Improving the Space-Time Efficiency of Processor-Oblivious Matrix Multiplication Algorithms

Yuan Tang
School of Computer Science, Fudan University
Shanghai, P. R. China
yuantang@fudan.edu.cn

***Abstract*—Classic cache-oblivious parallel matrix multiplication algorithms achieve optimality either in time or space, but not both, which promotes lots of research on the best possible balance or tradeoff of such algorithms. We study modern processor-oblivious runtime systems and figure out several ways to improve algorithm's time bound while still bounding space and cache requirements to be asymptotically optimal. By our study, we give out sublinear time, optimal work, space and cache algorithms for both general matrix multiplication on a semiring and Strassen-like fast algorithm. Our experiments also show such algorithms have empirical advantages over classic counterparts. Our study provides new insights and research angles on how to optimize cache-oblivious parallel algorithms from both theoretical and empirical perspectives.**

***Keywords*-space-time efficiency, cache-oblivious parallel algorithm, shared-memory multi-core or many-core architecture, matrix multiplication, modern processor-oblivious runtime**

I. Introduction

It's important to balance space-time requirements for an algorithm to achieve high performance on modern shared-memory multi-core and many-core systems. There are two big classes of parallel algorithms. One is processor-aware (PA), the other is processor-oblivious (PO).

Typical PA algorithms for Matrix Multiplication (MM) [1], [2], [3], [4], [5], [6], [7], [8] seem hard to achieve a perfect load balance on an arbitrary number, e.g. a prime number, of processors. The PA approach usually maps statically MM's computational DAG (Directed Acyclic Graph) of dimensions n-by-m-by-k , which stands for a multiplication of one n-by-k matrix with one k-by-m matrix, onto a 2D processor grid of dimensions $p^{1/2}$-by-$p^{1/2}$ [2], [7] , or a 3D grid of $p^{1/3}$-by-$p^{1/3}$-by-$p^{1/3}$ [1], or even a 2.5D grid of $(p/c)^{1/2}$-by-$(p/c)^{1/2}$-by-c [9], where p is processor count and $c \in \{1, 2, \ldots, p^{1/3}\}$ is a parameter depending on the availability of redundant storage. There are several concerns on classic PA algorithms.

1) There is no guarantee that $p^{1/2}$ or $p^{1/3}$ will be an integer number, which means that some degree of under-utilization of processors is unavoidable. In theory, p can even be a prime number. Hence, various tradeoffs need to be explored for a balance.
2) Obviously we need different shapes of processor grid when multiplying different shapes of MM for a balance of computation and communication (overall cache misses in the case of shared-memory architecture). For instances, multiplying a 1-by-n vector with an n-by-1 vector, an n-by-1 vector with a 1-by-n vector, or a general n-by-m matrix with an m-by-k matrix require different shapes of processor grid for an optimal balance in both computation and communication. However, the number of factorizations of p is usually much less than the number of possible shapes of MM, thus prior algorithms of 2D, 2.5D, or 3D are not flexible to adapt.

Fig. 1: Acronyms & Notations

MM	Matrix Multiplication
PO	Processor-Oblivious
PA	Processor-Aware
RWS	Randomized Work-Stealing
CAS	Compare-And-Swap
n	Problem dimension
p	Processor Count
ϵ_i	small constant
M	Cache size
B	cache line size
T_1	Work
T_∞	Time (span, depth, critical-path length)
T_p	Running time on p-processor system
T_1/T_∞	Parallelism
Q_1	Serial cache complexity
Q_p	Parallel cache complexity with p threads
$a \parallel b$	task b has *no* dependency on a
$a \; ; \; b$	task b has *full* dependency on a

