A Novel Parallel Scan for Multicore Processors and Its Application in Sparse Matrix-Vector Multiplication

Nan Zhang

Abstract—We present a novel parallel algorithm for computing the scan operations on x86 multicore processors. The existing best known parallel scan for the same platform requires the number of processors to be a power of two. But this constraint is removed from our proposed method. In the design of the algorithm architectural considerations for x86 multicore processors are given so that the rate of cache misses is reduced and the cost of thread synchronization and management is minimized. Results from tests made on a machine with dual-socket $\times$ quad-core Intel Xeon E5405 showed that the proposed solution outperformed the best known parallel reference. A novel approach to sparse matrix-vector multiplication (SpMV) based on the proposed scan is then explained. The approach, unlike the existing ones that make use of backward segmented operations, uses forward ones for more efficient caching. An implementation of the proposed SpMV was tested against the SpMV in Intel's Math Kernel Library (MKL) and merits were found in the proposed approach.

Index Terms—Parallel algorithms, parallel scan, prefix sum, multicore computing, sparse matrix-vector multiplication.

1 Introduction

The scan operation takes a binary associative operator $\oplus$ and an array $X = (x_0, x_1, \ldots, x_{n-1})$ of n elements and returns the resultant array $X' = (x_0, x_0 \oplus x_1, \ldots, x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1})$. When addition $(+)$ is used as the operator, the scan is often referred to as prefix sum [2] because it computes the sums over all prefixes of the array. All partial sums [3] and plus-scan [4] are also names that were given to the operation. Together with other scans, such as or-scan and max-scan, they have been proposed as primitive operations [5] whose importance are well understood. Radix sort, quicksort, minimum spanning tree construction, and sparse matrix-vector multiplication (SpMV) are only a few of the many algorithms where the scan operations find their applications [2].

Seemingly serial though, parallel solutions to the scan problem have long been developed. The early parallel software algorithms for the scan [3], [5], [6] were designed for supercomputers, such as the Connection Machines CM-5 [7], [8] and the Cray Y-MP C90 [8]. Recently, with the emergence of general purpose graphics processing units (GPUs) some of these algorithms were adapted for computing on NVIDIA's GPUs [9], [10], [11].

In this paper, we present a novel parallel scan algorithm for x86 multicore processors that are dominating today's market. The reasons why we need such a new algorithm are twofold. First, those algorithms for supercomputers and for many-core GPUs often assume unlimited number of processors. But the same assumption cannot be made by a scan algorithm for multicore processors. On x86 multicore processors, the performance of an application is often determined by the efficiency of caching or/and memory bandwidth. The concerns for designing algorithms on multicore processors are very different from those when working with supercomputers. Second, the current best known parallel scan algorithm [2] for the same platform works only when the number of processors is a power of two, because it needs to compute a balanced binary tree over the input. But as the introduction of Intel's and AMD's hex-core processors, a scan algorithm without this constraint will be more desirable.

The parallel scan proposed in this paper works on any number of processors. Parallelization is achieved through concurrent running threads. Its implementation was tested on a machine with dual-socket $\times$ quad-core Intel Xeon E5405 (Harpertown) at 2.0 GHz, and the proposed method was found outperforming the current best known parallel reference. Based on this proposed scan, we then designed a novel approach to sparse matrix-vector multiplication. The existing scan-based solutions [12], [13] to SpMV use backward segmented operations, whose access patterns will cause inefficiencies for x86 caches. But the SpMV we designed use forward segmented sum and scan, which will result in a more efficient caching. The SpMV was implemented and compared with the general SpMV in Intel's Math Kernel Library (MKL) 10.1 for Linux, and merits were found in our method.

The main contributions of this work are summarized as follows:

- A novel parallel scan algorithm is designed for x86 multicore processors. Unlike the existing best known

• *The author is with the Department of Computer Science and Software Engineering, Xi'an Jiaotong-Liverpool University, Building 2, Office D445, No. 111 Ren'ai Road, Suzhou Industrial Park, Suzhou, Jiangsu 215123, P.R. China. E-mail: nan.zhang@xjtlu.edu.cn.*

Manuscript received 16 Dec. 2010; revised 28 Mar. 2011; accepted 25 Apr. 2011; published online 13 June 2011.
Recommended for acceptance by J. Weissman.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2010-12-0722.
Digital Object Identifier no. 10.1109/TPDS.2011.174.

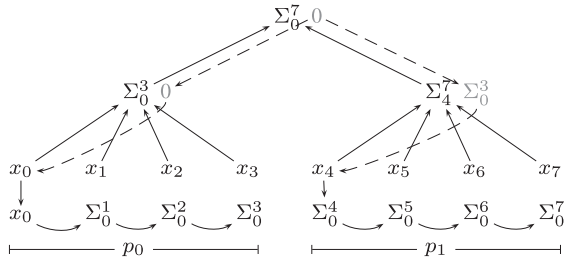


Fig. 1. Computing the prefix sum of eight values by two processors p_0 and p_1 using the algorithm discussed in [2]. This algorithm assumes p is fixed and $p < n$.

parallel scan for x86 multicore processors which requires power-of-two processors, the proposed algorithm does not have this limitation.

- The algorithm is optimized for x86 multicore processors, in which a general method for improving temporal locality in caching over multistage computations is invented and tested. The improvement is quantified.
- A novel scan-based SpMV method is proposed. The method depends on forward segmented sum and scan, rather than on the backward like the existing approaches. The x86 caches work more efficiently in forward accesses than in the backward.

