Journal of King Saud University – Computer and Information Sciences (2014) **26**, 41–54

King Saud University

Journal of King Saud University – Computer and Information Sciences

www.ksu.edu.sa
www.sciencedirect.com

ORIGINAL ARTICLE

Performance modeling and analysis of parallel Gaussian elimination on multi-core computers

Fadi N. Sibai *

P&CSD Dept. Center, Saudi Aramco, Dhahran 31311, Saudi Arabia

Received 16 October 2012; revised 2 February 2013; accepted 12 March 2013
Available online 20 March 2013

KEYWORDS

Gaussian elimination;
Multi-core computing;
Performance modeling and analysis

Abstract Gaussian elimination is used in many applications and in particular in the solution of systems of linear equations. This paper presents mathematical performance models and analysis of four parallel Gaussian Elimination methods (precisely the Original method and the new Meet in the Middle –MiM– algorithms and their variants with SIMD vectorization) on multi-core systems. Analytical performance models of the four methods are formulated and presented followed by evaluations of these models with modern multi-core systems' operation latencies. Our results reveal that the four methods generally exhibit good performance scaling with increasing matrix size and number of cores. SIMD vectorization only makes a large difference in performance for low number of cores. For a large matrix size ($n \geqslant 16$ K), the performance difference between the MiM and Original methods falls from 16× with four cores to 4× with 16 K cores. The efficiencies of all four methods are low with 1 K cores or more stressing a major problem of multi-core systems where the network-on-chip and memory latencies are too high in relation to basic arithmetic operations. Thus Gaussian Elimination can greatly benefit from the resources of multi-core systems, but higher performance gains can be achieved if multi-core systems can be designed with lower memory operation, synchronization, and interconnect communication latencies, requirements of utmost importance and challenge in the exascale computing age.

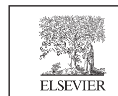
© 2013 Production and hosting by Elsevier B.V. on behalf of King Saud University.

1. Introduction

The large majority of modern microprocessors integrate multiple processing cores on the same package leading to great performance/power ratio gains. Today both central processing units (CPUs) and graphic processing units (GPUs) integrate multiple processing cores. Multi-core processors implement various types of parallelism including instruction-level parallelism, thread-level parallelism, and data-level parallelism. While the trend toward increasing number of cores per processor is strong, it is an interesting and important problem to identify the performance weaknesses of multi-core processors in order to keep the increasing number of cores per processor trend going and avoid the multicore performance "wall." While real application profiling and scalability measurement are accurate performance indicators, they are time consuming in particular when exploring application scalability over a very large number of cores. A more flexible and faster method is analytical performance modeling

* Tel.: +966 3 8808523; fax: +966 3 8758302.
E-mail address: fadi.sibai@aramco.com.

Peer review under responsibility of King Saud University.



Production and hosting by Elsevier

1319-1578 © 2013 Production and hosting by Elsevier B.V. on behalf of King Saud University.
http://dx.doi.org/10.1016/j.jksuci.2013.03.002

which captures the essence of the performance-impacting components of the software application and underlying hardware models, and provides sufficiently accurate answers in particular with regard to trends and key performance impactors. In this paper, we model the performance of parallel Gaussian Elimination (GE) on modern multi-core processors, and derive speedups versus increasing number of cores and derive learnings for future multicore processor designs.

Gaussian elimination (Grama, 2003; Quinn, 1994) is an efficient mathematical technique for solving a system of linear equations. Given the matrix set $A \times X = B$, where X is the variable matrix and A and B are constant matrices, Gaussian elimination (GE) solves for the elements of the X matrix. Gaussian Elimination has also other uses for it is also used in computing the inverse of a matrix. The algorithm is composed of two steps. The first step combines rows eliminating a variable in the process and reducing the linear equation by one variable at a time. This reduction step is repeated until the left side ($A \times X$) is a triangular matrix. The second step is a back substitution of the solved X variables into upper rows until further X variables can be solved.

The algorithm generally works and is stable and can be made more stable by performing partial pivoting. The pivot is the left-most non-zero matrix element in a matrix row available for reduction. When the pivot is zero (non-zero), exchanging the pivot row with another row usually with the largest absolute value in the pivot position may be required. When two rows are combined to reduce the equation by one variable, a multiplication of all elements in the pivot row by a constant and then subtracting all elements in the row by the ones in the other row are performed resulting in a new reduced row. During back substitution, a number of divisions, multiplications and subtractions are performed to eliminate solved variables and solve for a new one.

Partial pivoting works as follows. First the diagonal element in the pivot column with the largest absolute value is located and is referred to as the Pivot. In the Pivot row which is the row containing the pivot, every element is divided by the pivot to get a new pivot row with a 1 in the pivot position. The next step consists of replacing each 1 below the pivot by a 0. This is achieved by subtracting a multiple of the pivot row from each of the rows below it.

Because partial pivoting requires more steps and consumes more time and because without partial pivoting Gaussian Elimination is acceptably stable, we ignore partial pivoting herein and accept to trade off performance for stability.

The numerical instability is proportional to the size of the L and U matrices with known worst-case bounds. For the case without pivoting, it is not possible to provide a priori bounds for Yeung and Chan (1997) provide a probabilistic analysis of the case without pivoting.

Sankar (2004) proved that it is unlikely that A has a large growth factor under Gaussian elimination without pivoting. His results improve upon the average-case analysis of Gaussian elimination without pivoting presented by Yeung and Chan (1997).

Xiaoye and Demmel (1998) proposed a number of techniques in place of partial pivoting to stabilize sparse Gaussian elimination. They propose to not pivot dynamically thereby enabling static data structure optimization, graph manipulation and load balancing while remaining numerically stable. Stability is kept by a variety of techniques: pre-pivoting large elements to the diagonal, iterative refinement, using extra precision when needed, and allowing low rank modifications with corrections at the end.

Geraci (2008) also avoids partial pivoting and recently achieved a performance of 3.2Gflops with a matrix size of 33500 on the Cell Broadband Engine.

Parallel solutions of numerical linear algebra problems are discussed in Demmel (1993), Duff and Van der Vorst (1999). While the Gaussian Elimination method yields an exact solution to the $A \times X = B$ problem, iterative techniques trade off execution time for solution accuracy. Parallel iterative techniques are discussed in Barrett (1994), Greenbaum (1997), Saad (2003), Van der Vorst and Chan (1997). These techniques start with initial conditions for the X vector and keep refining the X solution with each iteration until the X vector converges to a solution vector. The different selections of the initial values of the X vector result in a variety of preconditioning techniques with various performance yields.

Further performance gains can be achieved when the matrices are sparse with many zero elements. Sparse matrix computation techniques are discussed in Heath (1997), Gupta (1997), Gupta et al. (1998), Demmel et al. (1999).

In this paper, we analyze the performance of Parallel Gaussian Elimination and an improved version on multi-core computers with increasing number of cores, and with and without SIMD vectorization. Section 2 reviews relevant literature. Section 3 presents the multi-core processor model on which the parallel GE algorithms are executed. Section 4 reviews the parallel GE algorithm and presents a new improved version: *Meet in the Middle* algorithm. Section 5 presents the mathematical performance model of the parallel GE algorithm. Section 6 presents the mathematical performance model of the improved parallel GE algorithm version. Section 7 states the assumptions made in this performance modeling study. Section 8 presents the parallel GE algorithms' performance results on the multicore processor model with 4–16 K cores, and their analysis. The paper concludes in Section 9.

2. Related background

Various performance analysis tools and methods are available (Prinslow, 2011). Profiling and scalability measurements of real application performance lead to accurate results. Application profiling can be achieved by tools such as Intel VTune, AMD Code Analyst, or GNU gprof. However this requires implementation of the application, and running it on various machines with various configurations, such as processors with various number of cores, a process which consumes long time and extensive resources. More importantly, this approach cannot be adopted in our study as we wish to analyze the application scalability up to 16 K cores, and current hardware limits our exploration to only about 10 cores. Benchmarks are also very efficient in studying computer performance. However they cannot be used in our study for the same reason as real applications: lacking scalable hardware. Simulators can provide up to cycle accurate performance but they require a very long time to develop and are a suitable approach when the simulator is intended to be used extensively. Analytic performance modeling provides a quick and fairly accurate method for identifying trends and the key performance impactors. It is flexible and allows for very large of cores to be modeled, more than currently available in the market. For these reasons we employ analytic performance modeling in this study.

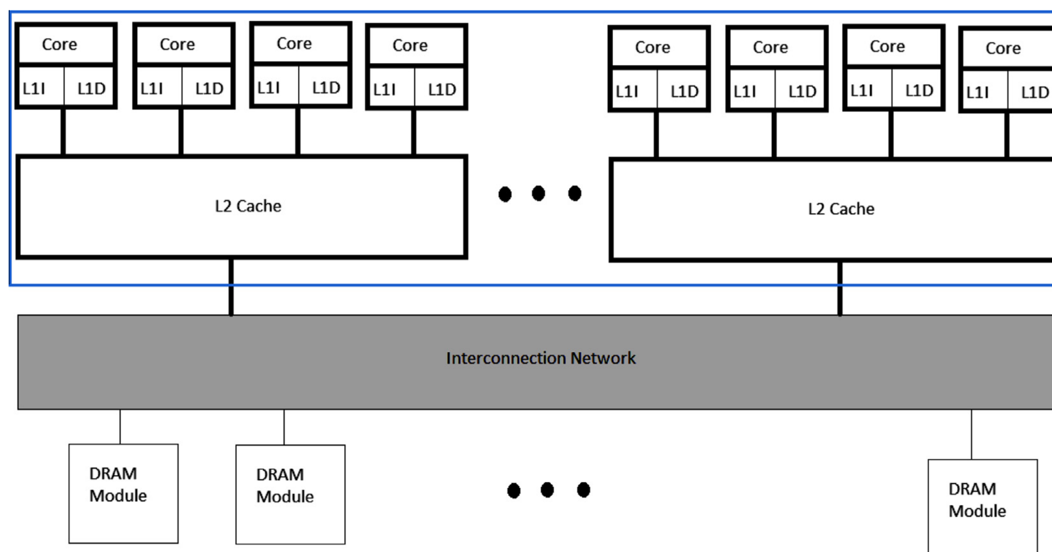


Fig. 1 Multi-core processor model.