The PO approach, on the contrary, just needs to specify the data and control dependency of computational DAG, then relies on a provably efficient runtime scheduler such as Cilk [10], [11] for a dynamic balance. Assuming a Randomized Work-Stealing scheduler [12] , an algorithm just needs to bound its critical-path length, also known as (a.k.a.) depth, span, or time bound [13], to be polylogarithmic, i.e. low-depth [14], its parallel running time is then $O(T_1/p + T_\infty)$ with high probability (w.h.p.) [15], where T_1 denotes total work and T_∞ denotes critical-path length. Similarly, if an algorithm's serial cache complexity is asymptotically optimal, its parallel cache complexity can be bounded by (1) [16], [17], where Q_1 denotes serial cache bound, Q_p denotes parallel cache bound, M is cache size and B is cache

Shanghai Natural Science Funding (No. 18ZR1403100)

Algo.	Work (T_1)	Time (T_∞)	Space (S_p)	Serial Cache (Q_1)
CO2	$O(n^3)$	$O(n)$	$O(n^2)$	$O(n^3/(B\sqrt{M}) + n^2/B)$
CO3	$O(n^3)$	$O(\log n)$	$O(n^3)$	$O(n^3/B)$
TAR-MM	$O(n^3)$	$O(n)$	$O(n^2 + pb^2)$	$O(n^3/(B\sqrt{M}) + n^2/B)$
SAR-MM	$O(n^3)$	$O(\log n)$	$O(p^{1/3}n^2)$	$O(n^3/(B\sqrt{M}) + n^2/B)$
STAR-MM	$O(n^3)$	$O(\sqrt{p}\log n)$	$O(n^2)$	$O(n^3/(B\sqrt{M}) + n^2/B)$
STRASSEN	$O(n^{\log_2 7})$	$O(\log n)$	$O(n^{\log_2 7})$	$O(n^{\log_2 7}/B)$
SAR-STRASSEN	$O(n^{\log_2 7})$	$O(\log n)$	$O(pn^2)$	$O(n^{\log_2 7}/(BM^{1/2\log_2 7-1}) + n^2/B)$
STAR-STRASSEN-1	$O(p^{0.09}n^{\log_2 7})$	$O(p^{1/2}\log n)$	$O(n^2)$	$O(p^{0.09}n^{\log_2 7}/(BM^{1/2\log_2 7-1}) + p^{1/2}n^2/B)$
STAR-STRASSEN-2	$O(n^{\log_2 7})$	$O(\log n)$	$O(p^{1/2\log_2 7}n^2)$	$O(n^{\log_2 7}/(BM^{1/2\log_2 7-1}) + p^{1/2\log_2 7-1}n^2/B)$

Fig. 2: Main results of this paper, with comparisons to typical prior works. CO2 stands for the MM (Matrix Multiplication) algorithm with $O(n^2)$ space (Fig. 3b); CO3 stands for the MM algorithm with $O(n^3)$ space (Fig. 3a); p denotes processor count, b is the base-case dimension.

line size of the ideal cache model [18].

$$Q_p = Q_1 + O(pT_\infty M/B) \tag{1}$$

From (1), we can see that for a PO algorithm to have efficient parallel cache bound, both its serial cache bound (Q_1) and critical-path length (T_∞) have to be optimal. However, if we look at two typical PO MM algorithms in Fig. 4, we can see that one algorithm (CO2 in Fig. 3b) has optimal serial cache bound but sub-optimal (linear) critical-path length and the other (CO3 in Fig. 3a) has optimal critical-path length but non-optimal serial cache and space bounds. To the best of our knowledge, no existing PO MM algorithm achieves both sublinear critical-path length and optimal serial cache bound.

Let's take a closer look at general MM $C = A \otimes B$ on a closed semiring $SR = (S, \oplus, \otimes, 0, 1)$, where S is a set of elements, $\oplus$ and $\otimes$ are binary operations on S, and 0, 1 are additive and multiplicative identities , respectively. For simplicity, we discuss only square MM. General MM can be computed recursively in a divide-and-conquer fashion as follows. At each level of recursion, the computation of an MM of dimension n (i.e. multiplication of two n-by-n matrices) is divided into four equally sized quadrants, which require updates from eight sub-MMs of dimension $n/2$ as shown in (2).

