#rhs=100	139.95	1.838	2.727	4.178	2.773	3.793	5.834

Table 3.1. Three sets of experiments are used to illustrate the performances and scalability of MKL-LAPACK and SPIKE-OpenMP for solving real and double precision diagonally dominant systems. Both solvers are running on a Intel Nehalem node $X5550$ featuring eight cores running at $2.66Ghz$, with $48Gb$ total memory. The accuracy results on the residuals obtained by both solvers, not reported here, are comparable (accuracy machine).

	SPIKE-TA OMP speed-up			SPIKE-TA MPI speed-up		
# threads	2	4	8	2	4	8
			b=200 #rhs=1			
N=20,000	1.822	2.391	1.384	1.914	2.415	1.851
N=40,000	1.853	2.540	1.769	2.030	2.249	2.040
N=80,000	2.014	2.643	2.204	2.163	2.523	2.029
N=160,000	2.137	2.697	3.011	2.249	2.781	3.05
N=320,000	2.178	2.689	3.482	2.301	3.011	3.347
N=640,000	2.192	2.730	3.765	2.287	3.053	3.911
N=1,280,000	2.195	2.719	3.940	2.288	3.061	4.121
			N=1,280,000 #rhs=1			
b=125	2.161	2.763	4.033	2.351	3.153	4.490
b=250	2.157	2.780	4.065	2.365	3.222	4.571
b=500	2.116	2.771	4.343	2.357	3.115	4.579

Table 3.2. Two sets of experiments are used to compare the performances and scalability of SPIKE-OpenMP and SPIKE-MPI, truncated implementations for solving real and double precision diagonally dominant systems.

are likely to arise from the limitations of the architecture. As clear from Figure 3.5, for appropriate N/b ratio, however, SPIKE-OpenMP is expected to keep scaling on shared memory architectures as we move to more than eight cores.

In the second set of experiments, we propose to fix the system size at $N = 1,280,000$ and decrease the N/b ratio by increasing the bandwidth b. The MKL-LAPACK scaling performances become better with larger bandwidth since the speed-up of $\sim 1.4\times$, $\sim 2.0\times$ and $\sim 2.9\times$ are respectively obtained for the case $b = 1,000$ running on two, four and eight cores. Indeed, the underlying threaded BLAS level-3 is likely to become much more effective for larger dense bandwidth. The scalability results obtained by SPIKE are consistent with the ones observed in the first set of experiments with a slight improvement here on eight cores.

Finally, both the speed-up of the threaded MKL-LAPACK and SPIKE-OpenMP improved drastically while considering multiple right hand sides. Indeed, it is unlikely that the solve stage (i.e. forward and backward steps) in MKL-LAPACK sequential,

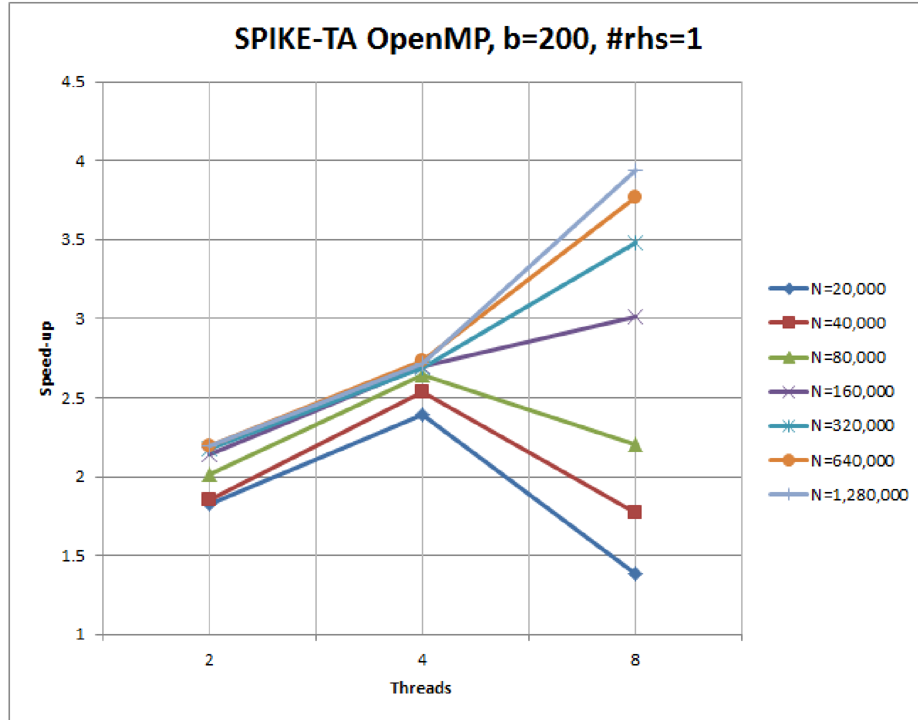

Figure 3.5. This figures shows the graphical representation of the speed-up of SPIKE-TA OpenMP, varying N with fixed b and #rhs

takes advantages of a BLAS-3 implementation since the underlying LAPACK basic algorithm does not (hence the need for developing the banded primitives library mentioned in the previous section). In this set experiments where the solve times becomes more important than the factorization times, the scalability results of MKL-LAPACK and SPIKE-OpenMP (i.e. going from two to four cores and then four to eight cores), perform well, but the SPIKE-OpenMP timing performances are still much faster.

The table 3.3 illustrates the results we obtain after performing two new sets of experiments on the same Intel Nehalem machine to gauge how SPIKE-TA OpenMP performs in comparison to its distributed memory MPI version: varying the size of the system, N from $20,000$ to $1,280,000$, with constant bandwidth, b during the first experiment (Figure3.6), and varying the b with constant N in the second experiment. As clear from the table, OpenMP matches up the MPI the speed-up for the runs

on two processors/threads irrespective of the size of the system or the bandwidth. For the runs on the 4 processors/threads, OpenMP evenly matches up MPI for small systems, but falls marginally short on speed-up as the size of the system or the bandwidth become large. For the 8 processor runs, OpenMP consistently gets a speed up of 4×, while MPI marginally strides ahead with a speed 4.2× for large N and/or b. However, our OpenMP implementation takes into account the times incurred for an additional copy of the system matrix, but if we would just account for the time taken for factorize and solve, the minor slump in speed-up would be negated, and exactly match up with that of MPI.