Many studies of linear solver performance on multi-core computers were conducted due the importance of both linear solvers used several scientific and engineering applications and multi-core computing. Wiggers et al. (2007) implemented the Conjugate Gradient algorithm in Intel Woodcrest and nVIDIA G680 GPU. The performance analysis on multi-core processors is interesting as sparse matrices show no temporal locality and limited data locality, and miss the cache memories and require access from main memory. Furthermore, sparse matrix vector multiplication perform many more slower memory loads than floating point instructions leading to exposing the "memory wall" problem and exploring ways to improve the utilization of floating point execution units. Li et al. (2011) investigated sparse linear solver performance on multi-core units for power applications. They found the application to be highly memory-bounded and to benefit from GPU acceleration when the matrices are well conditioned and large. Fernández et al. (2010) presented also parallelized sparse matrix multiplication and Conjugate gradient method on multicore processor.

Pimple and Sathe (2011) investigate cache memory effects and cache optimizations of three algorithms including Gaussian Elimination on Intel dual core, 12 core and 16 core machines. Their study emphasizes the importance of data cache management as the cache hierarchy gets deeper, and of cache-aware programming. They found exploiting the L2 cache affinity to be important in light of the L2 being shared by several application threads. Another study investigated parallel Gaussian Elimination with OpenMP and MPI (McGinn and Shaw, 2002), another investigated parallel GE with OpenMP (Kumar and Gupta, 2012) with scaling performance which was limited by the dual core processor employed in the study. Laure and Al-Shandawely (2011) implemented GE on a multicore system and discovered that their algorithm led to false sharing cache effects on pivots and locks arrays. For large matrices, their cache memory was often evicted in each iteration with no future data reuse. Their improved version made more pivots available for each column elimination step while the column data are still in the cache memory.

In this paper we seek to study the scalability of parallel GE on a multi-core computer with a higher number of cores than previously investigated and to shed light on key multi-core system performance issues to focus on in future designs. In that regard, this work is original.

3. Multi-core processor model

The 5 GE implementations (serial, parallel, parallel with SIMD, MIM, MIM with SIMD) which we present in the following Section are assumed to execute on the multi-core processor model of Figure 1. The processor includes multiple cores which communicate via shared memory. Each processing core integrates first level instruction (L1I) and data (L1D) caches which connect to a larger unified second level cache memory (L2) shared by 4 cores. The cores and L2 cache memories are part of the multi-core processor block of Figure 1(in blue).[1]

The processor's L2 cache memory modules connect to external dynamic random access memory (DRAM) modules via an interconnection network. Higher cache memory levels (such as third level) do not change the performance results based on uniform data communication time assumption in the next Section. The serial GE implementation runs on a single core while the other cores remain idle. The parallel GE implementations (with or without SIMD) execute data partitions on multiple cores simultaneously to cut the execution time down.

The processor executes operations varying from arithmetic and logical, to memory transfers to input/output operations. Examples of memory transfer operations include memory reads and writes. Examples of arithmetic and logical operations include add, subtract, multiply, and divide. Each operation time (or latency) is expressed in terms of processor cycles where one cycle is the inverse of the processor frequency. For instance, for a 2 GHz processor, the cycle time equals 0.5 ns.

Each core contains arithmetic and logic execution times for executing the arithmetic and logic operations. When these execution units are SIMDised (SIMD = Single Instruction Multiple Data), the same arithmetic and logical operation can execute different data portions at the same time (as if it

[1] For interpretation of color in Figure 1, the reader is referred to the web version of this article.

operates on a data vector), also reducing the execution time. For instance, four or more adds of different data can be performed simultaneously on the SIMD execution unit of each core.

4. Parallel GE algorithms

Given a system of linear equations $A \times X = B$ where A is an $n \times n$ matrix of constants, X is a $n \times 1$ matrix of unknown variables, and B is an $n \times 1$ matrix of constants, the augmented matrix can be obtained by merging the B matrix with the A matrix to facilitate representation and consolidate storage. For instance when n is six, the augmented matrix is structured as follows

$$\begin{bmatrix} a11 & a12 & a13 & a14 & a15 & a16 & | & b1 \\ a21 & a22 & a23 & a24 & a25 & a26 & | & b2 \\ a31 & a32 & a33 & a34 & a35 & a36 & | & b3 \\ a41 & a42 & a43 & a44 & a45 & a46 & | & b4 \\ a51 & a52 & a53 & a54 & a55 & a56 & | & b5 \\ a61 & a62 & a63 & a64 & a65 & a66 & | & b6 \end{bmatrix} \tag{1}$$

The B matrix appears as the seventh column in the above augmented matrix. This augmented matrix is a compact representation of the following system of linear equations

$$\begin{bmatrix} a11 & a12 & a13 & a14 & a15 & a16 \\ a21 & a22 & a23 & a24 & a25 & a26 \\ a31 & a32 & a33 & a34 & a35 & a36 \\ a41 & a42 & a43 & a44 & a45 & a46 \\ a51 & a52 & a53 & a54 & a55 & a56 \\ a61 & a62 & a63 & a64 & a65 & a66 \end{bmatrix} \times \begin{bmatrix} x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \end{bmatrix} = \begin{bmatrix} b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \end{bmatrix} \tag{2}$$

where $x1, \ldots, x6$ are the unknown variables to be solved. A pair of rows can be combined together eliminating the $x1$ variable in the process and yielding $n - 1$ unknown X variables. Given that

(i) there are s processing cores in a multi-core systems;
(ii) each pair of rows already yields $n - 1$ unknown variables (after 1 reduction);
(iii) $n - 1$ pairs of rows are needed to solve the remaining $n - 1$ variables, x2, ... ,xn; and
(iv) n is a power of 2 for simplicity,

the augmented matrix can be partitioned into $\left\lceil \frac{(n-1)}{s} \right\rceil$ sets of pairs of augmented rows where each set is assigned to a core for processing (reduction).

For instance, in the example of (1), the $6 - 1 = 5$ pairs of rows (1, 2), (3, 4), (5, 6), (2, 3), and (4, 5) can be assigned and distributed among the five cores as shown in Figure 2.

On the top of Figure 2 is the augmented matrix. Below the augmented matrix are the pairs of augmented row partitions allocated to each core. As a result of reduction, each pair of rows results in one reduced augmented row with $n - 1$ unknown variables. Each core keeps its reduced augmented row and communicates its reduced augmented row to its right neighbor (the rightmost core communicates its reduced augmented row to the leftmost core). As each core now has a pair of reduced augmented rows, the one it kept from the previous iteration and the one communicated by its left neighbor (both reduced rows with a "0" in the leftmost "reduced" position), another round of reduction can start in the next position represented by a "•". The unprocessed positions to be processed in the future are represented by a ".". These steps are repeated until the augmented matrix consists of all zeros and a single "•".

The back substitution step can be parallelized but we keep the serial version which runs on the leftmost core 1 which already has solutions of all unknown $(x1, \ldots, xn)$ variables. In

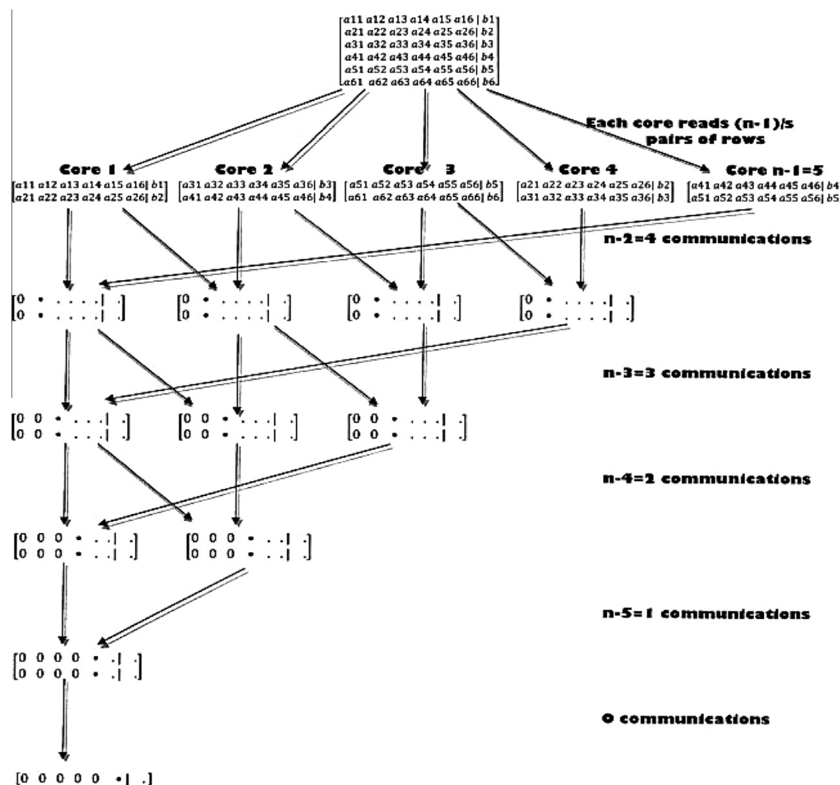


Fig. 2 Reduction: data row partitioning and distribution among the cores.

the serial version of back substitution no communication or synchronization calls are required among the cores as all substitutions occur serially in reverse order from n down to 1 on the same core. We refer to this GE algorithm with serial back substitution as the Original method. The performance of the Original method is limited by the serial back substitution time as dictated by Amdahl's law.

One way to improve the performance of the serial back substitution is to perform it on two cores, one solving for x_n down to $x_{n/2}$, and the other simultaneously solving for x_n to $x_{n/2-1}$. As the two cores solve for the two halves concurrently and in opposite directions and meet at the $n/2$ position, we refer to this technique as the *meet in the middle* (MiM) method.

Given that the two cores involved in the back substitution must have different solution sets and given that cores are plentiful in the multi-core era and that communication and synchronization operations are very costly, we choose to perform MiM as follows. The s cores are divided in two equal sets, where both sets of $s/2$ cores redundantly perform the reduction sets in order for one core in each set to have the solutions of all $n \times$ variables. The two sets of cores perform the reduction on augmented matrices in opposite orders, one with rows 1 to n, and the other with rows n down to 1. This one core in each set (specifically, cores 1 and $n/2$, respectively) then performs the serial back substitution, one solving for x_n down to $x_{n/2}$, and the other solving for x_n to $x_{n/2-1}$. Recall that in case the total number of cores $s > n-1$, only $n-1$ cores are used and involved in GE while the rest remain idle or can be used to run other workloads.