Organization of the rest of this paper. The selected parallel reference is pointed out in Section 2. The proposed algorithm is discussed in Section 3. The novel scan-based SpMV is explained in Section 4. Conclusions and future work are summarized in Section 5.

2 The Parallel Reference

The balanced binary tree algorithm discussed by Blelloch in [2] (Fig. 1) under the condition that p (the number of processors) is fixed and $p < n$ (the number of elements) is chosen as the parallel reference to which the proposed parallel scan later is compared. See the supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.174, to this paper for why it has been chosen as the parallel reference and why it is the best known parallel scan for cache-based x86 multicore processors.

3 The Proposed Algorithm

In what follows, the proposed algorithm is presented, analyzed, and tested assuming addition being the binary operator.

3.1 Overview of the Algorithm

Assuming we have p processors, the algorithm first divides an array X of n values into p segments, denoted by $X_0, X_1, X_2, \ldots, X_{p-1}$. Each of the p processors is then assigned a distinct segment, after which, all the p processors scan in parallel the values in the segment that has been assigned to it. Once this phase of local scan is finished, the last value in segment X_0 is added onto the last value in X_1, which is then added onto the last value in X_2, until the last value in X_{p-2} is added onto the last value in X_{p-1}, the last segment. At this point, for the values except the last one in segment $X_i, i \in [1, p-1]$, we only need to add onto them the last value in segment X_{i-1} to obtain their partial sums. This addition operation is carried out in parallel by $p-1$ processors on all the segments $X_1, X_2, \ldots, X_{p-1}$. An example of the proposed algorithm is illustrated in Fig. 2, and the algorithm is presented in Algorithm 1.

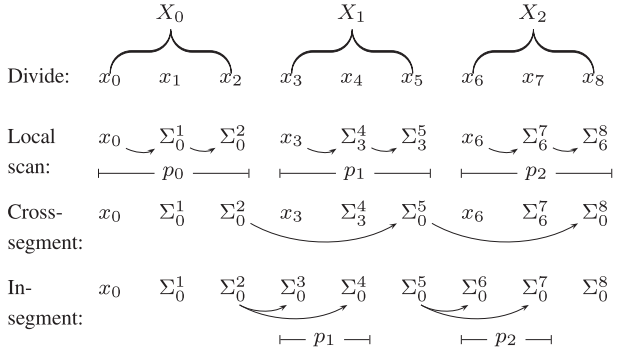


Fig. 2. Scanning nine values by three processors using the proposed algorithm.

Algorithm 1: The proposed parallel scan algorithm.

Input: An array $X = (x_0, x_1, \ldots, x_{n-1})$ of n values; number p of processors.
Output: Prefix sums $x_0, \Sigma_0^1, \Sigma_0^2, \ldots, \Sigma_0^{n-1}$ of all the values in X.

```
1  begin
2      Divide X into p segments, X_0, X_1, ..., X_{p-1}. Assume
       the number of values in X_i is m_i.
3      Assign X_i , i ∈ [0, p − 1] to a processor, e.g., p_i.
4      forall X_i = (x_{i,0}, x_{i,1}, ..., x_{i,m_i-1}), i ∈ [0, p − 1] in
       parallel do
5          for j ← 1 to m_i − 1 do
6              x_{i,j} = x_{i,j} + x_{i,j-1}
7      for i ← 0 to p − 2 do
8          Add the last number in segment X_i unto the last
           number in X_{i+1}.
9      forall X_i = (x_{i,0}, x_{i,1}, ..., x_{i,m_i-1}), i ∈ [1, p − 1] in
       parallel do
10         Add x_{i-1,m_{i-1}-1} unto x_{i,0}, x_{i,1}, ..., x_{i,m_i-2}.
               // x_{i-1,m_{i-1}-1} is the last value in
               segment X_{i-1}.
11 end
```

3.2 Improving Temporal Locality

In modern computer systems, because of the great speed disparity between CPU and memory it is often the performance of memory system that determines the runtime of an application, rather than the speed of processor. This is true for x86 multicore processors as well. So, in this algorithm we shall seek to optimize its memory accesses and make good use of cache.

This scan algorithm consists of multiple phases. To shorten its overall runtime, we may think about maximizing the number of valid cache lines which are fetched during the phase of the local scan and can be reused subsequently in the in-segment addition. Using segment $X_i, i \in [1, p-1]$ as an example, intuitively, we may think that over the local scan the data items in X_i are loaded into the last-level cache of the host machine and by the end of the phase some of them are still held in the cache while others may have

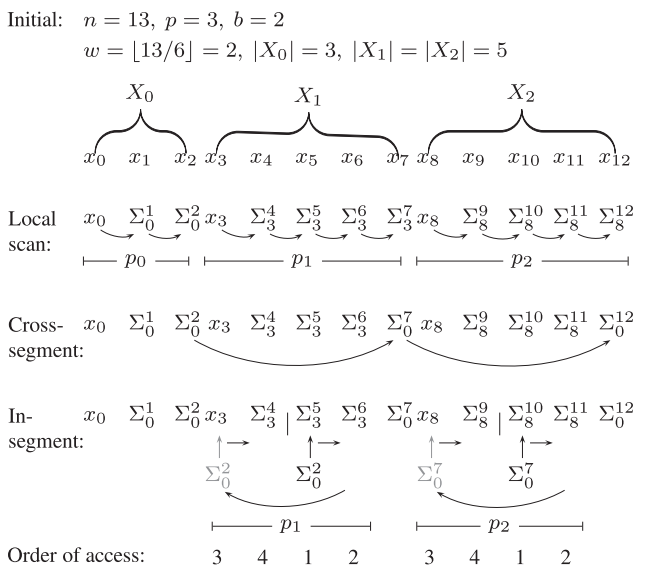


Fig. 3. Optimized access to enhance cache line reuse. The final result of the scan is not shown.

already been overwritten. We can assume that the data items near the end of segment X_i have a greater chance to be left in the last-level cache than the data items at the beginning. So, to reuse the cache lines that contain these data items in the final phase where the last value in segment X_{i-1} is added onto values in X_i we can start from values at the end of X_i and work backward to the values at the beginning. However, if the values in X_i are accessed consecutively backward from higher addresses to lower addresses, this pattern will make the working of the cache inefficient. So, we would further partition segment X_i into same-sized one-dimensional blocks, and the last value in X_{i-1} is added onto values within a block consecutively in sequential order, but this addition is applied to the last block first and all the way back to the first in reverse order.