$$\begin{aligned}\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} &= \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \otimes \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} \\ &= \begin{bmatrix} A_{00} \otimes B_{00} & A_{00} \otimes B_{01} \\ A_{10} \otimes B_{00} & A_{10} \otimes B_{01} \end{bmatrix} \\ &\quad \oplus \begin{bmatrix} A_{01} \otimes B_{10} & A_{01} \otimes B_{11} \\ A_{11} \otimes B_{10} & A_{11} \otimes B_{11} \end{bmatrix} \end{aligned} \tag{2}$$

Depending on the availability of extra space, the computation of eight sub-MMs can be scheduled to run either completely in parallel as shown in Fig. 3a or in two parallel steps as Fig. 3b [19]. More sophisticated approaches are studied in the literature and will be discussed in Sect. VI.

We can calculate the critical-path length (time), space and serial cache bounds of these two algorithms by the recurrences of (3) – (12). CO2 algorithm in Fig. 3b uses no more space than input and output matrices, thus its space bound is simply $O(n^2)$. Equation (4) says that CO3 algorithm allocates an n-by-n temporary matrix D before spawning subtasks at each recursion level, thus has an additional overhead of matrix addition to merge results as indicated by (3). Equation (6) shows that CO2 algorithm does not have this overhead. Because of the temporary matrix, CO3 algorithm can run all eight (8) subtasks derived at each recursion level completely in parallel, hence only one subtask sits on the critical path as indicated in (3), while CO2 has to separate eight subtasks into two parallel steps, i.e. two subtasks sitting on its critical path as in (6). Equation (8) says that as soon as the input and output matrices of CO2 are smaller than some constant fraction of cache size M, there will be no more cache misses than a serial scan. Equation (10), on the contrary, indicates that CO3 algorithm keeps allocating new memory for new subtasks , which is always assumed to incur cold cache misses. Equations (5) and (11) show that the matrix addition is also done by a 2-way divide-and-conquer and there are four (4) subtasks derived at each recursion level. There is no data dependency among subtasks thus only one sitting on its critical path. Solving the recurrences, we can see that CO3 algorithm in Fig. 3a has an optimal $O(\log n)$ time bound (critical-path length) [1] if counting only data dependency but a poor $O(n^3)$ space and $O(n^3/B)$ serial cache bounds; By contrast, CO2 algorithm has an optimal $O(n^2)$ space and $O(n^3/(B\sqrt{M})+n^2/B)$ serial cache bound, but a sub-optimal $O(n)$ time bound (critical-path

[1]The overhead of matrix addition at each recursion level is just $O(1)$, i.e. $T_{\infty,\text{MADD}}(n) = O(1)$.

```
CO3(C, A, B)
1  // C ← A × B
2  if (sizeof(C) ≤ BASE_SIZE)
3      BASE-KERNEL(C, A, B)
4      return
5  D ← alloc(sizeof(C))
6  // Run all 8 sub-MMs concurrently
7  CO3(C00, A00, B00) ‖ CO3(C01, A00, B01)
8  ‖ CO3(C10, A10, B00) ‖ CO3(C11, A10, B01)
9  ‖ CO3(D00, A01, B10) ‖ CO3(D01, A01, B11)
10 ‖ CO3(D10, A11, B10) ‖ CO3(D11, A11, B11)
11 ; // sync
12 // Merge matrix D into C by addition
13 madd(C, D)
14 free (D)
15 return
```

(a) Recursive MM algorithm with $O(n^3)$ space

```
CO2(C, A, B)
1  // C ← A × B
2  if (sizeof(C) ≤ BASE_SIZE)
3      BASE-KERNEL(C, A, B)
4      return
5  // Run the first 4 sub-MMs concurrently
6  CO2(C00, A00, B00) ‖ CO2(C01, A00, B01)
7  ‖ CO2(C10, A10, B00) ‖ CO2(C11, A10, B01)
8  ; // sync
9  // Run the next 4 sub-MMs concurrently
10 CO2(C00, A01, B10) ‖ CO2(C01, A01, B11)
11 ‖ CO2(C10, A11, B10) ‖ CO2(C11, A11, B11)
12 ; // sync
13 return
```