Figure 3 shows the example for MiM with 10 cores. $n-1 = 6$ cores are needed in each set to initially process the six pairs of augmented rows. Cores 1–5 perform the reduction on the augmented matrix starting with row 1 and ending with row 6. Cores 6–10 perform the reduction on the augmented matrix starting with row 6 and ending with row 1. Reduction is performed as in Figure 2. At the end of reduction, Cores 1 and 6 have all reduced augmented rows and can solve for all n unknown x variables. However, Core 1 only solves for x_1 up to x_3, while Core 6 simultaneously solves for x_6 down to x_4. Note that no cores between the two sets need to communicate or synchronize during the back substitution (except at the end) resulting in a big performance savings. Communication and synchronization are only required between cores in the same set as in the original method. These operations in each set take place concurrently. This performance enhancement occurs at the expense of doubling the number of cores needed to perform reduction but this is a low cost to pay in the era of many core processors. Another advantage of this MiM method is its programming simplicity.

5. Performance modeling: serial and parallel algorithms

In this Section and the next Section, we model the performance of both Original and MiM versions of Parallel Gaussian Elimination on a multi-core computer with and without SIMD vectorization. In this Section, we focus on the performance modeling of the serial and parallel GE algorithms.

For each $n \times (n+1)$ augmented matrix, a number of reductions combining each a pair of augmented matrix rows representing linear equations, followed by a number of substitutions to solve for unknown variables $x_1, \ldots, x_{n-1}$ take place. We assume that n is a power of 2, and that during back substitution any necessary reading of the parameters of previously reduced rows from the local cache also overlaps with the external communication.

One reduction is composed of

$$(n+1) \text{ multiplications} \times 2 \text{ rows} + (n+1) \text{ subtractions.}$$

where the "+1" in "$(n+1)$" corresponds to the "b" (rightmost) coefficient of each row. Under SIMD vectorization, and assuming that n is a multiple of 4, this number reduces to

$$(n/4+1) \text{ multiplications} \times 2 \text{ rows} + (n/4+1) \text{ subtractions.}$$

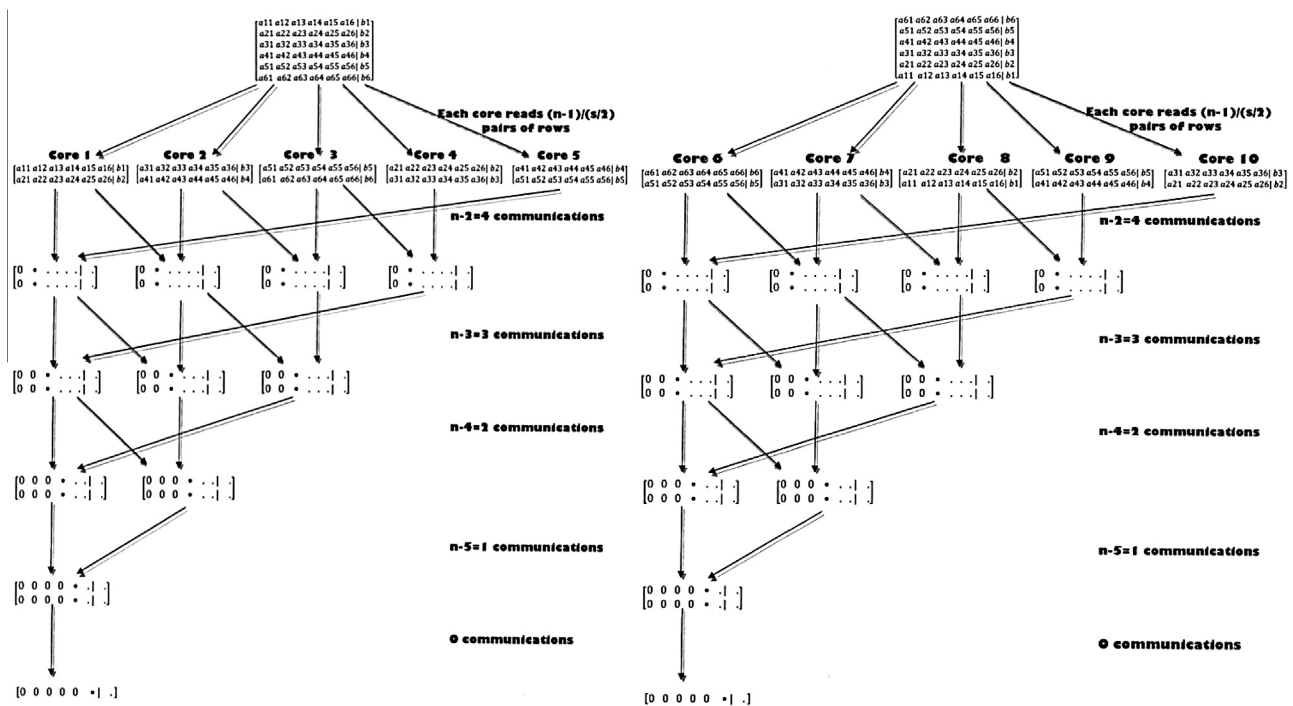


Fig. 3 Meet in the middle reduction: distribution among 10 cores.

In the serial solution with 1 core, and since $n-1$ pairs of rows are initially required (as the Gaussian Elimination technique requires $n-1$ pairs of augmented rows to be processed first, given that the initial combining of a pair of augmented rows results in the elimination of 1 variable), the total reduction time, TR, is related to the total number of reductions (NR) and is given by

$$\begin{aligned} TR_{serial} &= NR = (n-1) + (n-2) + \cdots + 1 \\ &= \sum_{i=1}^{n-1} i = \frac{n \times (n-1)}{2} \text{ reductions.} \end{aligned} \tag{3}$$

On the parallel solution with s cores, and assuming that the pairs of rows are evenly distributed over all cores such that

$$s = \min(\text{number of cores}, n-1) \text{ cores} \tag{4}$$

the overlapping of reductions in time modifies TR to

$$\begin{aligned} TR_{parallel} &= \left\lceil \frac{(n-1)}{s} \right\rceil + \left\lceil \frac{(n-2)}{s} \right\rceil + \cdots + \left\lceil \frac{1}{s} \right\rceil \\ &= \sum_{i=1}^{n-1} \left\lceil \frac{i}{s} \right\rceil \text{ reductions} \end{aligned} \tag{5}$$

as s simultaneous reductions overlap in time on s cores.

Expanding reductions in Eq. (3), in the serial case, the total time, TR_{serial}, involves

$$\begin{aligned} TR_{serial} &= \frac{n \times (n-1)}{2} \times 2 \times (n+1) \text{ multiplications} \\ &\quad + \frac{n \times (n-1)}{2} \times (n+1) \text{ subtractions} \\ &= n \times (n^2-1) \text{ multiplications} \\ &\quad + \frac{n \times (n^2-1)}{2} \text{ subtractions} \end{aligned} \tag{6}$$

In the parallel case, $TC_{parallel}$, involves the overlapping of reductions in time as follows

$$\begin{aligned} &\sum_{i=1}^{n-1} \left\lceil \frac{i}{s} \right\rceil \times 2 \times (n+1) \text{ multiplications} + \sum_{i=1}^{n-1} \left\lceil \frac{i}{s} \right\rceil \times (n+1) \\ &\text{subtractions} = \left\{ 2 \times \frac{(n-1) \times (n-1+s)}{2s} \times (n+1) \right. \\ &\left. \text{multiplications} + \frac{(n-1) \times (n-1+s)}{2s} \times (n+1) \text{ subtractions} \right\} \end{aligned} \tag{7}$$

as $\sum_{i=1}^{n-1} \left\lceil \frac{i}{s} \right\rceil = s \times \sum_{i=1}^{(n-1)/s} i = \frac{(n-1) \times (n-1+s)}{2s}$.

The substitution time, TS, is related to the total number of sequential substitutions performed on all cores (as illustrated in Figure 2), is given by

$$\begin{aligned} TS &= 1 \text{ division} + (2 \text{ divisions} + 1 \text{ multiplication} + 1 \text{ subtraction}) \\ &\quad + (3 \text{ divisions} + 2 \text{ multiplications} + 2 \text{ subtractions}) \\ &\quad + (4 \text{ divisions} + 3 \text{ multiplications} + 3 \text{ subtractions}) \\ &\quad + \cdots + (n \text{ divisions} + (n-1) \text{ multiplications} \\ &\quad + (n-1) \text{ subtractions}) = \sum_{i=1}^{n} ni \text{ divisions} \\ &\quad + \sum_{i=1}^{n-1} i \text{ multiplications} + \sum_{i=1}^{n-1} i \text{ subtractions}) \\ &= \frac{n \times (n+1)}{2} \text{ divisions} + \frac{n \times (n-1)}{2} \text{ multiplications} \\ &\quad + \frac{n \times (n-1)}{2} \text{ subtractions} \end{aligned} \tag{8}$$

On the parallel algorithm, TS is also given by (8) as back substitution is also serial. On the serial version, no inter-core communication is required. On the parallel version, the total number of communications required, NC, is given by

$$\begin{aligned} NC &= \sum_{i=1}^{n-1} \min(i-1, s) \text{ communications} \\ &= \sum_{i=2}^{n-1} \min(i-1, s) \text{ communications} \end{aligned} \tag{9}$$

as there are $n-1$ passes, and pairs of augmented rows on the same core can combine with another pair of rows in the same core (no inter-core communication or memory reads are required on the same core), but the last pair in the partition must communicate with a pair in the next core.

As these parallel inter-core data communications all overlap in time, the net number of communication operations is given by

$$NC = \sum_{i=2}^{n-1} 1 = \frac{nx(n-1)}{2} - 1 \tag{10}$$

parallel communications starting with $(n-1)$ floating point numbers each, down to just one floating point number in the last communication operation.

Therefore, at each step i, the communication size is $(n-i+1)$ single precision floating point numbers as the row size (in units of number of single precision Floating Point numbers) shrinks with each step, and only one row is communicated as a pair of rows is combined by reduction. These numbers are communicated by the source core writing them to memory and then the destination core reading them from memory.