Taking load balancing into consideration, for array X of n values, with p processors, if we use b to denote the size of a block, that is, the number of values in each block, the number w of blocks in each of the segments $X_1, X_2, \ldots, X_{p-1}$ is calculated as $w = \lfloor n/bp \rfloor$. Counting in the last value which is got right by the additions across the segments, the length of segment $X_i, i \in [1, p-1]$, that is, the number of values in X_i is set to $|X_i| = wb + 1$, and the length of the first segment X_0 is set to $|X_0| = n - (p-1)|X_i|$. Because the numbers in segment X_0 are not involved in the computation of the final phase, we can set its size slightly different to make all the other segments equal in length. So, for a segment $X_i, i \in [1, p-1]$, we denote its length X_i by m with $m = wb + 1$, and we denote the segment by $X_i = (X_{i,0}, X_{i,1}, X_{i,2}, \ldots, X_{i,b-1}, x_{i,m-1})$, where $X_{i,j}, j \in [0, b-1]$ is a block of numbers. So, in the final phase where the last value of segment X_{i-1} is added onto the numbers in X_i, the values in the last block $X_{i,b-1}$ are accessed in sequential order, and then the numbers in $X_{i,b-2}$, until the numbers in $X_{i,0}$ are dealt with. This process is carried out in parallel on all the segments $X_1, X_2, \ldots, X_{p-1}$. See Fig. 3 for an example with $n = 13, p = 3$, and $b = 2$.

To verify this blocked method can indeed achieve better locality, we made five tests on the testing machine (dual-socket × quad-core E5405) to compare the numbers of cache lines fetched into the L2 caches during the final phase with and without the method. We define the ratio R of cache lines reused during the final phase as

$$R = 1 - \frac{\text{number of cache lines actually fetched}}{\text{total of cache lines needed}}. \quad (1)$$

The number of cache lines fetched into the L2 caches can be obtained by summing up the count on the number of cache lines brought into the L2 caches because of demand requests originated from the L1 data caches, and the count on the number fetched by prefetch requests directed to the L2 caches [14]. The former is captured by the event L2_LINES_IN.SELF.DEMAND ($\mathcal{L}_I^D$) and the latter by L2_LINES_IN.SELF.PREFETCH ($\mathcal{L}_I^P$) over all the processing cores. So, the sum $\mathcal{L}_I^D + \mathcal{L}_I^P$ is the total number of cache lines fetched into the L2 caches. The counts can be collected by Intel VTune Performance Analyzer [15].

In each of the tests, an array of 4-byte single-precision floats was scanned using the proposed algorithm with the blocked method and without. The difference was most evident when the size of the data items accessed during the final phase exceeded the aggregate size of the L2 caches of the host machine. In all the tests, the block size was set to 4,096, that is, a block contained 4,096 floats. Using the test when the array size was 6,291,456 ($96n, n = 65{,}536$), as an example we show how the ratios were calculated. With the blocked method, when $n = 6{,}291{,}456$, as it was explained above, the number w will be $w = \lfloor n/bp \rfloor = 192$, and so $|X_1| = |X_2| = \cdots = |X_7| = wb + 1 = 786{,}433$. So, in the final phase the total floats that will be accessed is $(p-1)|X_1|$, which is $786{,}433 \times 7 = 5{,}505{,}031$. To hold 5,505,031 floats in the L2 caches with 64-byte cache lines totally $\lceil 5{,}505{,}031 \times 4/64 \rceil = 344{,}065$ lines will be needed. But in fact over the execution we collected the events $\mathcal{L}_I^D$ and $\mathcal{L}_I^P$ and found them to be 11,712 and 47,696, respectively. This meant that during the final phase actually the number of cache lines fetched was $11{,}712 + 47{,}696 = 59{,}408$, and so the ratio R of reused cache lines was $1 - 59{,}408/344{,}065 \approx 0.827$. On the other hand, with the unblocked method, the array was partitioned into eight equal-length segments, with $|X_0| = |X_1| = \cdots = |X_7| = n/p = 6{,}291{,}456/8 = 786{,}532$. This meant that over the final phase there was totally $(p-1)|X_1| = 786{,}532 \times 7 = 5{,}505{,}024$ floats to be accessed, and so needed 344,064 64-byte cache lines. But, in the test, over the execution the counts on events $\mathcal{L}_I^D$ and $\mathcal{L}_I^P$ collected by VTune were 44,288 and 93,874. So, the ratio R' in this test was $1 - 138{,}162/344{,}064 \approx 0.598$.

The ratios obtained from the five tests are plotted in Fig. 4, where it can be seen that the blocked method reduced the number of memory accesses and thus the runtimes could be shortened.