(b) Recursive MM algorithm with $O(n^2)$ space

Fig. 3: Recursive MM algorithms. "‖" and ";" are linguistic constructs of Nested Dataflow model [20] (Fig. 1, Sect. II).

length). In the literature, it is known as space-time tradeoff.

$$T_{\infty,\text{CO3}}(n) = T_{\infty,\text{CO3}}(n/2) + T_{\infty,\text{MADD}}(n) \quad (3)$$
$$S_{\text{CO3}}(n) = 8S_{\text{CO3}}(n/2) + n^2 \quad (4)$$
$$T_{\infty,\text{MADD}}(n) = T_{\infty,\text{MADD}}(n/2) \quad (5)$$
$$T_{\infty,\text{CO2}}(n) = 2T_{\infty,\text{CO2}}(n/2) \quad (6)$$
$$Q_{1,\text{CO2}}(n) = 8Q_{1,\text{CO2}}(n/2) \quad (7)$$
$$Q_{1,\text{CO2}}(n) = O(n^2/B) \quad \text{if } n \leq \epsilon M \quad (8)$$
$$Q_{1,\text{CO3}}(n) = 8Q_{1,\text{CO3}}(n/2) + Q_{1,\text{MADD}}(n) \quad (9)$$
$$Q_{1,\text{CO3}}(1) = O(1) \quad (10)$$
$$Q_{1,\text{MADD}}(n) = 4Q_{1,\text{MADD}}(n/2) \quad (11)$$
$$Q_{1,\text{MADD}}(n) = O(n^2/B) \quad \text{if } n \leq \epsilon M \quad (12)$$

An interesting research question is if it is possible to achieve a sublinear time bound while still bounding space and cache complexities to be asymptotically optimal.

Our Contributions (Fig. 2):

- We look into runtime system of PO algorithms and prove that if a runtime follows the "busy-leaves" property [12], there will be no more than p subtasks of the same depth co-exist at any time in a p processor system. If a runtime memory allocator stands by the "Last-In First-Out" principle, memory blocks on the same processor are then largely reused, thus avoiding most of CO3 algorithm's cache misses without sacrificing critical-path length. Moreover, we propose a novel "lazy allocation" strategy such that a subtask allocates temporary space "*after*" it is spawned and "*after*" it makes sure that it is running simultaneously with its siblings that target the same output region. By the strategy, we reduce the space and cache requirement of PO MM algorithms to be asymptotically optimal while still keeping a sublinear critical-path length. In Sect. IV, we show how to extend the approach to Strassen-like fast algorithms.
- We show by experiments that our new PO MM algorithms do have performance advantage over both classic CO2 and CO3 algorithms in a fair comparison.

Organization: Sect. II introduces the cost models and programming model for algorithm design and analysis; Sect. III discusses the intuitions behind our new PO MM algorithms, as well as its step-by-step construction; Sect. IV extends our approach to Strassen-like fast MM algorithm; Sect. V experiment and compare our new algorithms with classic counterparts in a fair fashion; Sect. VI concludes the paper and discusses related works.

II. Cost Models and Programming Model

This section briefly introduces the theoretical models used in algorithm design and analysis.

Parallel Performance Model: We adopt the work-time model [13] (also known as work-span model [19]) to calculate time complexities. The model views a parallel computation as a DAG. Each vertex stands for a piece of computation that has no parallel construct and each edge represents some control or data dependency. For simplicity, we count each arithmetic operation such as multiplication, addition, and comparison uniformly as an $O(1)$ operation. The model calculates an algorithm's time bound (also known as critical-path length, span, depth, denoted by T_∞) by counting the number of arithmetic operations along critical path. Work bound (T_1) is then the sum of all arithmetic operations. Time bound T_∞ and work bound T_1 characterize the running time of parallel algorithm on infinite number and one processor(s), respectively. This paper assumes a Randomized Work-Stealing (RWS) scheduler that has the "busy-leaves" property as specified in [12]. More discussions on the property and its application can be found in Sect. III-B. By an RWS scheduler, a better time bound, or equivalently a shorter critical-path length, means more