Data communications over the interconnection network are assumed to be pipelined with a bandwidth of 2 single precision floating-point numbers per clock after a communication latency paid on the first pairs of FP numbers communicated. In the worst case in large systems, data communications, reads and writes are not overlapped. Including at each step a synchronization operation time, the total data communication time required is thus given by

$$\begin{aligned} TC_{parallel} &= NC \\ &\times \left(\text{communication latency} + \left\lceil \frac{\text{communication size}}{2} \right\rceil - 2 \text{ cycles} \right) + NC \\ &\times (\text{synchronization time} + \lceil \text{communication size} \rceil \text{ memory writes} \\ &+ \lceil \text{communication size} \rceil \text{ memory reads}) = \left(\frac{n \times (n-1)}{2} - 1 \right) \\ &\times \{\text{communication latency} - 2 + \text{synchronization time}\} + \left\lceil \sum_{i=1}^{n-1} 1 \times (n-i+1) \right\rceil \\ &\text{writes} + \left\lceil \sum_{i=1}^{n-1} 1 \times (n-i+1) \right\rceil \text{reads} + \left\lceil \frac{\sum_{i=1}^{n-1} 1 \times (n-i+1)}{2} \right\rceil \text{cycles} \\ &= \left(\frac{n \times (n-1)}{2} - 1 \right) \times \{\text{communication latency} - 2 + \text{synchronization time}\} \\ &+ \left\lceil \sum_{i=1}^{n-1} (n+1) - \sum_{i=1}^{n-1} i \right\rceil \text{writes} + \left\lceil \sum_{i=1}^{n-1} (n+1) - \sum_{i=1}^{n-1} i \right\rceil \text{reads} \\ &+ \left\lceil \frac{\sum_{i=1}^{n-1} (n+1) - \sum_{i=1}^{n-1} i}{2} \right\rceil \text{cycles} = \left(\frac{n \times (n-1)}{2} - 1 \right) \times \{\text{communication latency} \\ &-2 + \text{synchronization time}\} + \left\lceil \frac{(n+2) \times (n-1)}{2} \right\rceil \text{writes} \\ &+ \left\lceil \frac{(n+2) \times (n-1)}{2} \right\rceil \text{reads} + \left\lceil \frac{(n+2) \times (n-1)}{4} \right\rceil \text{cycles} \end{aligned} \tag{11}$$

where cycle is the inverse of the processor frequency. In addition, each computation requires a reading of $(n-1)$ pairs $\times(n+1)$ augmented row numbers $= 2(n-1)(n+1)$ floating point numbers. As memory reads are pipelined, the total input data read operations consume

Read latency $+ 2\times(n-1)\times(n+1)-1$ cycles.

In addition each computation run requires a writing of n floating point numbers (corresponding to the X vector solution) consuming

Write latency $+ n-1$ cycles.

Thus $TT_{parallel}$, the total time of the parallel Gaussian Elimination (original method), is given by

$$\begin{aligned}TT_{parallel} &= TR_{parallel}+TS+TC=\Big\{2\times\frac{(n-1)\times(n-1+s)}{2\times s}\\&\times(n+1)\text{ multiplications}+\frac{(n-1)\times(n-1+s)}{2\times s}\\&\times(n+1)\text{subtractions}\Big\}+\Big\{\frac{n\times(n+1)}{2}\text{divisions}\\&+\frac{n\times(n-1)}{2}\text{multiplies}+\frac{n\times(n-1)}{2}\text{subtractions}\Big\}\\&+\Big\{\left(\frac{n\times(n-1)}{2}-1\right)\times(\text{communication latency}\\&-2+\text{synchronization time})+\left\lceil\frac{(n+2)\times(n-1)}{2}\right\rceil\\&\text{writes}+\left\lceil\frac{(n+2)\times(n-1)}{2}\right\rceil\text{reads}\\&+\left\lceil\frac{(n+2)\times(n-1)}{4}\right\rceil\text{cycles}\Big\}\end{aligned}\tag{12}$$

where the ceiling function is applied to all fractions although it may not be shown.

In addition, $TT_{parallel}$ is augmented by

Read latency $+ 2\times(n-1)\times(n+1)-1+$ write latency $+n-1 =$ Read latency $+ 2\times(n-1)\times(n+1)+$ write latency $+ n-2$ cycles for reading the augmented matrix data and writing the output vector X.

When SIMD vectorization is used with 4 simultaneous subtractions, multiplies, or divisions at a time, NC_{SIMD} is unchanged from NC, but NR_{SIMD} and NS_{SIMD} are modified as follows

$$\begin{aligned}NR_{SIMD} &= \Big\{\sum_{i=1}^{n-1}\left\lceil\frac{i}{s}\right\rceil\times\left\lceil\frac{2\times(n+1)}{4}\right\rceil\text{simultaneous SIMD multiplications}\\&+\sum_{i=1}^{n-1}\left\lceil\frac{i}{s}\right\rceil\times\left\lceil\frac{(n+1)}{4}\right\rceil\text{simultaneous SIMD subtractions}\Big\}\end{aligned}\tag{13}$$

$$\begin{aligned}NS_{SIMD} &= \sum_{i=1}^{n}\left\lceil\frac{i}{4}\right\rceil\text{simultaneous SIMD divisions}\\&+\sum_{i=1}^{n-1}\left\lceil\frac{i}{4}\right\rceil\text{simultaneous SIMD multiplies}\\&+\sum_{i=1}^{n-1}\left\lceil\frac{i}{4}\right\rceil\text{ simultaneous SIMD subtractions}\\&=\frac{n\times(n+4)}{8}\text{ simultaneous SIMD divisions}\\&+\frac{(n-1)\times(n+3)}{8}\text{ simultaneous multiplies}\\&+\frac{(n-1)\times(n+3)}{8}\text{ simultaneous SIMD subtractions}\end{aligned}\tag{14}$$

as

$$\sum_{i=1}^{n}\left\lceil\frac{i}{4}\right\rceil = 4\times\sum_{i=1}^{n/4}i = 4\times\frac{\frac{n}{4}\times\left(\frac{n}{4}+1\right)}{2}=\frac{n\times\left(\frac{n}{4}+1\right)}{2} = \frac{n\times(n+4)}{8}.$$

Thus with SIMD vectorization, the total time equals the total number of operations required and is given by

$$\begin{aligned}TT_{SIMD} &= TR_{SIMD}+TS_{SIMD}+TC_{SIMD}=\Big\{\sum_{i=1}^{n-1}\left\lceil\frac{i}{s}\right\rceil\\&\times\left\lceil\frac{2\times(n+1)}{4}\right\rceil\text{simultaneous SIMD multiplications}\\&+\sum_{i=1}^{n-1}\left\lceil\frac{i}{s}\right\rceil\times\left\lceil\frac{(n+1)}{4}\right\rceil\text{simultaneous SIMD subtractions}\Big\}\\&+\Big\{\frac{n\times(n+4)}{8}\text{simultaneous SIMD divisions}\\&+\frac{(n-1)\times(n+3)}{8}\text{simultaneous multiplies}\\&+\frac{(n-1)\times(n+3)}{8}\text{simultaneous SIMD subtractions}\Big\}\\&+TC_{SIMD}=\Big\{\frac{(n-1)\times(n-1+s)}{2\times s}\times\left\lceil\frac{2\times(n+1)}{4}\right\rceil\\&\textit{simultaneous SIMD multiplications}\\&+\frac{(n-1)\times(n-1+s)}{2\times s}\times\left\lceil\frac{(n+1)}{4}\right\rceil\\&\text{simultaneous SIMD subtractions}+\Big\{\frac{n\times(n+4)}{8}\\&\text{simultaneous SIMD divisions}+\frac{(n-1)\times(n+3)}{8}\\&\text{simultaneous multiplies}+\frac{(n-1)\times(n+3)}{8}\\&\text{simultaneous SIMD subtractions}\}\\&+\Big\{\left(\frac{n\times(n-1)}{2}-1\right)\times(\text{communication latency}-2\\&+\text{synchronization time})+\left\lceil\frac{(n+2)\times(n-1)}{2}\right\rceil\text{writes}\\&+\left\lceil\frac{(n+2)\times(n-1)}{2}\right\rceil\text{reads}+\left\lceil\frac{(n+2)\times(n-1)}{4}\right\rceil\text{cycles}\Big\}\end{aligned}\tag{15}$$

in addition to read latency $+ 2\times(n-1)\times(n+1)+$ write latency $+ n-2$ cycles.

6. Meet in the middle

The meet in the middle method consists of top down back substitutions performed simultaneously with bottom up back substitutions until both substitution streams meet in the middle (at $x_{n/2}$). It is assumed that $s/2$ (a whole number) cores work top down and the remaining $s/2$ cores work bottom up, and both sets of cores perform simultaneous reductions but in reverse orders, and then perform serial substitutions again in reverse order until they meet in the middle, i.e., the top down set of cores solves $x_1,\ldots,x_{n/2}$, while the bottom up set of cores solves $x_{n/2+1},\ldots,x_n$.

In this case, at the start, each core reads $(n-1)/(s/2)$ pairs of augmented rows. The reductions are performed twice in parallel by all cores but in opposite orders of the X vectors in each set, and the substitutions are performed serially on 2 processors, one in each set, one processor solving for $x_1\ldots,x_{n/2}$ while the second one simultaneously solving for $x_{(n/2)+1}\ldots x_n$.

Therefore, with no SIMD vectorization, the total number of operations on s cores equals the total number of operations in top down and the total number of operations in bottom up, as follows

$$\begin{aligned}\mathrm{NO_{MiM}} &= \mathrm{NO_{top\text{-}down}} + \mathrm{NO_{bottom\text{-}up}} = (\mathrm{NR}+\mathrm{NS}+\mathrm{NC})_{\text{top-down}} \\ &+ (\mathrm{NR}+\mathrm{NS}+\mathrm{NC})_{\text{bottom-up}} = (\mathrm{NR_{top\text{-}down}} \\ &+ \mathrm{NS_{top\text{-}down}} + \mathrm{NC_{top\text{-}down}}) + (\mathrm{NR_{bottom\text{-}up}} \\ &+ \mathrm{NS_{bottom\text{-}up}} + \mathrm{NC_{bottom\text{-}up}}) \qquad (16)\end{aligned}$$

where

$$\mathrm{NR_{top\text{-}down}} = \mathrm{NR_{bottom\text{-}up}}$$
$$\mathrm{NS_{top\text{-}down}} = \mathrm{NS_{bottom\text{-}up}}$$
$$\mathrm{NC_{top\text{-}down}} = \mathrm{NC_{bottom\text{-}up}}$$

as now each reduction involves half or $s/2$ cores compared to before. The number of substitutions in both sets is given by

$$\begin{aligned}\mathrm{NS_{top\text{-}down}} = \mathrm{NS_{top\text{-}down}} &\approx \sum_{i=1}^{\frac{n}{2}} i \text{ divisions} + \sum_{i=1}^{\frac{n}{2}-1} i \text{ multiplies} \\ &+ \sum_{i=1}^{\frac{n}{2}-1} i \text{ subtractions} = \left\{\frac{n\times(n+2)}{8}\text{divisions}\right. \\ &+ \left.\frac{n\times(n-2)}{8}\text{multiplies} + \frac{n\times(n-2)}{8}\text{subtractions}\right\} \qquad (17)\end{aligned}$$

as each of the two series of substitutions simultaneously cover only half of the data matrix size ($n/2$).