3.3 Synchronization

For a multithreaded approach to this scan problem, with p processors, we can create p threads in the phase of the local scan and destroy them afterward, and then after the cross-segment addition create another group of $p-1$ threads for the in-segment addition. But this will be inefficient because for an in-segment addition to start on a segment, it does not

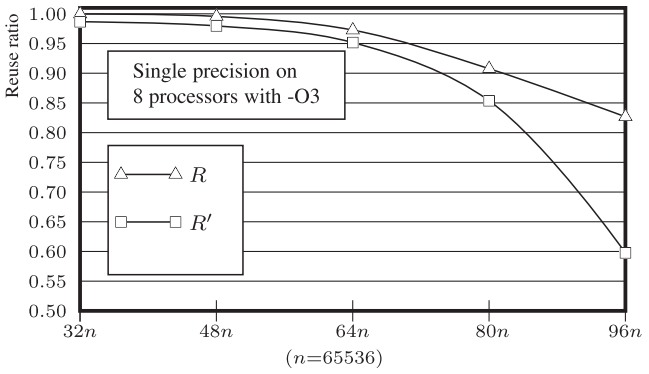


Fig. 4. Comparison in the ratios of cache lines reused. R denotes the ratios obtained with the blocked method, and R' the unblocked.

have to wait until all segments have been scanned. In designing the Synchronization strategy, we would

- give a thread as much work as we can before it has to be reaped,
- minimize the waiting time of each thread.

To scan an array of n values by p processors, we used p threads, and we bound each of the p threads onto a distinct processor. So, we use $p_i, i \in [0, p-1]$ to denote the thread that has been bound onto processor p_i. We used a group of p conditional variables for the phase of the local scan, and another group of p variables for the phase of the in-segment addition. We denote the first group by $P_1 = (P_1^0, P_1^1, \ldots, P_1^{p-1})$ and the second group by $P_2 = (P_2^0, P_2^1, \ldots, P_2^{p-1})$ (In fact P_1^0 and P_2^0 were not used.). These conditional variables were initialized as zeros. Associated with these conditional variables, we also used two groups of p binary mutexes and two groups of p signals. We assume thread $p_i, i \in [0, p-1]$ is assigned segment X_i.

The synchronization we designed works like this. Initially, all the p threads scan in parallel the segment that has been assigned to them. For thread p_0, once it has finished scanning segment X_0, it will repeatedly check if the scan on segment X_i, i from 1 to $p-1$, is finished ($P_1^i \stackrel{?}{=} 1$). If it is, thread p_0 will add the last value in X_{i-1} onto the last value in X_i. Then, it will set the conditional variable P_2^i to 1 and signal the change. For all the other threads, they will wait for the signals from thread p_0 to start the in-segment addition. For example, when $p_i, i \in [1, p-1]$ finishes scanning segment X_i, it will set the variable P_1^i to 1 and signal the change. It then will wait for P_2^i to become 1 so that it can continue with the in-segment addition. Thread p_0 is reaped

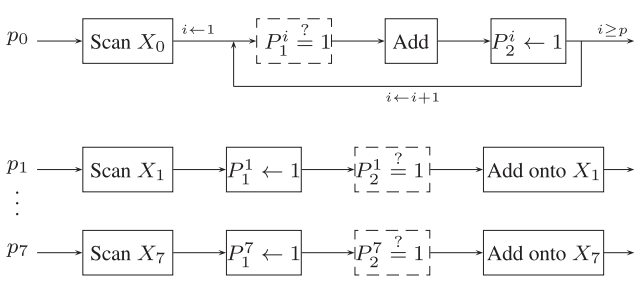


Fig. 5. The synchronization mechanism. A dashed boxed signifies the thread has to wait for the condition to become true to proceed.

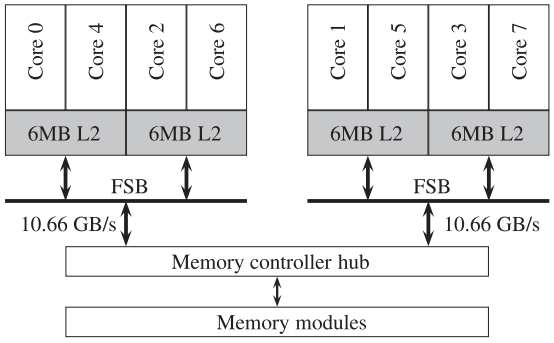


Fig. 6. The dual-socket × quad-core E5405. The numeric identities of the processing cores were assigned by operating system.

when the cross-segment addition is finished. Thread $p_i, i \in [1, p-1]$ is reaped when it finishes the in-segment addition. This mechanism is illustrated in Fig. 5 with $p = 8$.