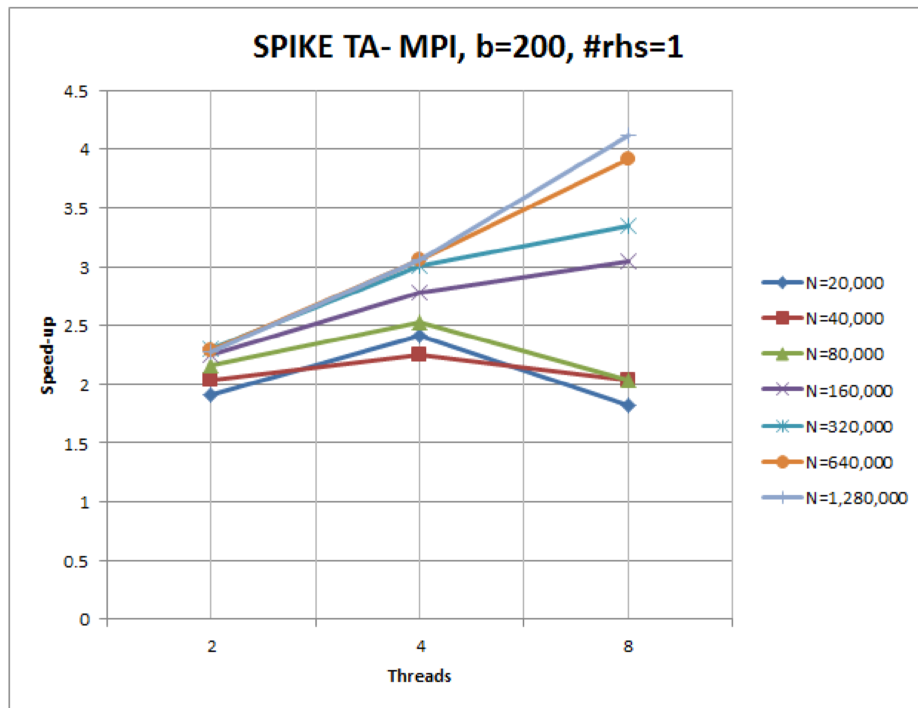

Figure 3.6. This figures shows the graphical representation of the speed-up of SPIKE-TA MPI, varying N with fixed b and #rhs, qualitatively similar to OpenMP

CHAPTER 4

SPIKE OPENMP FOR NON-DIAGONALLY DOMINANT SYSTEMS

4.1 Schemes

We have implemented the following two schemes in particular to tackle the systems that are non-diagonally dominant :-

1. SPIKE-RL: We have implemented the SPIKE-RL (recursive) version in OpenMP, as numerical experiments show that good-scaling in observed in comparison to LAPACK for solving non-diagonally dominant systems. A description of the RL-scheme is summarized in the next section.

2. Spike-calling-Spike (SCS): Based on good scalability results for the timings we obtained on 2-threads (processors), we implemented a 2×2 MPI kernel in addition to the 2×2 OpenMP factorize and solve routines, and from within the MPI kernel call these routines for solving non-diagonally dominant systems. The timings were expected to be similar in comparison to the SPIKE-RL for the 2 and 4 threads case. We hence implemented a SCS solver, specific for the 4-processor case, so we could compare the timing results with our OpenMP implementation of SPIKE-RL scheme.

   The 2×2 MPI kernel has been implemented using the modified *Banded Primitive* [23] routines, and the 2×2 OpenMP factorize and solve routines were extracted from our OpenMP implementation of the SPIKE-TA algorithm. The following table shows preliminary speed-up results we obtained for the 4-processor case:-

Time(s)	SCS MPI-OpenMP speed-up 4-threads
b=200	**#rhs=1**
N=20,000	1.379
N=40,000	1.405
N=80,000	1.466
N=160,000	1.507
N=320,000	1.580
N=640,000	1.590
N=1,280,000	1.592

Table 4.1. SCS speed-up on 4-processors

N denotes the size of the system matrix, b denotes the bandwidth and # *rhs*, the number of right-hand sides. The speed-up results for different system sizes Table(4.1), suggest that the speed-up on 4-threads is less in comparison to SPIKE-RL scheme for same system parameters (*N, b, # rhs*). We hence conclude that 2×2 kernel, which showed excellent scalability on 2-processors, isn't effective when cast for 4-processors using nested 2×2 kernel calls, because recursive implementation on large vectors, i.e. spikes V and W, is less effective in comparison to a recursive implementation on smaller vectors like reduced system, as is done in the SPIKE-RL scheme. Consequentially, we instead pursued with implementing the SPIKE-RL (recursive) algorithm in OpenMP.

4.2 Recursive SPIKE

4.2.1 Two partitions case

The recursive version of the SPIKE algorithm is general enough for handling non-diagonally dominant systems. We need just the tips of the spikes in the RL scheme for forming the reduced system and then solving it. The RL-scheme saves memory as we are not required to store the entire SPIKE matrix. The RL scheme uses recursion on reduced system which is much smaller in comparison to doing recursion on the

entire partitions of the system(which is the case in Spike-calling-Spike version). Here, since we are dealing with non-diagonally dominant systems, we need both the top and bottom of both the spikes, unlike the Truncated scheme for the diagonally dominant case, where we could neglect the top of V (super-diagonal spike) and bottom of W (sub-diagonal spike). Another point worth mentioning for the RL-scheme is that the # partitions = # processors, in power of 2. The following steps demonstrate the working of the RL scheme:

1. Preprocessing Stage:

   The first step of the preprocessing stage is to factorize the diagonal blocks A_j. For numerical stability, LAPACK's XGBTRF routines can be used to LU factorize them with partial pivoting. Alternatively, they can be also factorized without partial pivoting but with a "diagonal boosting" strategy. The latter method tackles the issue of singular diagonal blocks. In concrete terms, the diagonal boosting strategy is as follows. Let ϵ denote a configurable machine zero and $\| \bullet \|$ is the 1-norm. In each step of LU factorization, the pivot should satisfy the condition: $|pivot| > \epsilon \parallel A \parallel_1$