As before but accounting for half the matrix size, the number of communication operations is given by

$$\begin{aligned}\mathrm{NC_{top\text{-}down}} &= \mathrm{NC_{bottom\text{-}up}} \\ &= \left(\frac{\frac{n}{2}\times\left(\frac{n}{2}-1\right)}{2} - 1\right) \\ &\quad \times \{\text{communication latency} - 2 \\ &\quad + \text{synchronization time}\} \\ &\quad + \left\lceil \sum_{i=1}^{\frac{n}{2}-1} 1\times\left(\frac{n}{2}-i+1\right)\right\rceil \text{writes} \\ &\quad + \left\lceil \sum_{i=1}^{\frac{n}{2}-1} 1\times\left(\frac{n}{2}-i+1\right)\right\rceil \text{reads} \\ &\quad + \left\lceil \frac{\sum_{i=1}^{\frac{n}{2}-1} 1\times\left(\frac{n}{2}-i+1\right)}{2}\right\rceil \text{cycles} \\ &= \left(\frac{n\times(n-2)}{8} - 1\right) \\ &\quad \times \{\text{communication latency} - 2 \\ &\quad + \text{synchronization time}\} \\ &\quad + \left\lceil\frac{(n+4)\times(n-2)}{8}\right\rceil \text{writes} \\ &\quad + \left\lceil\frac{(n+4)\times(n-2)}{8}\right\rceil \text{reads} \\ &\quad + \left\lceil\frac{(n+4)\times(n-2)}{8}\right\rceil \text{cycles} \qquad (18)\end{aligned}$$

Therefore, substituting for (17) and (18) in (16) yields

$$\begin{aligned}\mathrm{NO_{MiM}} &= 2\times\left\{\left\{2\times\sum_{i=1}^{\frac{n}{2}-1}\left\lceil\frac{i}{s}\right\rceil\times(n+1)\text{ multiplications}\right.\right. \\ &+ \left.\sum_{i=1}^{\frac{n}{2}-1}\left\lceil\frac{i}{s}\right\rceil\times(n+1)\text{subtractions}\right\} \\ &+ \left\{\frac{n\times(n+2)}{8}\text{divisions} + \frac{n\times(n-2)}{8}\text{multiplications}\right. \\ &+ \left.\left.\frac{n\times(n-2)}{8}\text{subtractions}\right\} + \mathrm{NC_{top\text{-}down}}\right\} \\ &= \left\{4\times s\times\sum_{i=1}^{\left(\frac{n}{2}-1\right)/s} i\times(n+1)\text{multiplications}\right. \\ &+ \left. 2\times s\times\sum_{i=1}^{\left(\frac{n}{2}-1\right)/s} i\times(n+1)\text{subtractions}\right\} \\ &+ \left\{\frac{n\times(n+2)}{4}\text{divisions} + \frac{n\times(n-2)}{4}\text{multiplications}\right. \\ &+ \left.\frac{n\times(n-2)}{4}\text{subtractions}\right\} + \{2\times\mathrm{NC_{top\text{-}down}}\} \\ &= \left\{4\times\frac{(n-2+2s)\times(n-2)}{8\times s}\right. \\ &\times(n+1)\text{multiplications} + 2\times\frac{(n-2+2s)\times(n-2)}{8\times s} \\ &\times\left.(n+1)\text{subtractions}\right\} + \left\{\frac{n\times(n+2)}{4}\text{divisions}\right. \\ &+ \left.\frac{n\times(n-2)}{4}\text{multiplications} + \frac{n\times(n-2)}{4}\text{subtractions}\right\} \\ &+ 2\times\left\{\left(\frac{n\times(n-2)}{8} - 1\right)\right. \\ &\times\{\text{communication latency} - 2 + \text{synchronization time}\} \\ &+ \left\lceil\frac{(n+4)\times(n-2)}{8}\right\rceil\text{writes} + \left\lceil\frac{(n+4)\times(n-2)}{8}\right\rceil\text{reads} \\ &+ \left.\left\lceil\frac{(n+4)\times(n-2)}{8}\right\rceil\text{ cycles}\right\} \qquad (19)\end{aligned}$$

as

$$\sum_{i=1}^{\frac{n}{2}-1}\left\lceil\frac{i}{s}\right\rceil = s\times\sum_{i=1}^{\left(\frac{n}{2}-1\right)/s} i = \frac{(n-2+2s)\times(n-2)}{8\times s}$$

As the top-down and bottom-up operations overlap in time, the total time is half the number of operations in (19), and is given by

$$\begin{aligned}\mathrm{TT_{MiM}} &= \left\{2\times\frac{(n-2+2s)\times(n-2)}{8\times s}\times(n+1)\text{multiplications}\right. \\ &+ \left.\frac{(n-2+2s)\times(n-2)}{8\times s}\times(n+1)\text{subtractions}\right\} \\ &+ \left\{\frac{n\times(n+2)}{8}\text{divisions} + \frac{n\times(n-2)}{8}\text{multiplies}\right. \\ &+ \left.\frac{n\times(n-2)}{8}\text{subtractions}\right\} + \left\{\left(\frac{n\times(n-2)}{8} - 1\right)\right. \\ &\times\{\text{communication latency} - 2 + \text{synchronization time}\} \\ &+ \left\lceil\frac{(n+4)\times(n-2)}{8}\right\rceil\text{writes} + \frac{(n+4)\times(n-2)}{8}\text{ reads} \\ &+ \left.\left\lceil\frac{(n+4)\times(n-2)}{8}\right\rceil\text{cycles}\right\} \qquad (20)\end{aligned}$$

in addition to {read latency + $2 \times (n-1) \times (n+1)$ + write latency + $n-2$ cycles} for reading the input data and writing the output data.

With SIMD vectorization with four simultaneous division, multiplication or subtraction operations at a time, the total number of operations on all s cores become

$$\begin{aligned}
NO_{MiM,SIMD} &= NO_{top\text{-}down,SIMD} + NO_{bottom\text{-}up,SIMD} \\
&= (NR+NS+NC)_{top\text{-}down,SIMD} \\
&+ (NR+NS+NC)_{bottom\text{-}up,SIMD} \\
&= (NR_{top\text{-}down,SIMD} + NS_{top\text{-}down,SIMD} \\
&+ NC_{top\text{-}down,SIMD}) + (NR_{bottom\text{-}up,SIMD} \\
&+ NS_{bottom\text{-}up,SIMD} + NC_{bottom\text{-}up,SIMD}) \\
&= 2 \times \left\{\left\{\sum_{i=1}^{\frac{n}{2}-1} \left\lceil \frac{i}{s} \right\rceil \times \left\lceil \frac{2 \times (n+1)}{4} \right\rceil \right.\right. \\
&\text{simultaneous SIMD multiplications} + \sum_{i=1}^{\frac{n}{2}-1} \left\lceil \frac{i}{s} \right\rceil \\
&\left. \times \left\lceil \frac{(n+1)}{4} \right\rceil \text{simultaneous SIMD subtractions} \right\} \\
&+ \left\{ \sum_{i=1}^{\frac{n}{2}} \left\lceil \frac{i}{4} \right\rceil \text{simultaneous SIMD divisions} \right. \\
&+ \sum_{i=1}^{\frac{n}{2}-1} \left\lceil \frac{i}{4} \right\rceil \text{simultaneous SIMD multiplications} \\
&\left. + \sum_{i=1}^{\frac{n}{2}-1} \left\lceil \frac{i}{4} \right\rceil \text{simultaneous SIMDsubtractions} \right\} \\
&\left. + NC_{top\text{-}down,SIMD} \right\} \\
&= \left\{ 2 \times \frac{(n-2+2s) \times (n-2)}{8 \times s} \right. \\
&\times \left\lceil \frac{2 \times (n+1)}{4} \right\rceil \text{simultaneous SIMD multiplications} \\
&+ 2 \times \frac{(n-2+2s) \times (n-2)}{8 \times s} \\
&\left. \times \left\lceil \frac{(n+1)}{4} \right\rceil \text{simultaneous SIMD subtractions} \right\} \\
&+ \left\{ \frac{n \times (n+8)}{16} \text{ simultaneous SIMD divisions} \right. \\
&+ \frac{(n+6) \times (n-2)}{16} \text{ simultaneous SIMD multiplications} \\
&\left. + \frac{(n+6) \times (n-2)}{16} \text{simultaneous SIMD subtractions} \right\} \\
&+ 2 \times NC_{top\text{-}down,SIMD}
\end{aligned} \tag{21}$$

As the top-down and bottom-up operations overlap in time, the total time is half the number of operationsin addition to

Table 1 Operation latencies.