3.4 Analysis

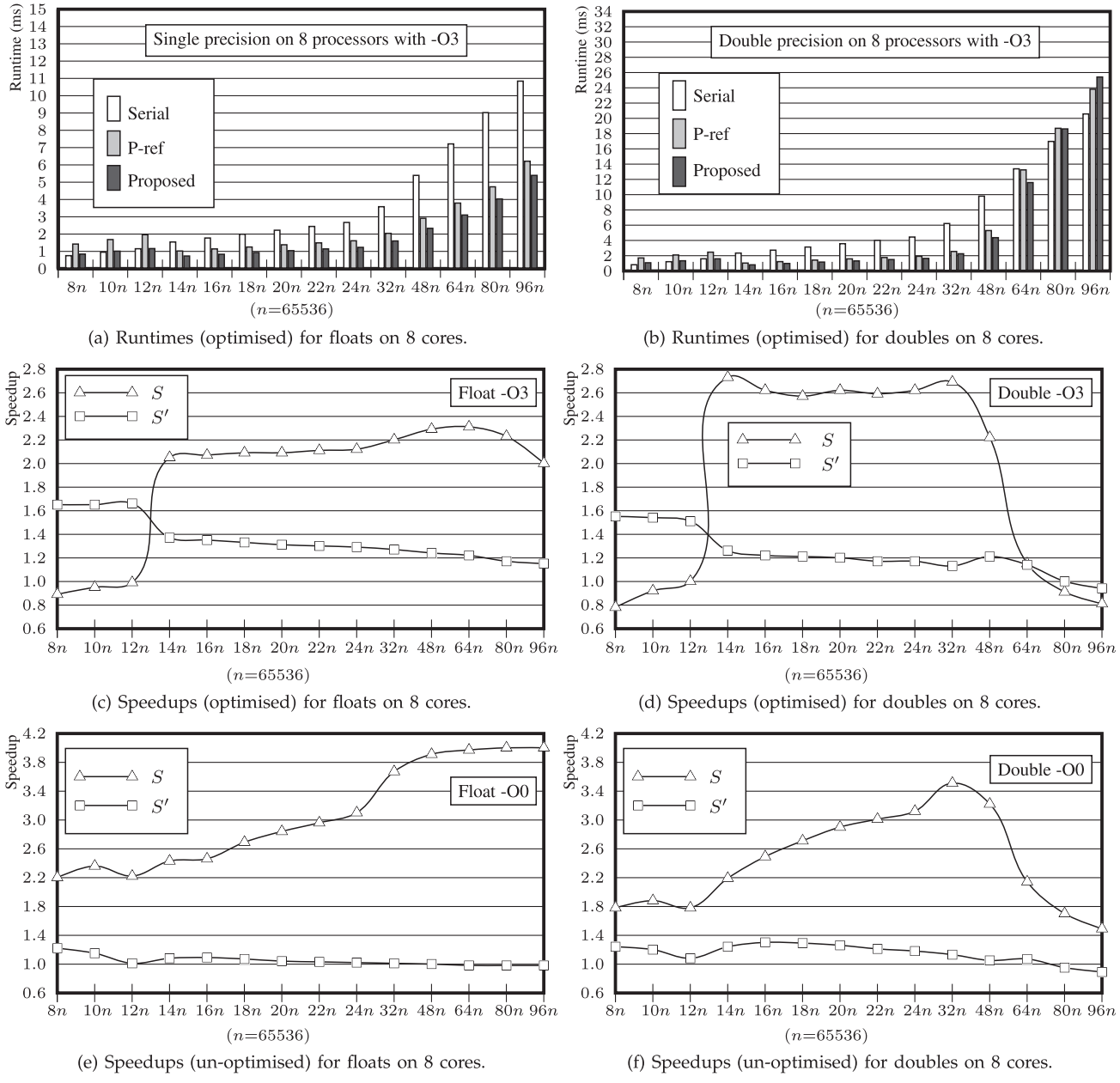
The proposed parallel scan is cost optimal. Its speedup S against the serial is $S = \frac{n-1}{2n/p} \approx p/2$. So, its theoretical performance should be similar (see the plots labeled by S' in Figs. 7e and 7f to the parallel reference algorithm. See the supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.174, for more analysis on this.

3.5 Implementation and Tests

The proposed algorithm was implemented in C/C++ via POSIX Threads, and was tested on a machine with dual-socket × quad-core Intel Xeon E5405 (Fig. 6). Instead of creating eight working threads while blocking the main thread, in the computation, seven threads were created and the main thread was assigned an equal portion of work. The main thread at its starting was bound onto processing core 0, and all the other seven threads were bound onto other distinct cores.

In the tests, the proposed algorithm was compared with the serial scan and the parallel reference, that is, the algorithm discussed by Blelloch [2] assuming fixed number of processors. All the source code were compiled by Intel C/C++ compiler icpc 11.0 for Linux. The testing machine was running Ubuntu Linux 8.10 64-bit version. The POSIX thread library used was native POSIX thread library (NPTL) 2.8.90. Various source code optimization techniques were applied to give the implementations optimal performance, e.g., replacing the division-by-two operations in the up-sweep and the down-sweep phase of the Blelloch's method by right shifting.

The tests were made on 4-byte floats and 8-byte doubles, under full compiler optimization (-ipo and -O3) and no compiler optimization, using different numbers of processing cores. In each group of the tests, the length of the number array varied from 8×2^{16} to 96×2^{16}. The block size b was set to 4,096. The runtimes were measured using the K-best scheme [16], where for each test maximal M results were collected, and if the K shortest results $t_0, t_1, \ldots, t_{K-1}$ satisfied the condition $(1+\epsilon)t_0 \geq t_{K-1}$ the process was stopped and the average of the K shortest times was used as the result. If, however, M results were collected and yet



(a) Runtimes (optimised) for floats on 8 cores.

(b) Runtimes (optimised) for doubles on 8 cores.

(c) Speedups (optimised) for floats on 8 cores.

(d) Speedups (optimised) for doubles on 8 cores.

(e) Speedups (un-optimised) for floats on 8 cores.

(f) Speedups (un-optimised) for doubles on 8 cores.

Fig. 7. Speedups and runtimes. S denotes the speedup of the proposed method against the serial, and S' denotes the speedup of the proposed against the parallel reference. P-ref stands for the parallel reference.

the ratio t_{K-1}/t_0 still not fell within the range $[1, 1+\epsilon]$, the process was aborted and the average of the K best results was used. In all the tests, M was set to 20, ϵ to 0.001 and K to 5. The runtimes and the derived speedups are presented in Fig. 7.