   If the pivot does not satisfy the condition, it is then boosted by:

   $pivot = pivot + \epsilon' \parallel A_j \parallel_1 \qquad if \quad pivot \geq 0$

   $pivot = pivot - \epsilon' \parallel A_j \parallel_1 \qquad if \quad pivot \leq 0$

2. Post-processing Stage:

   In the two-partition case, i.e., when $p = 2$, the reduced system $\hat{S}\hat{X} = \hat{G}$ has the form

$$\begin{pmatrix} I_m & 0 & V_1^{(t)} & 0 \\ 0 & I_m & V_1^{(b)} & 0 \\ 0 & W_2^{(t)} & I_m & 0 \\ 0 & W_2^{(t)} & I_m & 0 \end{pmatrix} \begin{pmatrix} X_1^{(t)} \\ X_1^{(b)} \\ X_2^{(t)} \\ X_2^{(b)} \end{pmatrix} = \begin{pmatrix} G_1^{(t)} \\ G_1^{(b)} \\ G_2^{(t)} \\ G_2^{(b)} \end{pmatrix} \tag{4.1}$$

An even smaller system can be extracted from the center:

$$\begin{pmatrix} I_m & V_1^{(b)} \\ W_2^{(t)} & I_m \end{pmatrix} \begin{pmatrix} X_1^{(b)} \\ X_2^{(t)} \end{pmatrix} = \begin{pmatrix} G_1^{(b)} \\ G_2^{(t)} \end{pmatrix} \tag{4.2}$$

that can be solved using the blocked LU factorization.

$$\begin{pmatrix} I_m & V_1^{(b)} \\ W_2^{(t)} & I_m \end{pmatrix} = \begin{pmatrix} I_m & 0 \\ W_2^{(t)} & I_m \end{pmatrix} \begin{pmatrix} I_m & V_1^{(b)} \\ 0 & I_m - W_2^{(t)} V_1^{(b)} \end{pmatrix} \tag{4.3}$$

Once $X_1^{(b)}$ and $X_2^{(t)}$ are computed, $X_1^{(b)}$ and $X_2^{(t)}$ can be retrieved using the following:

$$X_1^{(t)} = G_1(t) - V_1^{(t)} X_2^{(t)} \tag{4.4}$$

$$X_2^{(b)} = G_2(t) - W_2^{(b)} X_1^{(b)} \tag{4.5}$$

This can be generalized to cases where p or the number of partitions is a power of two, i.e., $p = 2^d$.

4.2.2 Multiple partitions case

It is assumed that the number of partitions is given by $p = 2^d$ $(d > 1)$. After forming the spike matrix S, the number of partitions of the new linear system $SX = G$ can be divided by two, and another level of the SPIKE algorithm may be applied. This process is repeated recursively until only two partitions for the newest matrix S are obtained. The resulting reduced system has then the form (4.2).

Recursive scheme is not concerned with the overall matrix S but rather with the matrix $\hat{S}$ of the reduced system itself. This allows simplified implementation, and reduces the memory requirements while saving all the different levels of the new spikes. In the reduced system, the matrix $\hat{S}$ is block tridiagonal.

Observing that we can also extract an independent reduced system of an order $2mp$, rather than order $2m(p-1)$, if we include in addition the top m rows and the bottom m rows of the first and last partition, respectively. The structure of the reduced system remains block tridiagonal but in this case, each diagonal block is an identity matrix of order *2m*, and the off-diagonal blocks associated with the kth diagonal block are given by

$$\begin{pmatrix} 0 & W_k^{(t)} \\ 0 & W_k^{(b)} \end{pmatrix} \quad for \;\; k = 2, ..., p \;\; and, \begin{pmatrix} V_k^{(t)} & 0 \\ V_k^{(b)} & 0 \end{pmatrix} \quad for \;\; k = 1, ..., p-1 \tag{4.6}$$

Denoting the spikes of the new reduced system at level 1 of the recursion by $v_k^{[1]}$ and $w_k^{[1]}$, where

$$v_k^{[1]} = \begin{pmatrix} V_k^{(t)} \\ V_k^{(b)} \end{pmatrix}, \;\; and \;\; w_k^{[1]} = \begin{pmatrix} W_k^{(t)} \\ W_k^{(b)} \end{pmatrix} \tag{4.7}$$

the matrix $\tilde{S}_1$ of the new reduced system, for $p = 4$ takes the form

$\tilde{\mathbf{S}}_1 =$ [block matrix with diagonal blocks I]

right spikes: $v_k^{[1]}$; $k = 1, 2, 3$

left spikes: $w_k^{[1]}$; $k = 2, 3, 4$.

Figure 4.1. The spike matrix of the new reduced system for p=4

In preparation for level 2 of the recursion of the SPIKE algorithm, we choose now to partition the matrix $\tilde{S}_1$ using *p/2* partitions each of size *4m*. The matrix can then be factored as,

$$\tilde{S}_1 = D_1 \tilde{S}_2 \tag{4.8}$$

where D_1 is formed by the *p/2* diagonal block of $\tilde{S}_1$ each of size *4m*, thus $\tilde{S}_2$ represents the new Spike matrix at level 2 composed of the spikes $v_k^{[2]}$ and $w_k^{[2]}$. For four partitions, $p = 4$, these matrices are of the form,

$\mathbf{D_1} =$ [block diagonal matrix with blocks I, 0, 0, I and spikes]

right spikes: $v_k^{[1]}$; $k = 1, 3$

left spikes: $w_k^{[1]}$; $k = 2, 4$

and

$\tilde{\mathbf{S}}_\mathbf{2} =$ [matrix with diagonal blocks I, I and spikes]

right spikes: $v_k^{[2]}$; $k = 1$

left spikes: $w_k^{[2]}$; $k = 2$.