Operation	Latency (in cycles)
Add, subtract	1
Multiply, divide	4
Memory read	100
Memory write	100
Synchronization between 2 cores	500
Communication latency	64

read latency + $2 \times (n-1) \times (n+1)$ + write latency + $n-2$ cycles as

$$\sum_{i=1}^{n/2} \left\lceil \frac{i}{4} \right\rceil = 4 \times \sum_{i=1}^{n/8} i = 4 \times \frac{\frac{n}{8} \times \left(\frac{n}{8}+1\right)}{2} = \frac{n}{4} \times \left(\frac{n}{8}+1\right) = \frac{n(n+8)}{32}.$$

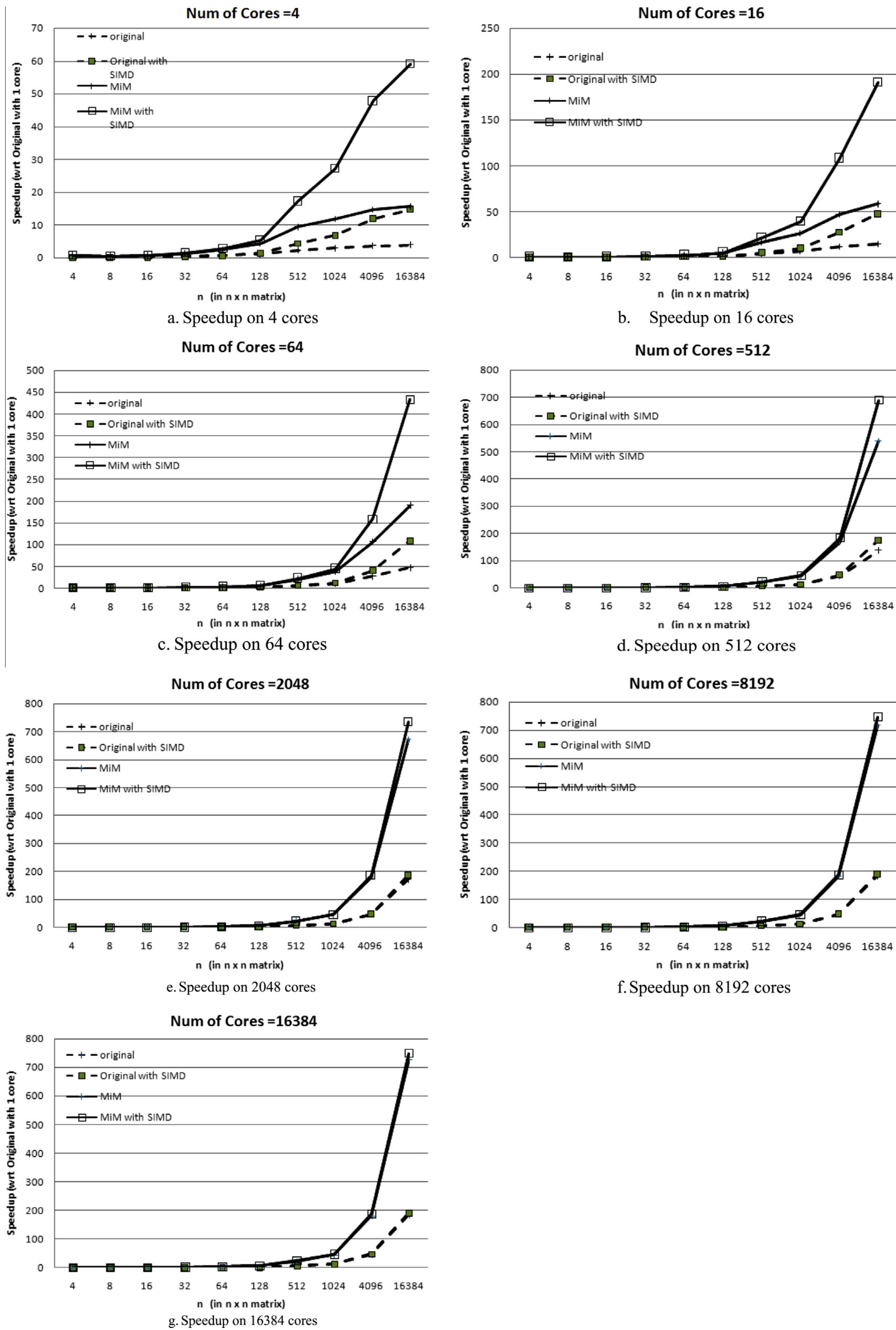
7. Assumptions

The major operation latencies are assumed as shown in Table 1. These values are derived from Fog (2012), Bader and and Agarwal (2007) and based on the following assumptions $a-c$, and are later inserted into the time equations of the original method's serial version (TT_{serial}, Eq. (6)), parallel versions ($TT_{parallel}$, Eq. (11)), and parallel version with SIMD vectorization (TT_{SIMD}, Eq. (15)), MiM (TT_{MiM}, Eq. (20)), and MiM with SIMD vectorization ($TT_{MiM,SIMD}$, Eq. (22)).

Other assumptions follow.

a. It is assumed that the CPU frequency is high (~2 GHz or higher) and that arithmetic and logic instructions are pipelined, leading to add/sub, multiply/divide, read/write, synchronization, and communication latencies of 0.5 ns, 2 ns, 50 ns, 250 ns, 32 ns, respectively for a 2 GHz processor. For simplification, we assume that divisions and memory writes consume equal latencies to multiplications and memory reads, respectively, although the latter operations usually take longer times to complete on modern processors. The multiply and division latency assumptions are near Nehalem CPU's FMUL (5 cycles) and FDIV (7 cycles) (Fog, 2012.
b. All memory reads and writes are assumed to DRAM (and therefore miss all cache memories on the way) so the 100 cycles (50 ns for 2 GHz processor) latency is justified.
c. Bader and and Agarwal (2007) measured the tree-based synchronization with up to 8 synergistic processing elements (SPEs) on the Sony-Toshiba-IBM Cell Broad-

$$\begin{aligned}
TT_{MiM,SIMD} &= TT_{top\text{-}down,SIMD} + TT_{bottom\text{-}up,SIMD} = (TR+TS+TC)_{top\text{-}down,SIMD} + (TR+TS+TC)_{bottom\text{-}up,SIMD} \\
&= (TR_{top\text{-}down,SIMD} + TS_{top\text{-}down,SIMD} + TC_{top\text{-}down,SIMD}) + (TR_{bottom\text{-}up,SIMD} + TS_{bottom\text{-}up,SIMD} + TC_{bottom\text{-}up,SIMD}) \\
&= \left\{ \frac{(n-2+2s) \times (n-2)}{8 \times s} \times \left\lceil \frac{2 \times (n+1)}{4} \right\rceil \text{simultaneous SIMD multiplications} + \frac{(n-2+2s) \times (n-2)}{8 \times s} \times \left\lceil \frac{(n+1)}{4} \right\rceil \text{ simultaneous SIMDsubtractions} \right\} \\
&+ \left\{ \frac{n \times (n+8)}{32} \text{simultaneous SIMD divisions} + \frac{(n+6) \times (n-2)}{32} \text{simultaneous SIMD multiplications} + \frac{(n+6) \times (n-2)}{32} \text{simultaneous SIMD subtractions} \right\} \\
&+ \left\{ \left(\frac{n \times (n-2)}{8} - 1 \right) \times \{\text{communication latency} - 2 + \text{synchronization time}\} + \left\lceil \frac{(n+4) \times (n-2)}{8} \right\rceil \text{writes} + \left\lceil \frac{(n+4) \times (n-2)}{8} \right\rceil \text{reads} + \left\lceil \frac{(n+4) \times (n-2)}{8} \right\rceil \text{cycles} \right\}
\end{aligned} \tag{22}$$



a. Speedup on 4 cores

b. Speedup on 16 cores

c. Speedup on 64 cores

d. Speedup on 512 cores

e. Speedup on 2048 cores

f. Speedup on 8192 cores

g. Speedup on 16384 cores

Fig. 4 Speedup with respect to serial time versus matrix size with various number of cores.

band Engine processor (IBM Corporation) to take 3200 cycles (at 3.2 GHz frequency). On the Pentium 4, LOCK CMPXCHG and MFENCE consume at least 100 cycles each. Thus the assumption that the synchronization between a pair of cores consumes 500 cycles is also reasonable.

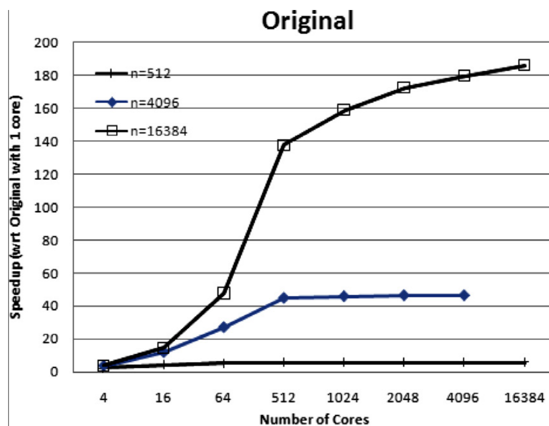
d. For simplification, we assume that synchronization times and communication latencies are constant and independent of the number of cores.
e. With SIMD vectorization, the arithmetic operations take the same amount of time (cycles) except that it is assumed that 4 such operations can be performed simultaneously per core in the above number of cycles.
f. The major operation times are included in the performance model in order to assess their impact on the overall performance scalability with number of cores. Other secondary operations are omitted. The execution times also include the reading of the input data and the writing of the output data times. Dynamic random memory (DRAM) space is assumed to be sufficient to avoid virtual memory effects.
g. Inter-core data communication time is assumed to be uniform, pipelined with 2 single precision floating point numbers communicated per clock after initial communication latency elapses, and is integrated in the model's TC equations (see Eqs. (1)–(22)).
h. The performance models include the effects of thread-level and data-level parallelisms. Thread management times are ignored. Moreover, instruction-level parallelism is ignored in the performance models of both serial and parallel algorithms.

8. Performance results and analysis

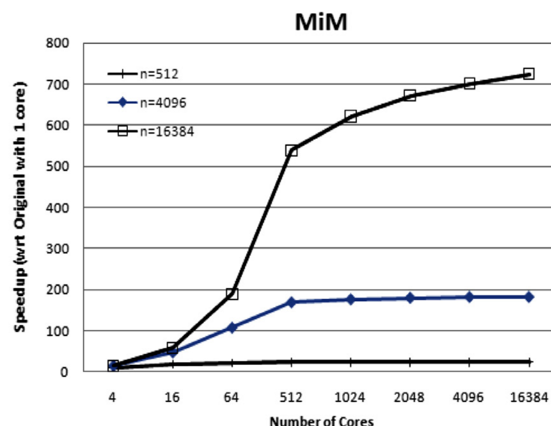
For space reduction, we omit the matrix sizes of $n = 256$, 2048, and 8 K reducing the smoothness of the plots.

The plots of the speedups with respect to the original serial times for various number of matrix sizes (n = 4-16384) and on various number of cores (s = 4-16384) are plotted in Figure 4, for the original parallel method, the original method with SIMD, MiM, and MiM with SIMD. Presently, multicore processors have tens of cores so the upper range of 16384 cores is for future performance projections of future systems, assuming that geometric technological improvements of integrated circuits keep going. The speedup is defined as the ratio of the serial time over the parallel time. For n = 4-16384, the speedups of all 4 parallel methods exhibit a performance loss for $n \leqslant 16$ and a performance gain for $n > 16$. Performance gains of at least 50% occur for MiM and MiM with SIMD for $n \geqslant 32$. Performance gains for the original and original with SIMD methods occur for $n \geqslant 128$ (4 times larger than for MiM and MiM with SIMD). The performance scalings with matrix size n and number of cores s are evident in Figure 4.