From the presented results, it can be seen that without compiler optimization the proposed algorithm performed very closely with the parallel reference (Figs. 7e, and 7f). Comparing to the serial scan, the proposed demonstrated four times speedups on the large test cases (Fig. 7e) when the numbers were single-precision floats. This result was inline with the theoretical analysis where $S = p/2 = 4$. However, when compiled with the optimizations (-O3 and -ipo), the proposed scan outperformed the parallel reference in most of the cases (Figs. 7c and 7d), but the speedups against the serial did not meet the theoretical prediction. To explain this, we have to understand that the compiler optimizations reduced the runtimes of all the three, so the parallel computations became memory bound rather than processor bound. (See more on this point in the supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.174.)

4 Application in SpMV

The sparse matrix-vector multiplication operation we considered is $Y \leftarrow Y + MX$, where M is a sparse matrix, and X, Y are dense vectors. From previous works [11], [12], we know that SpMV can be implemented via backward segmented scans. But neither of the works [11], [12] was done on x86 processors, on which a backward segmented

$$M = \begin{bmatrix} a_0 & 0 & 0 & a_1 & 0 \\ 0 & a_2 & 0 & 0 & 0 \\ 0 & 0 & a_3 & 0 & 0 \\ 0 & a_4 & 0 & a_5 & a_6 \\ 0 & 0 & 0 & 0 & a_7 \end{bmatrix} \quad \begin{aligned} A &= (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \\ E &= (0, 3, 1, 2, 1, 3, 4, 4) \\ T &= (0, 2, 3, 4, 7, 8) \end{aligned}$$

Fig. 8. A CSR storage format for matrix M.

scan (segmented suffix sum) will cause inefficiencies for caching. To overcome this drawback, we propose a SpMV on x86 multicore processors based on more efficient forward segmented operations.

We assume that a m (row) $\times$ n (column) sparse matrix M with w nonzero elements is stored in memory in the compressed sparse row (CSR) format, where a w-dimensional vector A stores all the w nonzero elements, a w-dimensional vector E stores the column indexes of each of the nonzero elements, and a $(m+1)$-dimensional array T keeps track of where each row starts in A. The last element of T is the number of nonzeros in M. This is the format adopted by the CSR in Intel's Math Kernel Library. An example of this CSR with zero-based index is found in Fig. 8.

For a $m \times n$ sparse matrix (w nonzeros) represented by CSR vectors A, E, T, and vectors $X = (x_0, x_1, \ldots, x_{m-1})$ and $Y = (y_0, y_1, \ldots, y_{m-1})$, to compute the SpMV, we first compute a w-dimensional array $V = (v_0, v_1, \ldots, v_{w-1})$ where each $v_i, i \in [0, w-1]$, is the product of a_i (the ith element in A) and x_{e_i} (the e_ith element in X, and e_i is the ith element in E). We then compute a w-dimensional flag array F. If the index of an element of F is found in T, the value of the element is set to 1, signifying a start of segment in V. All the other elements in F are set to zeros. Now, we derive a m-dimensional array D from the $(m+1)$-dimensional T, where $d_i, i \in [0, m-1]$, equals $t_{i+1} - 1$. This is where we differ from the previous approaches. This index array D will enable us to perform a forward segmented scan on vectors V and F, which is the work carried out in the next step. After the forward segmented scan, we do an elementwise sum on each $y_i, i \in [0, m-1]$, (the ith element in Y) and v_{d_i} (the d_ith element in V, and d_i is the ith element in D). An example which shows the steps is found in Fig. 9.

To perform the segmented scan on vectors $V = (v_0, v_1, \ldots, v_{w-1})$ and $F = (f_0, f_1, \ldots, f_{w-1})$ an approach of operator transformation [17], [18] is often adopted, where a new operator $+_s$ is defined. With the operator, the segmented scan is performed just like an ordinary scan. The new operator $+_s$ operates on two pairs (v_i, f_i) and $(v_j, f_j), i, j \in [0, w-1]$, assuming (v_i, f_i) is the source operand and (v_j, f_j) is the destination whose values are modified according to the following rules:

$$v_j = \begin{cases} v_i + v_j & f_j = 0 \\ v_j & f_j = 1 \end{cases}, \quad f_j = \begin{cases} f_i & f_j = 0 \\ f_j & f_j = 1. \end{cases} \tag{2}$$

This approach to SpMV, however, does more addition operations than is actually needed.[1] The segmented scan must pass over the vector V twice and perform a total $2(w-1)$ additions, whereas only $(w-1)$ additions are necessary, because only the accumulated sums in vector V whose positions are pointed by the values in the vector D are going to be collected, while the other sums in V are nowhere used

1. This was pointed out in [12].

CSR store

$$M = \begin{bmatrix} a_0 & 0 & 0 & a_1 & 0 \\ 0 & a_2 & 0 & 0 & 0 \\ 0 & 0 & a_3 & 0 & 0 \\ 0 & a_4 & 0 & a_5 & a_6 \\ 0 & 0 & 0 & 0 & a_7 \end{bmatrix} \quad \begin{aligned} A &= (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \\ E &= (0, 3, 1, 2, 1, 3, 4, 4) \\ T &= (0, 2, 3, 4, 7, 8) \end{aligned}$$

SpMV

$$Y = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} + \begin{bmatrix} a_0 & 0 & 0 & a_1 & 0 \\ 0 & a_2 & 0 & 0 & 0 \\ 0 & 0 & a_3 & 0 & 0 \\ 0 & a_4 & 0 & a_5 & a_6 \\ 0 & 0 & 0 & 0 & a_7 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