In general, at level i of the recursion, the spikes $v_k^{[i]}$, and $w_k^{[i]}$, with k ranging from 1 to $p/(2^i)$, are of order $2^i m * m$. Thus, if the number of the original partitions p is equal to 2^d, the total number of recursion levels is $d - 1$ and the matrix $\tilde{S}_1$ can be expressed in the factored form,

$$\tilde{S}_1 = D_1 D_2 .... D_{d-1} \tilde{S}_d \tag{4.9}$$

where the matrix $\tilde{S}_d$ has only two spikes $v_1^{[d]}$ and $w_2^{[d]}$. The linear reduced system can then be written as

$$\tilde{S}_d \tilde{X} = B \tag{4.10}$$

where B is the modified right-hand-side given by

$$B = D_{d-1}^{-1} \dots D_2^{-1} D_1^{-1} \tilde{G} \tag{4.11}$$

assume that the spikes $v_1^{[i]}$ and $w_2^{[i]}$ of the matrix $\tilde{S}_i$ are known at a given level i, then we can compute the spikes $v_k^{[i+1]}$ and $w_k^{[i+1]}$ at level *i+1* as follows:

Step 1: Denoting the bottom and top blocks of the spikes at the level i by

$$v_k^{[i](b)} = \begin{pmatrix} 0 & I_m \end{pmatrix} v_k^i; \;\; w_k^{[i](t)} = \begin{pmatrix} I_m & 0 \end{pmatrix} w_k^i \tag{4.12}$$

and the middle block of *2m* rows of the spikes at the level *i+1* by

$$\begin{pmatrix} \dot{v}_k^{[i+1]} \\ \ddot{v}_k^{[i+1]} \end{pmatrix} = \begin{pmatrix} 0 & I_{2m} & 0 \end{pmatrix} v_k^{[i+1]}; \;\; \begin{pmatrix} \dot{w}_k^{[i+1]} \\ \ddot{w}_k^{[i+1]} \end{pmatrix} = \begin{pmatrix} 0 & I_{2m} & 0 \end{pmatrix} w_k^{[i+1]} \tag{4.13}$$

and the reduced systems can be formed as following:

$$\begin{pmatrix} I_m & v_{2k-1}^{[i](b)} \\ w_{2k}^{[i](t)} & I_m \end{pmatrix} \begin{pmatrix} \dot{v}_k^{[i+1]} \\ \ddot{v}_k^{[i+1]} \end{pmatrix} = \begin{pmatrix} 0 \\ v_{2k}^{[i](t)} \end{pmatrix}, \;\; k = 1, 2, \dots, \frac{p}{2^{i-1}} - 1 \tag{4.14}$$

and

$$\begin{pmatrix} I_m & v_{2k-1}^{[i](b)} \\ w_{2k}^{[i](t)} & I_m \end{pmatrix} \begin{pmatrix} \dot{w}_k^{[i+1]} \\ \ddot{w}_k^{[i+1]} \end{pmatrix} = \begin{pmatrix} w_{2k-1}^{[i](b)} \\ 0 \end{pmatrix}, \;\; k = 2, 3, \dots, \frac{p}{2^{i-1}} \tag{4.15}$$

These reduced systems are solved similarly to (4.2) to obtain the solutions of the center parts of the spikes at the level i + *1*.

Step 2: The entire solution of the spikes at the level *i + 1* is retrieved as follows:

$$\begin{pmatrix} I_{2^i m} & 0 \end{pmatrix} v_k^{[i+1]} = -v_{2k-1}^{[i]} \ddot{v}_k^{[i+1]}, \quad \begin{pmatrix} 0 & I_{2^i m} \end{pmatrix} v_k^{[i+1]} = v_{2k}^{[i]} - w_{2k}^{[i]} \dot{v}_k^{[i+1]} \tag{4.16}$$

and

$$\begin{pmatrix} I_{2^i m} & 0 \end{pmatrix} w_k^{[i+1]} = w_{2k-1}^{[i]} - v_{2k-1}^{[i]} \ddot{w}_k^{[i+1]}, \quad \begin{pmatrix} 0 & I_{2^i m} \end{pmatrix} v_k^{[i+1]} = -w_{2k}^{[i]} \dot{w}_k^{[i+1]} \tag{4.17}$$

In order to compute one step of the modified RHS as $\tilde{G}_i = D_i^{-1}\tilde{G}_{i-1}$ (with the first being $\tilde{G}_1 = D_1^{-1}\tilde{G}$) one has to solve the linear system $D_i\tilde{G}_i = \tilde{G}_{i-1}$, which is a block diagonal. For each diagonal block k, the reduced systems are similar to those in (4.14) and (4.15), but the RHS is now defined as a function of $\tilde{G}_{i-1}$. Once we get the partial solution at the center part of each $\tilde{G}_i$, associated with each block k in D_i, the entire solution is retrieved as in (4.16) and (4.17). In the same way, the linear system in 4.10 involves only one reduced system to solve and only one retrieval stage to get the solution $\tilde{X}$. Finally, the overall solution X is obtained using the following equations:

$$X'_1 = G'_1 - V'_1 X_2^{(t)} \tag{4.18}$$

$$X'_j = G'_j - V'_j X_{j+1}^{(t)} - W'_j X_{j-1}^{(b)}, \quad j = 2, \ldots, p-1. \tag{4.19}$$

$$X'_p = G'_p - W'_j X_{p-1}^{(b)} \tag{4.20}$$

4.3 Implementation

As mentioned previously, for the recursive spike algorithm, we have number of partitions same as the number of processors. The recursive scheme unlike the truncated scheme requires generation of entire spike, as the tips of the spikes, V_i^b, $V_i^t (i =$

$1, \ldots, p-1$) and $W_j^b, W_j^t (j = 2, \ldots, p)$, are required to computed for forming reduced systems at different levels of recursion.