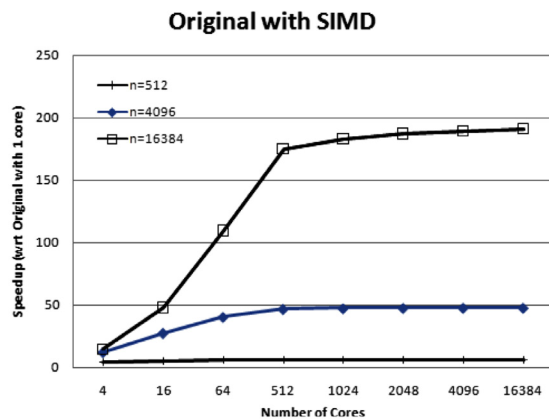
For n and s of 16384, the speedups reach 185.9, 190.7, 723.8, and 748.6 for the original parallel method, the original method with SIMD, MiM, and MiM with SIMD, respectively. These latter three speedups jump to 2000, 5000 and 7000 when no communication and synchronization times are accounted



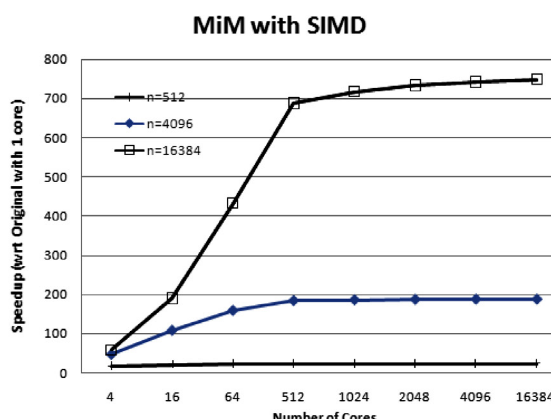
a. Speedup vs. Number of Cores for the Original Method



c. Speedup vs. Number of Cores for the MiM Method



b. Speedup vs. Number of Cores for the Original-with-SIMD Method



d. Speedup vs. Number of Cores for the MiM with SIMD Method

Fig. 5 Speedups vs. the number of cores.

for in the performance model, approximately 10x higher. For low number of cores (4-64), the original method with SIMD and MiM method speedups cross over, near $n = 16\,K$ for $s = 4$, and at higher n's for $s = 16$-64. The cross point moves to the right as the number of cores increases. For larger number of cores (512 and higher), these two curves diverge with MiM showing much better performance of at least 3×. The difference in performance between MiM and MiM with SIMD is large for $n \geqslant 512$ but the performance gap shrinks with increasing number of cores due to increasing communication and synchronization overheads. When s = 2K, the MiM and MiM with SIMD curves start to completely merge and appear to be completely merged for $s > 8192$. Figure 4g still indicates a good performance scaling with increasing matrix size.

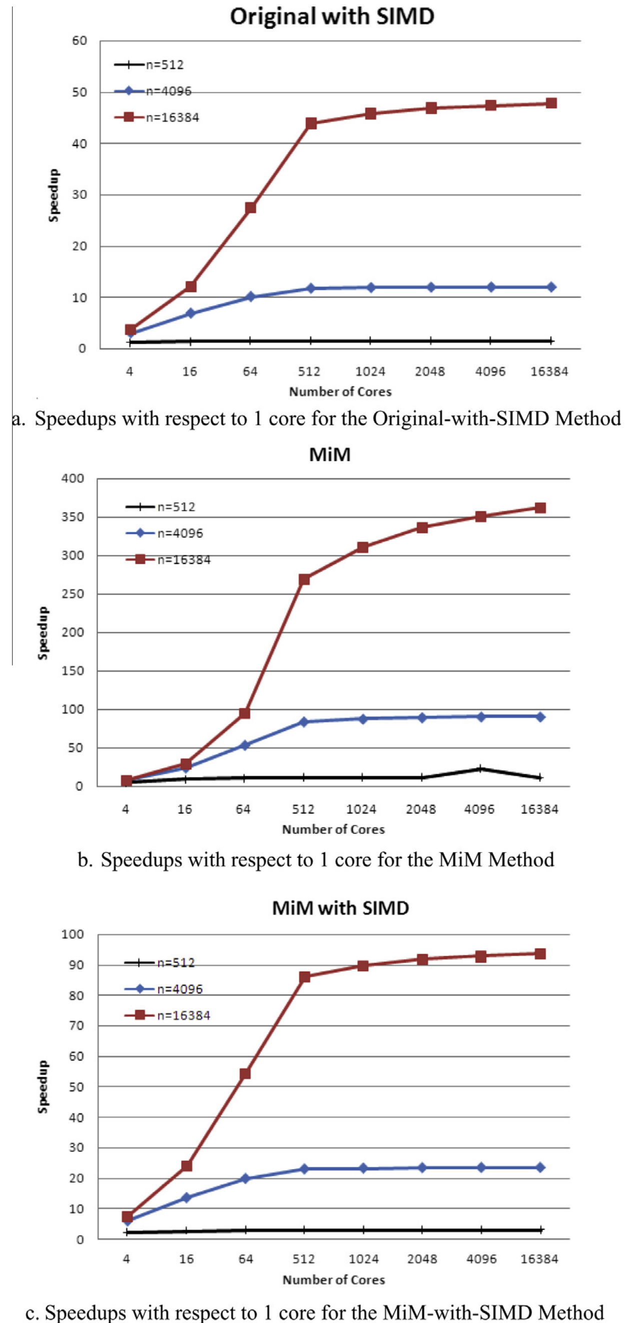
The speedups versus the number of cores are plotted in Figure 5 for the four methods indicating good performance scaling with number of cores. The reader should note that the X axis is not drawn to scale as some n points are missing e.g. $n = 32$, 128, and 8K.

The following observations are made

a. SIMD vectorization makes a large difference for low number of cores (with little communication and synchronization overhead).
b. The difference between MiM and MiM with SIMD performances shrinks with increasing number of cores (with increasing communication and synchronization overhead) and is more evident for 1 K or less cores.
c. With 4 cores and 16 K matrix size, the speedups of the original parallel method, the original method with SIMD, MiM, and MiM with SIMD, are 3.9, 14.8, 15.7 and 59, respectively. SIMD vectorization results in a 277% higher performance for both the original and MiM methods. MiM with SIMD delivers 15x better performance than the original parallel method.
d. With 64 cores and 16 K matrix size, the respective speedups jump to 47.8, 109.5, 189.7 and 433.2. SIMD vectorization now results in only 128% higher performance than without it. MiM with SIMD delivers 9x better performance than the original parallel method.
e. With 512 cores and 16 K matrix size, the respective speedups jump to 137.5, 174.9, 539.2 and 687.7. SIMD vectorization now results in only 27% higher performance than without it. MiM with SIMD delivers 5x better performance than the original parallel method.
f. With 16 K cores and 16 K matrix size, the respective speedups jump to 185.9, 190.7, 723.8 and 748.6. SIMD vectorization now results in only 2.6%-3.4% higher performance than without it. MiM with SIMD still delivers a significant performance (4x better) than the original parallel method.

Figure 6 displays the speedups of the original with SIMD, MiM, and MiM with SIMD methods with respect to 1 core. In Figure 6, the speedup is defined as the ratio of the execution time of the parallel method with 1 core over the execution time of the same method on n cores. These speedups are lower when SIMD vectorization is employed owing to the smaller absolute execution times with SIMD.

The efficiencies defined as the ratio of the speedup over the number of cores are good (i.e., 50% or above) for large n's (generally 16 K and larger, and in all cases at least 128 or larger) for up to 512 cores. For 512 cores, only the MiM efficiency is good for 16 K matrix size or larger. With 128 cores, the efficiency is good for MiM with 4 K matrix sizes and up, and for the original method with matrix sizes of 16 K and up. With 64 cores, the efficiencies of the original and MiM methods are good for matrix sizes of 4 K and up. These matrix sizes lower limits keep going down with the lower number of cores. For instance with 8 cores, the original and MiM methods' efficiencies are good for 1 K and larger matrix sizes for the original method, and 264 and larger for the MiM method. With only 4 cores, the original and MiM methods' efficiencies are good



a. Speedups with respect to 1 core for the Original-with-SIMD Method

b. Speedups with respect to 1 core for the MiM Method

c. Speedups with respect to 1 core for the MiM-with-SIMD Method

Fig. 6 Speedups with respect to 1 core versus the number of cores.