Compute V: $V = (a_0x_0, a_1x_3, a_2x_1, a_3x_2, a_4x_1, a_5x_3, a_6x_4, a_7x_4)$

Compute F: $F = (1, \quad 0, \quad 1, \quad 1, \quad 1, \quad 0, \quad 0, \quad 1)$

Compute D: $D = (\quad 1, \quad 2, \quad 3, \quad\quad\quad\quad 6, \quad 7)$

Segmented scan on V and F: $V = (a_0x_0, \underset{1}{a_0x_0 + a_1x_3}, \underset{2}{a_2x_1}, \underset{3}{a_3x_2}, a_4x_1, \underset{6}{a_4x_1 + a_5x_3 + a_6x_4}, \underset{7}{a_7x_4})$

Compute Y: $Y = (y_0 + a_0x_0 + a_1x_3, \;\; y_1 + a_2x_1, \;\; y_2 + a_3x_2, \;\; y_4 + a_4x_1 + a_5x_3 + a_6x_4, \;\; y_5 + a_7x_4)$

Fig. 9. The SpMV based on the forward segmented scan. The array D stores the positions from which the sums are going to be collected when the resultant vector Y is finally computed.

in the subsequent computations. In the example presented in Fig. 9, after the segmented scan the sum $a_4x_1 + a_5x_3$ in vector V is nowhere used, and therefore the addition operation and the memory write are redundant.

For this reason, the authors of [12] after discussing the segmented-scan-based SpMV continued to present another SpMV algorithm that was based on segmented sum and scan. The access patterns in their design, however, was backward as well. So, we applied our idea and converted all the backward accesses in their algorithm to forward ones. We then worked out a parallel implementation for the SpMV via segmented sum and scan which does $(w-1)$ additions. A few optimizations were attempted in the implementation, where the computation of the vector V was parallelized, and the computation of D was merged with the computation of Y at the final step. To reduce the amount of memory traffic, 16-bit unsigned shorts were used in the vector E for column indexes (This optimization also was adopted in [19].). Because of this, the dimensions of the matrices used in the tests were all less than $2^{16} \times 2^{16}$. Memory write operations were performed only if they were necessary. However, from the literatures [19], [20], we understand that the implementation was far from optimal.

The implementation was tested on the dual-socket $\times$ quad-core E5405 (Fig. 6), and was compared with the same general SpMV operation in the Intel MKL 10.1 for Linux with OpenMP enabled. We did not choose OSKI [21] as a reference, because it has not got a parallel implementation. The tests were made on eight square matrices (shown in Table 1) randomly selected from the University of Florida Sparse Matrix Collection [22]. Each nonzero element in the matrices was represented by an 8-byte double-precision float.

The results of the tests are reported in Fig. 10, which show that the SpMV based on our approach outperformed the one in MKL on certain matrices, especially the ones with large numbers of nonzero elements. And from this, we also found

TABLE 1
Overview of Sparse Matrices Used in Tests, Ordered in the Ascending Order of the Number of Nonzero Elements

Spyplot	Name	Dimension	Nonzeros	Kind
	pde2961	2,961	14,585	2D/3D problem
	FEM_3D	17,880	430,740	Thermal problem
	TSOPF_RS1	15,374	610,299	Power network problem
	sme3Da	12,504	874,887	Structural problem
	li	22,695	1,350,309	Electromagnetics problem
	av41092	41,092	1,683,902	2D/3D problem
	water_tank	60,740	2,035,281	Computational fluid dynamics
	TSOPF_RS2	28,338	2,943,887	Power network problem

FEM_3D is the short for FEM_3D_thermail, TSOPF_RS1 for TSOPF_RS_b162_c3 and TSOPF_RS2 for TSOPF_RS_b300_c2.

that the runtimes of the MKL SpMV with OpenMP were very unstable and unpredictable. On the contrary, the results from our SpMV were much more stable, and because the approach has complexity $O(w)$ (w is the number of nonzeros) we can expect that its runtime increases corresponding to the number of nonzeros in the matrix.

5 Conclusion

We have presented a novel parallel scan algorithm that is meant for platforms hosting x86 multicore processors. We have argued that because of the architectural differences the early scan algorithms for supercomputers and the recent for GPUs are not well suited for x86 multicore processors. The proposed algorithm gives much consideration in achieving load balance, enhancing locality, and minimizing the overhead of synchronization and thread management. The power-of-two constraint in the best known parallel reference is removed from the proposed, which enables it to work on Intel's and AMD's hex-core processors and on all future models that do not have power-of-two processing cores. The experiments made on the dual-socket × quad-core Intel E5405 showed that although the proposed outperformed the parallel reference, memory bandwidth was still the bottleneck.

To quantify an effectiveness of cacheline reuse over multiphase computations, the ratio of reused cachelines was defined. We discussed how this ratio can be calculated from the relevant performance event counts collected in runtime. To enhance this ratio so as to achieve better temporal locality in the parallel scan algorithm, a blocked

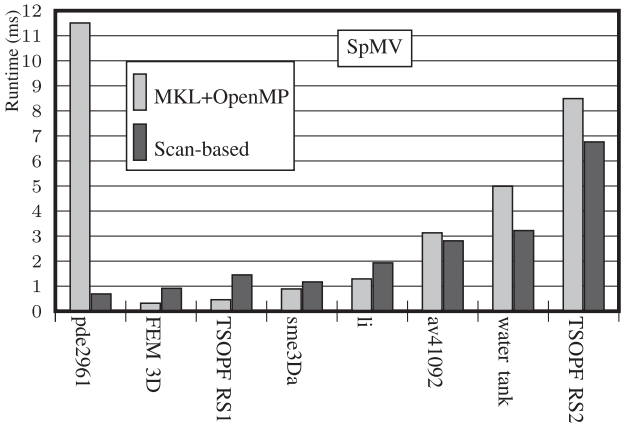


Fig. 10. SpMV performance on the testing machine.

method was developed and tested, which enabled more valid cachelines fetched by the parallel local scan be reused over the parallel in-segment addition. In general, computations consisting of multiple stages can be benefited from techniques of this kind.