However, since preprocessing factorization stage which involves computing spikes is predominantly the most time consuming part of the entire solver, we use a trick to speed up this stage. We use LU/UL strategy that allows us to save two sweeps in total on the first and the last partition combined. Furthermore, as an enhancement, for the first and the last partition, we use only a small solve sweep (DTBSM) for L and U, and complete partition sweeps on the middle partitions, totalling three sweeps. Therefore, we save four sweeps in total, two on both the first and the last partition. We can do this V_1^t and W_n^b, n : *last partition index* are not used anywhere during the final retrieval of the solution, and doesnt corrupt the final residual.

We cast two subroutines, the first one, *dspikerl_trf* which assimilates the spikes matrices and reduced system at different recursion levels, and the second one *dspikerl_trs*, that obtains the modified rhs, solves the reduced system and performs the retrieval of the overall solution.

Figure 4.2. Figure showing the factorization step, and forming reduced systems at different levels of recursion for SPIKE-RL

The above figure (4.2) illustrates the factorization stage of system matrix, i.e. calculating spikes V,W and forming the reduced systems at different levels of recursion. We associate to each CPU a particular reduced system (which is required for the

recursion). For example, as above, when the number of partitions are eight (run on 8-threads or processors equivalently), the following are the reduced system-processor associations:

1. for level 1 of recursion we get the reduced systems associated with CPU= 1,3,5,7
2. for level 2 we get the reduced systems associated with CPU= 2,6
3. for level 3 we get the reduced systems associated with CPU= 4

4.3.1 Load-balancing

For the recursive scheme, we use a non-uniform partitioning strategy for execution on more than 2 two threads (or processors). Since the first and the last matrix partitions do less computational sweeps in comparison to middle partitions, we hence make their size larger than the rest of the partitions. This allows us to leverage the timing benefit, by doing full-computation on smaller matrix partitions. An input parameter *ratio*, which is the ratio between the size of first (last) partition and middle partitions, governs how large the first and last partitions would be relative to the middle ones. We formulate an expression based on the size of a partition, computational cost associated with that partition, and assuming the cost of LU is greater than one sweep but less than two sweeps ($1 - sweep < LU < 2 - sweeps$). Using this relation, we determine an optimum value of this ratio, and confirm it using numerical evidence (provided in next section).

Let *nbpart* denote the number of partitions. It may be noted that the size of,

$$partition(1) = partition(nbpart) \tag{4.21}$$

$$partition(2) = partition(i) \quad i = 3, \ldots, (nbpart - 1) \tag{4.22}$$

The optimal load balancing ratio is obtained by solving the following set (4.3.1), (4.3.1) of expressions:

$$2 * x + (nbpart - 2) * y = 1, \tag{4.23}$$

where, **x**: first partition fraction of total, **y**: second partition fraction of total

$$(2^i + 2) * y = \frac{1}{2} \tag{4.24}$$

where, **i** is the total number of recursion levels.

We hence coin a new term, the optimal ratio, R_t , given by,

$$R_t = \frac{x}{y} \tag{4.25}$$

Using the above relations, we get $R_t = 3$.

4.3.2 Retrieval

Since in many applications, often many linear systems with the same coefficient matrix A but with different RHSs have to be solved, optimization of the solve stage is also necessary. We aim to save time during the retrieval by doing a small solve sweep (DTBSM) for U for the first partition and a similar small sweep for L for the last partition, while solving for the modified rhs i.e. $A_j g_j = f_j$. After obtaining the reduced solution, we finish U on the first partition, and L on the last partition, prior to performing vector addition to update the final solution. The retrieval in essence is similar to the one performed in the truncated scheme.

4.4 Results and Discussion

We use the same shell variables as described in the previous chapter for the truncated spike algorithm. The MKL-LAPACK results obtained with $k = 1$ (k defines the

number of threads used by OpenMP and MKL-LAPACK) are used as references for the sequential times, and the parallel runs are performed for the cases $k = 2, k = 4$ and $k = 8$. Denoting N the size of the system matrix (which is taken symmetric here), b its bandwidth, and #rhs the number of right hand sides, we perform three sets of experiments reported in Table(4.4). For non-diagonally dominant systems, $A = [0.1 \ldots -4.0 \ldots 3.0 \ldots -4.0d]$ and the constant vector 1.0 as the RHS, for all the experiments.

	Time(s)	LAPACK speed-up			SPIKE-RL OMP speed-up		
# threads	1	2	4	8	2	4	8
b=200 #rhs=1							
N=20,000	0.112	1.047	1.190	1.235	1.821	1.864	1.136
N=40,000	0.209	1.050	1.135	1.304	1.837	2.245	1.935
N=80,000	0.414	1.015	1.180	1.222	1.964	2.565	2.192
N=160,000	0.822	1.026	1.117	1.196	2.081	2.740	3.102
N=320,000	1.626	1.007	1.053	1.221	2.077	2.756	3.380
N=640,000	3.230	1.009	1.042	1.149	2.083	2.692	3.557
N=1,280,000	6.425	1.004	1.193	1.262	2.059	2.700	3.564
N=1,280,000 #rhs=1							
b=125	4.051	1.0080	1.0050	1.0374	1.897	2.544	3.328
b=250	9.070	1.0549	1.3437	1.4239	2.057	2.751	3.476
b=500	27.78	1.2314	1.7717	2.1857	2.283	2.806	3.566
b=1000	99.09	1.4022	2.6090	3.3727	2.522	3.161	3.788
N=1,280,000 b=1000							
#rhs=10	104.9	1.3863	2.5812	3.3915	2.576	3.178	3.957
#rhs=100	158.5	1.5680	2.7081	4.0649	3.040	3.738	4.731

Table 4.2. Scalability comparison, LAPACK vs SPIKE-RL OpenMP: Three sets of experiments are used to illustrate the performances and scalability of MKL-LAPACK and SPIKE-RL OpenMP for solving real and double precision non-diagonally dominant systems. Both solvers are running on a Intel Nehalem node X5550 featuring eight cores running at 2.66Ghz, with 48Gb total memory. The accuracy results on the residuals obtained by both solvers are comparable of the order of $\sim 10^{-10}$, and no diagonal-boosting is observed.