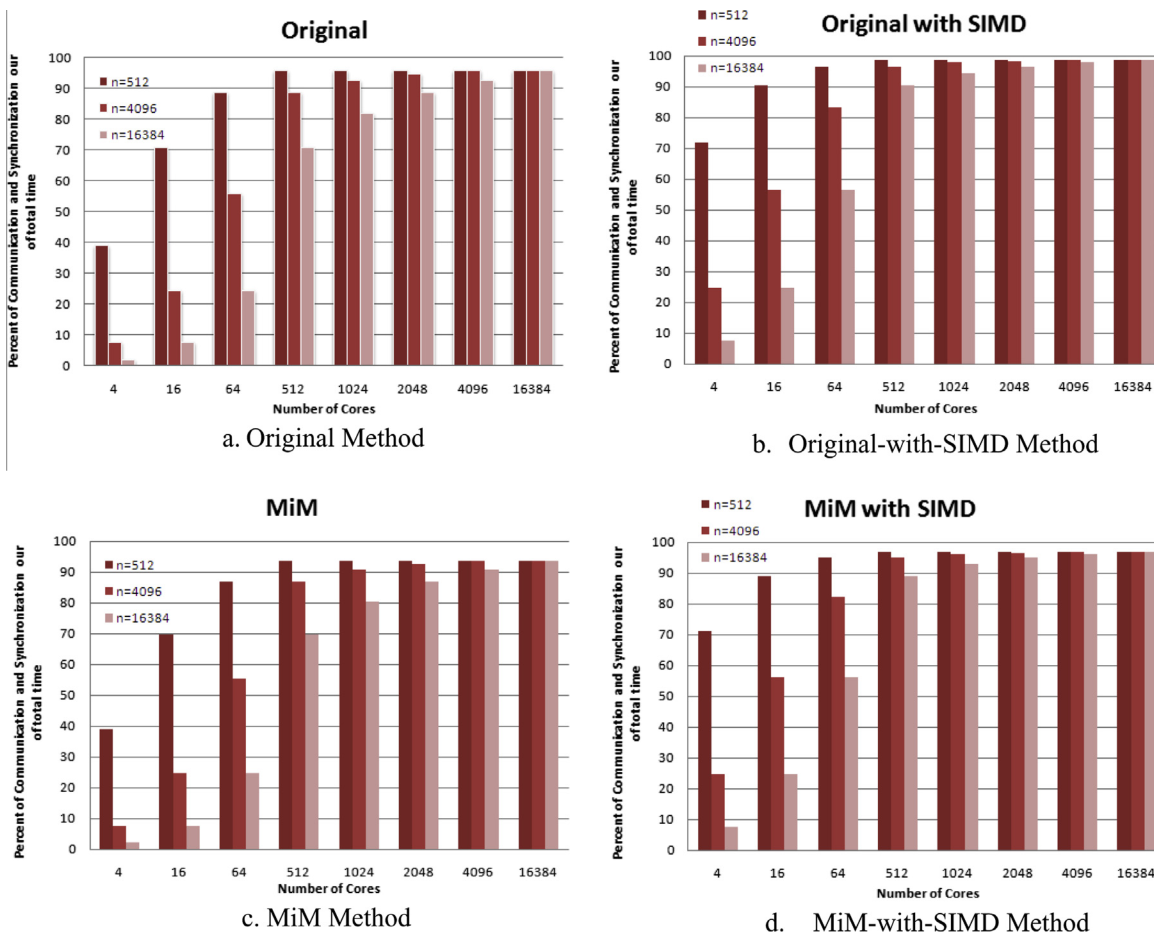


Fig. 7 Percent of communication and synchronization overhead.

for 512 and larger matrix sizes for the original method, and 128 and larger for the MiM method. Generally, the efficiencies of all 4 methods are not good with 1 K cores or more. This is a major problem in multicores and manycores with a large number of cores where the network-on-chip and memory latencies must be kept under control. Although in our performance model we kept the communication and memory read and write latencies constant even with increasing number of cores (in reality, these should increase with longer chip and wire distances and increasing memory hierarchy latencies of larger systems), still the efficiencies with 1 K core or more reveal that communication, synchronization, and memory latencies are major performance limiters in multicores and manycores.

Figure 7 displays the percent of communication (memory operations) and synchronization out of the total execution time for all four parallel methods with increasing number of cores, and for n = 512, 4 K and 16 K. It is evident that the 512 (lower) matrix size curve reaches above 90% levels before the larger matrix size curves (4 K and 16 K)s, as less data entail smaller execution times with larger portions of communication and synchronization. Another observation is that the SIMD vectorization curves (Figs. 7b and d) reach higher levels of communication and synchronization overheads with the same number of cores and the same matrix sizes as their non-SIMD counterparts. This is because with SIMD, the execution times are smaller and the communication and synchronization overheads take a large portion of the pie.

Thus with larger execution times due to larger matrix sizes or no SIMD vectorization, the communication and synchronization portions of the total execution time pie decrease. This results into higher speedups with more cores and higher matrix sizes as seen in Figure 5, in particular for the original and MiM methods with larger speedup slopes than their SIMD counterparts, and with larger slopes for smaller number of cores than larger ones. For the original and MiM methods, the synchronization and communication overhead exceeds 50% of the total execution time with 256 or more cores, on all matrix sizes tested. For the original with SIMD and MiM with SIMD methods, the synchronization and communication overhead exceeds 50% of the total execution time with only 64 or more cores, for all matrix sizes tested, i.e., 4× less cores than the original and MiM methods, respectively.

The MiM and MiM with SIMD plots are very near the original and original with SIMD plots, respectively, with the MiM method resulting in the lowest or best percentage of communication and synchronization overhead, slightly below the original method.

9. Conclusion

We analyzed the performance of four parallel Gaussian Elimination algorithms: the Original method, the Original method with SIMD vectorization, the Meet in the Middle method, and the Meet in the Middle method with SIMD vectorization.

We developed a performance model for each of those methods and computed the speedups with respect to the serial original GE algorithm time, and speedups with respect to the same method on 1 core.

For large n and s (16 K), the speedups reach 185.9, 190.7, 723.8, and 748.6 for the original parallel method, the original method with SIMD, MiM, and MiM with SIMD, respectively, approximately 10x times lower than when no communication and synchronization times are accounted for in the performance model. SIMD vectorization makes a large difference for low number of cores (up to 1 K cores, with relatively smaller communication and synchronization overheads than larger systems) but the difference in performance between MiM and MiM with SIMD shrinks with increasing number of cores.

The efficiencies are good for large n's for up to 512 cores. The efficiencies of all four methods are low with 1 K cores or more stressing a major problem of highly parallel multi-core systems where the network-on-chip and memory latencies are still too high in relation to basic arithmetic operations. The efficiencies with 1 K core or more reveal that communication, synchronization, and memory latencies are major performance limiters in multi-core systems. The synchronization and communication overheads exceed 50% of the total execution time starting with 64 (256) or more cores for the SIMD methods (non-SIMD methods).

Thus the need exits to design multi-core systems with lower memory (and interconnect) operation latencies as both communication and synchronization steps involve memory operations. As the speedup does not increase much between 512 cores and 16 K cores, we question the reasoning behind exclusively integrating more cores in future submicron chips rather than integrating both processing cores and DRAM on the same chip. Although memory hiding techniques such as Geraci's (2008) in-Core LU solver and double buffering (Vianney et al., 2008) are effective in overlapping computation with memory operations, these techniques are not simple for the average parallel programmer, in addition to synchronization consuming longer and longer times. Improving memory and synchronization latencies are two key issues for future multi-core system designs as we approach the exascale computing age.

References

Bader, D. and Agarwal, V., 2007. FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine, HiPC 2007, LNCS 4873, Springer-Verlag, pp. 172–184.

Barrett, R. et al, 1994. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM.

Demmel, J. et al, 1993. Parallel numerical linear algebra. Acta Numer. 2, 111–197.

Demmel, J., Gilbert, J., Li, X., 1999. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. SIAM J. Matrix Anal. Appl. 20 (4), 915–952.

Duff, I., Van der Vorst, H., 1999. Developments and trends in the parallel solution of linear systems. Parallel Comput. 25, 1931–1970.

Fernández, D.M., Giannacopoulos, D., Gross, W.J., 2010. Multicore Acceleration of CG Algorithms Using Blocked-Pipeline-Matching Techniques. IEEE Trans. Magnet. 46 (8), 3057–3060.

Fog, A., 2012. Instruction tables -Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, http://www.agner.org/optimize/instruction_tables.pdf.

Geraci, J., 2008. A Comparison of Parallel Gaussian Elimination Solvers for the Computation of Electrochemical Battery Models on the Cell Processor, Ph.D. Dissertation, EECS Department, Massachussets Institute of Technology.

Grama, A. et al, 2003. Introduction to Parallel Computing, 2nd ed. Addison Wesley.

Greenbaum, A., 1997. Iterative Methods for Solving Linear Systems. SIAM.

Gupta, A. et al, 1997. Highly scalable parallel algorithms for sparse matrix factorization. IEEE Trans. Parallel Distrib. Syst. 8 (5), 502–520.

Gupta, A., et al., 1998. Design and implementation of a scalable parallel direct solver for sparse symmetric positive definite systems, Research Report UMSI 98/16, University of Minnesota Supercomputing Institute.

Heath, M., 1997. Parallel direct methods for sparse linear systems. In: Keyes, D.E., Sameh, A., Venkatakrishnan, V. (Eds.), Parallel Numerical Algorithms. Kluwer Academic Publishers., Boston, pp. 55–90.

IBM Corporation. Cell Broadband Engine technology. http://www.alphaworks.ibm.com/topics/cell.

Kumar, S., Gupta, K., 2012. Performance Analysis of Parallel Algorithms on Multi-core System Using OpenMP. Int. J. Comput. Sci. Eng. Inform. Technol. 2 (5), 55–64.

Laure, E., Al-Shandawely, M., 2011. Improving Gaussian Elimination on Multi-Core Systems, Fourth Swedish Workshop on Multicore Computing (MCC-2011), Linköping University.

Li, Z., Donde, V., Tournier, J.-C., Yang, F., 2011. On limitations of traditional multi-core and potential of many-core processing architectures for sparse linear solvers used in large-scale power system applications, IEEE Power and Energy Society General Meeting, pp. 1–8.

McGinn, S.F., Shaw, R.E., 2002. Parallel Gaussian elimination using OpenMP and MPI. High Performance, 2002. Proceedings. 16th Annual International Symposium on Computing Systems and Applications, pp. 169–173.

Pimple, M., Sathe, S., 2011. Architecture Aware Programming on Multi-core Systems. Int. J. Adv. Comput. Sci. Appl. 2 (6), 105–111.

Prinslow, G., 2011. Overview of Performance Measurement and Analytical Modeling Techniques for Multi-core Processors, http://www.cse.wustl.edu/ ~ jain/cse567-11/ftp/multcore. pdf.

Quinn, M., 1994. Parallel Computing. McGraw Hill.

Saad, Y., 2003. Iterative Methods for Sparse Linear Systems, 2nd ed. SIAM.

Sankar, A., 2004. Smoothed Analysis of Gaussian Elimination, Ph.D. Dissertation, Math Department, Massachussets Institute of Technology.

Van der Vorst, H., Chan, T., 1997. Linear system solvers: sparse iterative methods. In: Keyes, D.E., Sameh, A., Venkatakrishnan, V. (Eds.), Parallel Numerical Algorithms. Kluwer, pp. 91–118.

Vianney, D., et al., 2008. Performance analysis and visualization tools for Cell/B.E. multicore environment. ACM IFMT '08, Cairo, Egypt.

Wiggers, W., et al. 2007. Implementing the conjugate gradient algorithm on multi-core systems, IEEE International Symposium on System-on-Chip, pp. 1–4.

Xiaoye, S., Demmel, J., 1998. Making sparse Gaussian elimination scalable by static pivoting, IEEE/ACM Conference on Supercomputing.

Yeung, M., Chan, T., 1997. Probabilistic Analysis of Gaussian Elimination without Pivoting. SIAM J. Matrix Anal. Appl., 499–517.