work available for randomized stealing along critical path at runtime, hence a better "dynamic load-balance". We call a parallel algorithm ***work-efficient*** if its total work T_1 matches asymptotically the time bound of best serial algorithm of the same problem. Analogously, we have the notions of ***space-efficient*** and ***cache-efficient***.

Memory Model: We calculate only an algorithm's serial cache bound in the ideal cache model [18] since corresponding parallel cache bound under RWS scheduler is bounded by (1) [16], [17]. Therefore, in the rest of paper, the term "cache bound (complexity)" stands for "serial cache bound (complexity)" unless otherwise specified.

The ideal cache model has an upper level cache of size M and a lower level memory of infinite size. Data exchange between the upper and lower level is coordinated by an omniscient (offline optimal) cache replacement algorithm in cache line of size B. It also assumes a tall cache, i.e. $M = \Omega(B^2)$. To accommodate parallel execution, we further assume that the lower level memory follows CREW (Concurrent Read Exclusive Write) convention. Every concurrent reads from the same memory location can be accomplished in $O(1)$ time, while n concurrent writes to the same memory cell have to be serialized by some order and take $O(n)$ total time to complete. That is to say, no matter these n concurrent writes are coordinated by user's atomic operation such as Compare-And-Swap (CAS) or by system's synchronization facility such as "`cilk_sync`" in Cilk system, we always count their overall overhead by $O(n)$. By (1), $Q_p - Q_1 = O(pT_\infty M/B)$, we can see that the extra cache misses in a parallel execution to its serial execution is proportional to T_∞, the critical-path length. Hence, a shorter critical-path length means less parallel cache misses.

Programming Model: We use the notation "$a \parallel b$" to indicate that no subtasks of b depend on any subtasks of a, i.e. *no* dependency, while "a ; b" says that all subtasks of b depend on all subtasks of a, i.e. a *full* dependency.

III. Space-Time Adaptive and Reductive (STAR) MM Algorithm

Organization: Sect. III-A parallelizes all multiplications of CO2 algorithm in Fig. 3b without using much space; Based on CO3 algorithm in Fig. 3a, Sect. III-B reduces space requirement by exploiting the "busy-leaves" property [12], bounds cache complexity to be asymptotically optimal by requiring a "Last-In First-Out" memory allocator, and further improves space and cache requirements by "lazy allocation"; Sect. III-C bounds the space complexity to be asymptotically optimal by one level of indirection of TAR and SAR based on processor count p.

A. Time Adaptive and Reductive (TAR) MM Algorithm

A close look at CO2 algorithm in Fig. 3b reveals several aspects for further improvement.

1) It imposes more control dependency than necessary data dependency to keep the algorithm correct. That is, the all-to-all synchronization on line 8 of Fig. 3b serializes eight sub-MM's of each recursion levels to two parallel steps. For an instance, by this synchronization, the computation of CO2(C_{00}, A_{01}, B_{10}) not only waits on CO2(C_{00}, A_{00}, B_{00}) , which writes to the same C_{00} quadrant, but also has to wait for the computation working on completely disjoint quadrants, i.e. C_{01}, C_{10}, and C_{11}.
2) The synchronization on line 8 of Fig. 3b essentially serializes n multiplications updating the same cell of output matrix to n parallel steps. However, multiplications by themselves do not have any data dependency among each other and should be parallelized. The serialization makes sense only on later writing back by additions.