As clear from the table(4.4) above, we see that LAPACK doesn't scale at all for small N and small bandwidths. Scaling is only observed for extremely large N

($\sim 1,280,000$) with large bandwidths (~ 1000) and multiple right hand sides. As such for bigger matrices, use of BLAS-threaded in their implementation boosts the speed-up on four and eight processors.

In comparison to our in-house banded primitives Table(4.4) ([23]) , consistent scaling, 1.8× on two processors, 2.5× on four processors and 2.9× on eight processors, is observed by by our SPIKE- Recursive OpenMP implementation. It may be noted that our implementation does not do pivoting, as is the case with banded primitive.

Like LAPACK, the banded primitive routines, only show scaling for large system parameters. Moreover, SPIKE-RL is expected to scale beyond 8-processors, while banded-primitive speed-ups will level-off or saturate after a point as we go beyond eight processors. The banded primitives for smaller systems, don't show any significant speed-up(scaling). On 2-processors in fact, it is slower ($\sim 0.95\times$) in comparison to sequential run on one processor($k = 1$). In our SPIKE-RL implementation, as a time-saving artifice, the generation of the spikes at the various levels is included in the factorization step. In this way, the solver makes use of the spikes stored in the memory thus allowing solving the reduced system quickly and efficiently. We also optimize and speed-up the retrieval, which is a part of the solve stage, by doing it similar to way we did for the SPIKE-truncated case. The residual using all the schemes, namely, LAPACK, Banded Primitive and SPIKE-RL, are comparable and of the order of $\sim 10^{-10}$, and is even better for smaller systems.

Determining R_t experimentally

After conducting numerical experiments for different ratio values (in steps of 0.2) as shown in Table(4.4), Figure(4.3) and Figure(4.4), we find the run time is minimum when the ratio of the sizes is 3.2, which is in very good agreement to our mathematical estimate of $R_t = 3$. This enables us good time savings and helps us to beat LAPACK.

Furthermore, we gauge impact of R_t on time taken for the solve stage (solve time), as shown in the Figure(4.5) for $N = 640,000, b = 200, \#rhs = 1$.

	Time(s)	Banded Primitive speed-up			SPIKE-RL OMP speed-up		
# threads	1	2	4	8	2	4	8
b=200 #rhs=1							
N=20,000	0.0993	0.984	1.160	0.982	1.615	1.502	1.007
N=40,000	0.1863	0.999	1.210	1.150	1.637	2.001	1.725
N=80,000	0.3505	0.969	1.247	1.105	1.661	2.170	1.854
N=160,000	0.6895	0.974	1.245	1.313	1.746	2.298	2.602
N=320,000	1.3603	0.955	1.112	1.228	1.737	2.306	2.828
N=640,000	2.702	0.953	1.185	1.206	1.742	2.252	2.976
N=1,280,000	5.387	0.960	1.113	1.187	1.726	2.263	2.988
N=1,280,000 #rhs=1							
b=125	3.3790	0.987	0.985	0.972	1.583	2.122	2.776
b=250	7.7801	1.009	1.228	1.439	1.765	2.306	2.982
b=500	22.0112	1.243	1.830	2.223	1.810	2.223	2.825
b=1000	73.9901	1.554	2.599	3.904	1.883	2.360	2.828
N=1,280,000 b=1000							
#rhs=10	77.07	1.5716	2.5385	3.9082	1.893	2.335	2.907
#rhs=100	99.09	1.6287	2.6616	4.0896	1.901	2.338	2.959

Table 4.3. Scalability comparison, Banded Primitive vs SPIKE-RL OpenMP: The same three sets of experiments, on a non-diagonally dominant real double-precision system matrix, carried out on the same machine, are used to illustrate their performance and scalability

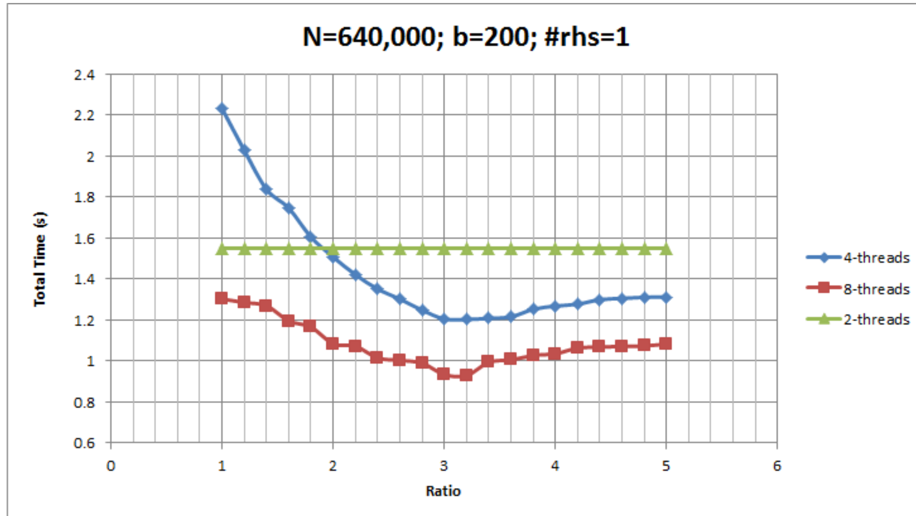

Figure 4.3. Total time in seconds for, $N = 640,000, b = 200$, and $\#rhs = 1$, on two, four and eight-threads. Total Time for two threads = 1.551 sec. Fastest run (smallest time) on both 4 and 8-threads is at value(ratio) 3.2