The SpMV we proposed avoids the backward access patterns in the existing segmented-operation-based SpMV methods. Instead, it uses forward segmented operations which better suit x86 caches. Comparing to the SpMV in the Intel MKL which was parallelized through OpenMP, the runtime performance of our SpMV was more stable and predictable, and was faster on the tested matrices with large numbers of nonzero elements. As a future work, we can further optimize the proposed SpMV to reduce the amount of memory traffic by employing the blocked compressed sparse row (BCSR) format.

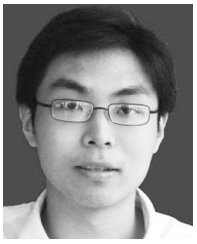
Acknowledgments

Preliminary conference version appeared as [1].

References

[1] N. Zhang, "A Novel Parallel Prefix Sum Algorithm and Its Implementation on Multi-Core Platforms," *Proc. Second Int'l Conf. Computer Eng. and Technology*, vol. 2, pp. 66-70, Apr. 2010.
[2] G.E. Blelloch, "Prefix Sums and Their Applications," Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon Univ., http://www.cs.cmu.edu/~guyb/papers/Ble93.pdf, Nov. 1990.
[3] W.D. Hillis and G.L. Steele Jr, "Data Parallel Algorithms," *Comm. ACM*, vol. 29, no. 12, pp. 1170-1183, Dec. 1986.
[4] K.E. Iverson, *A Programming Language*. John Wiley & Sons, Inc, Dec. 1962.
[5] G.E. Blelloch, "Scans as Primitive Parallel Operations," *IEEE Trans. Computers*, vol. 38, no. 11, pp. 1526-1538, Nov. 1989.
[6] G.E. Blelloch, "NESL: A Nested Data-Parallel Language (Version 2.6)," Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon Univ., 1993.
[7] W.D. Hillis, *The Connection Machine*. The MIT Press, 1985.
[8] G.E. Blelloch, J.C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee, "Implementation of a Portable Nested Data-Parallel Language," *J. Parallel and Distributed Computing*, vol. 21, no. 1, pp. 4-14, Apr. 1994.
[9] D. Horn, "Stream Reduction Operations for GPGPU Applications," *GPU Gems 2*, M. Pharr and R. Fernando, eds., ch. 36, pp. 573-589, Addison-Wesley Professional, 2005.
[10] S. Sengupta, A.E. Lefohn, and J.D. Owens, "A Work-Efficient Step-Efficient Prefix Sum Algorithm," *Proc. Workshop Edge Computing Using New Commodity Architectures*, pp. D-26-D-27, May 2006.

[11] M. Harris, S. Sengupta, and J.D. Owens, "Parallel Prefix Sum (Scan) with CUDA," *GPU Gems 3,* H. Nguyen, ed., ch. 39, Addison-Wesley, Aug. 2007.
[12] G.E. Blelloch, M.A. Heroux, and M. Zagha, "Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors," Technical Report CMU-CS-93-173, School of Computer Science, Carnegie Mellon Univ. and Cray Research, Inc., Aug. 1993.
[13] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens, "Scan Primitives for GPU Computing," *Proc. 22nd ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware,* pp. 97-106, 2007.
[14] *Intel 64 and IA-32 Architectures Optimization Reference Manual,* Intel Corporation, 248966-018, Mar. 2009.
[15] R.K. Malladi, "Using Intel VTune Performance Analyzer Events/Ratios and Optimizing Applications," http://software.intel.com, Jan. 2009.
[16] R.E. Bryant and D.R. O'Hallaron, *Computer Systems: A Programmer's Perspective,* ch. 9, p. 671. Prentice Hall, 2002.
[17] J.T. Schwartz, "Ultra-Computers," *ACM Trans. Programming Languages and Systems,* vol. 2, no. 4, pp. 484-521, Oct. 1980.
[18] S. Sengupta, M. Harris, and M. Garland, "Efficient Parallel Scan Algorithms for GPUs," Technical report, NVIDIA Corporation, Dec. 2008.
[19] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," *Parallel Computing,* vol. 35, no. 3, pp. 178-194, Mar. 2009.
[20] M. Krotkiewski and M. Dabrowski, "Parallel Symmetric Sparse Matrix-Vector Product on Scalar Multi-Core CPUs," *Parallel Computing,* vol. 36, no. 4, pp. 181-198, Apr. 2010.
[21] R. Vuduc1, J.W. Demmel, and K.A. Yelick, "OSKI: A Library of Automatically Tuned Sparse Matrix Kernels," *J. Physics: Conf. Series,* vol. 16, pp. 521-530, 2005.
[22] T.A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," Submitted to ACM Trans. Math. Software. http://www.cise.ufl.edu/~davis/techreports/matrices.pdf, 2010.



Nan Zhang received the BEng degree in computer science from the University of Shandong, China and the PhD and MSc degrees from the School of Computer Science, the University of Birmingham, United Kingdom. He is a lecturer in the Department of Computer Science and Software Engineering at Xian Jiaotong-Liverpool University. His research interests focus on high-performance parallel computing and its applications on financial derivative pricing.