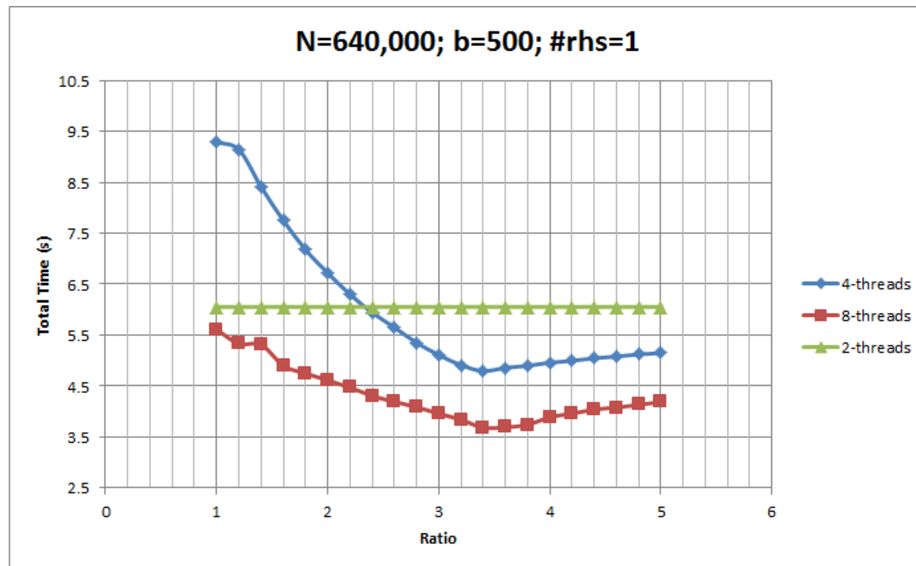

Figure 4.4. Total time in seconds for, $N = 640,000, b = 500$, and $\#rhs = 1$, on two, four and eight-threads. Total Time for two threads = 6.054 sec. Fastest run (smallest time) on both 4 and 8-threads is at value(ratio) 3.4

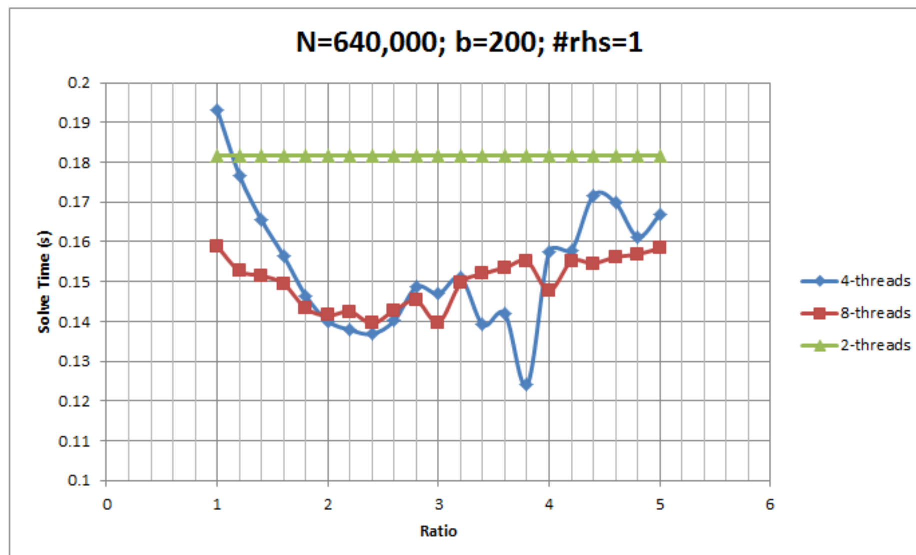

Figure 4.5. Solve time in seconds for, $N = 640,000, b = 200$, and $\#rhs = 1$, on two, four and eight-threads. Solve Time for two threads = 0.181 sec.

Ratio	**N=640,000, b=200, # rhs=1**		**N=640,000, b=500, # rhs=1**	
# threads	4	8	4	8
1	2.230	1.302	9.290	5.610
1.2	2.027	1.285	9.140	5.331
1.4	1.837	1.269	8.410	5.314
1.6	1.745	1.195	7.751	4.900
1.8	1.604	1.165	7.177	4.748
2	1.507	1.084	6.720	4.615
2.2	1.421	1.072	6.302	4.472
2.4	1.351	1.015	5.942	4.308
2.6	1.302	1.003	5.652	4.193
2.8	1.248	0.989	5.350	4.087
3	1.205	0.934	5.110	3.960
3.2	1.202	0.928	4.911	3.839
3.4	1.208	0.993	4.796	3.686
3.6	1.215	1.005	4.860	3.890
3.8	1.252	1.027	4.899	3.741
4	1.268	1.035	4.961	3.891
4.2	1.277	1.062	4.997	3.960
4.4	1.299	1.068	5.048	4.042
4.6	1.305	1.070	5.081	4.079
4.8	1.311	1.073	5.123	4.134
5	1.315	1.083	5.15	4.188

Table 4.4. Total(Fact + Solve) time in seconds for different partition-size ratios for N=640,000; B=200 and B=500; # rhs=1, for two, four and eight threads.

We can conclude that the solve time, which is faster in comparison to factorize time, is the fastest at an average ratio of about 3.2 with regard to four and eight-threads. The solve time vs. R_t curve for four and eight threads seems to be noisy, as the solve time being only a fraction of the total time, exhibits more volatility arising on account of cache access misses.

Hence, with experimental timing evidence backing our claim, we set the value of the partition size ratio(R_t) to 3.2 while carrying out the three sets of timing experiments for SPIKE-RL.

CHAPTER 5

CONCLUSION

A shared-memory version of the SPIKE-TA algorithm has been implemented using OpenMP directives for solving diagonally dominant banded linear systems. To extend the solver capability, another version, the SPIKE-RL algorithm, has also been implemented for solving non-diagonally dominant systems. system has also been implemented for shared-memory machines The resulting SPIKE-OpenMP solver aims at providing a scalable shared-memory parallel version of the sequential LAPACK banded routines, as well as competing with LAPACK performances obtained using a BLAS-threaded library. Significant performance and scalability results have been obtained. In general, such results have to be expected from a higher level parallelism paradigm (i.e. SPIKE-TA acts at a coarser parallelism level) rather than exclusively using low-level parallel optimizations. As compared to the LAPACK sequential LU algorithm, however, the SPIKE-TA scheme inherits a more memory expensive preprocessing stage if the number of partitions is greater than two. The SPIKE-Recursive scheme for non-diagonally dominant systems on the other hand is less expensive in memory, requiring only the tips of the spikes to be saved for computation during the factorization and solve stages. It may be noted that while the SPIKE-RL scheme does not use pivoting, LAPACK on the contrary does use pivoting for non-diagonally dominant systems, making it more expensive in time.

Finally, we note that the TA-scheme using two partitions can also be cast as a direct system solver when applied to non-diagonally dominant systems. No spike truncation is necessary, and although the factorizations are performed without pivoting,

the overall system would only be slightly modified if a diagonal boosting takes place, leading to very accurate solutions after very few iterative refinements. In Table(3.3), one can observe that the speed-up $\sim 2\times$ obtained using SPIKE-TA OpenMP on two cores is, for most cases, superior to the speed-up obtained by MKL-LAPACK on eight cores for diagonally dominant systems. As evident from Table(4.4), the SPIKE-RL shows $\sim 2\times$ on two cores, $\sim 2.7\times$ on four and $\sim 3.5\times$ on eight. We expect it to continue scaling beyond eight cores as well. Thus, the scalability results hold for non-diagonally systems as well. Although many other SPIKE-schemes have been proposed and implemented using MPI-directives to deal with non-diagonally systems [16, 17], they may suffer from expensive memory management on shared memory architectures using more than two cores.

The goal would now be to develop a banded package for shared-memory machines as an alternative(replacement) to(for) LAPACK-BLAS threaded solver, 'X'GBTRF (factorize), 'X'GBTRS (solve) routines with the SPIKE counterparts, developed during this research for both diagonally-dominant and non-diagonally dominant systems. We need to match-up with the current LAPACK libraries only in terms of functionality, as we are more competitive in scalability. Adding an option for the transpose('T') of the matrix, along with supporting single-precision and complex arithmetic, and enhancing the capability to run on odd number of processors, would round-up the SPIKE-OpenMP package and make it equally competent in terms of functionality as well.

BIBLIOGRAPHY

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, et al. *LAPACK users guide.* 3rd ed. Philadelphia, PA: Society for Industrial and Appl. Math.; (1999).

[2] A. Cleary and J. Dongarra.*Implementation in ScaLAPACK of divide and conquer algorithms for banded and tridiagonal linear systems.* University of Tennessee Computer Science Technical Report, UT-CS-97-358; (1997).

[3] S.C. Chen, D.J. Kuck, and A. Sameh, *Practical parallel band triangular system solvers*, ACM Trans. Math. Software, 4, pp. 270277(1978).

[4] L.S. Blackford, J. Choi, A. Cleary, E. DAzevedo, J. Demmel, I. Dhillon, et al. *ScaLAPACK users guide.* Philadelphia, PA: Society for Industrial and Appl. Math.; (1997).

[5] P. Arbenz, A. Cleary, J. Dongarra, and M. Hegland, *A comparison of parallel solvers for diagonally dominant and general narrow banded linear systems II*, EuroPar Parallel Processing, p. 10781087;(1999)

[6] A. Sameh. *Numerical parallel algorithms: a survey.* In: Kuck D, Lawrie D, Sameh A, editors. High speed computer and algorithm organization. New York: Academic Press; p. 207-228 (1977).

[7] A. Sameh and D. Kuck. *On stable parallel linear system solvers.* J. ACM, 25, p. 81-91; (1978).

[8] A. Sameh. *On two numerical algorithms for multiprocessors.* Proc of NATO adv res workshop on high-speed comp. Series F: computer and systems sciences, vol. 7. Berlin: Springer; p. 31128, (1983).

[9] D. Lawrie and A. Sameh.*The computation and communication complexity of a parallel banded system solver.* ACM Trans Math Software, 10(2); p. 185-195; (1984).

[10] J. Dongarra and A. Sameh.*On some parallel banded system solvers.* Parallel Computing, 1, p. 223-235; (1984).

[11] J.-L. Larriba-Pey, A. Jorba, and J.J. Navarro, *Spike algorithm with savings for strictly diagonal dominant tridiagonal systems*, Microprocessing and Microprogramming, 39, pp. 125128 (1993).

[12] D.H. Lawrie and A. Sameh, The computation and communication complexity of a parallel banded system solver, ACM Trans. Math. Software, 10, pp. 185195 (1984).

[13] M. Berry and A. Sameh. em Multiprocessor schemes for solving block tridiagonal linear systems. Int J. Supercomput. Appl., 2(3): p. 37-57; (1988).

[14] N.J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, Philadelphia, PA, (2002).

[15] A. Sameh, and V. Sarin. *Hybrid Parallel Linear Solvers.* International Journal of Computational Fluid Dynamics, Vol 12, p. 213-223, (1999).

[16] E. Polizzi, A. Sameh. *A Parallel Hybrid Banded System Solver: The SPIKE Algorithm*, Parallel Computing, V. 32, 2, p. 177-194 (2006).

[17] E. Polizzi, A. Sameh. *SPIKE: A parallel environment for solving banded linear systems*, Computers & Fluids, 36 p. 113-120 (2007).

[18] A distributed memory version of the SPIKE package can be obtained from: *http://software.intel.com/en-us/articles/intel-adaptive-spike-based-solver/*

[19] E. Polizzi. *The SPIKE software.* Book Chapter in Springer Encyclopedia of Parallel Computing, D. Padua (Ed.), Springer, to appear. (2010).

[20] O. Schenk and K. Gartner. *Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO.* Journal of Future Generation Computer Systems, 20(3), p. 475-487; (2004).

[21] S. Demko, W.F. Moss and P.W. Smith.*Decay rates for inverses of band matrices.* Math Comput., 43(168), p. 491-499; (1984).

[22] C.C.K. Mikkelsen and M. Manguoglu. *Analysis of the Truncated Spike Algorithm.* SIAM Journal on Matrix Analysis and Applications, 30(4), p.1500-1519; (2008).

[23] *http://www.ecs.umass.edu/∼polizzi/feast*; (2009).

[24] X.-H. Sun, H. Zhang, and L.M. Ni, *Efficient tridiagonal solvers on multicomputers*, IEEE Trans. Comput., 41, p. 286296 (1992).

[25] X.-H. Sun, *Application and accuracy of the parallel diagonal dominant algorithm*, Parallel Comput., 21, p. 12411267 (1995).