Based on the above observations, we devise a Time Adaptive and Reductive (TAR) algorithm to remove unnecessary control dependency from critical path. Figure 4a shows the pseudo-code. TAR-MM algorithm spawns all eight sub-MM's at each level of recursion simultaneously to maximize parallelism and serialize only concurrent writes to the same output region. Though we employ atomic operation such as Compare-And-Swap (ATOMIC-MADD on line 7) in our pseudo-code for the serialization, we feel that it's possible to design a dataflow operator like the "$\rightsquigarrow$" operator in Nested Dataflow model [20] or "`cilk_sync`" in Cilk system for the purpose. When output region C is of BASE_SIZE, the algorithm requests temporary storage from memory allocator (line 4) before base-case computation.

Memory Allocator: Memory allocator is a key component to guarantee the reuse of data blocks across requests on each processor. The reason that CO3 algorithm incurs $O(n^3/B)$ cache misses, which is asymptotically more than that of CO2, i.e. $O(n^3/(B\sqrt{M}))$, is because it repeatedly requests space before spawning subtasks at each depth, and people assume that allocation of space will always incur cold cache misses to fill it. If a memory allocator can guarantee the reuse of memory block, the above assumption is no longer true, thus we can save lots of un-necessary cold misses. That is to say, though CO3 algorithm still requests space at each depth, if the space is reused from prior requests, the fill-out of space by new data will not incur any cold misses because a smart cache can hold the reused memory block in cache by the omniscient cache replacement policy [18]. More precisely, all requests on the same processor should be served in a Last-In, First-Out (LIFO) fashion like a stack so that a smart cache can hold most recently used data blocks in cache to avoid thrashing. More precisely, if a user's program requests the same sized memory block on the same processor, allocator should guarantee to return exactly the same memory block for reuse.

We have an observation that each processor can work on only one task (one nested function call in an invocation tree) at a time. We further assume that a task computing a base case can not block or be preempted (according to the "busy-leaves" property [12]), which is true by Cilk's RWS scheduler [21]. Then we have Theorem 1.

Theorem 1: There is a TAR-MM algorithm that computes

general square MM of dimension n on a semiring in $O(n)$ time, $O(n^2+pb^2)$ space, and optimal $O(n^3/(B\sqrt{M})+n^2/B)$ cache misses, where b denotes the dimension of base case. If assuming b is some small constant, the space bound reduces to $O(n^2+p)$.

Proof: **Time bound:** The critical-path length of $O(n)$ follows from the fact that the algorithm parallelizes all multiplications and serializes only concurrent writes to the same memory location.

Space bound: Space bound follows from the facts that each processor can work on only one base case at a time and base-case computation can not block or be preempted. The temporary space for base-case computation on each processor thus are reused across different invocations.

Cache bound: The recurrences for cache bound are almost identical to that of CO2 except that the stop condition is changed to (14) as follows.

$$Q_{1,\text{TAR-MM}}(n) = 8Q_{1,\text{TAR-MM}}(n/2) \tag{13}$$

$$Q_{1,\text{TAR-MM}}(n) = O(n^2/B) \qquad \text{if } 3n^2+b^2 \le \epsilon M \tag{14}$$

Equation (13) recursively calculates a dimension-n (an n-by-n matrix multiplies another n-by-n matrix) TAR-MM's cache misses to eight (8) dimension-$(n/2)$ TAR-MM's cache misses. When the space requirements of input, output and temporary storage (a b-by-b storage allocated on line 4 of Fig. 4a) of a dimension-n TAR-MM are less than or equal some constant factor of M, any further recursion will not incur more cache misses than a serial scan as addressed by (14). Solving the recurrences will yield the bound. ■

B. Space Adaptive and Reductive (SAR) MM algorithm

A close look at CO3 algorithm (Fig. 3a) shows that it is designed for system with infinite or sufficient number of processors (proportional to algorithm's parallelism of T_1/T_∞). At each level of recursion, regardless of availability of idle processors, the algorithm always allocates a temporary matrix D of the same size as output matrix C (line 5 in Fig. 3a). By recursion, it allocates n^3-n^2 total temporary space on a p-processor system , where $p \ll T_1/T_\infty = O(n^3/\log n)$ usually holds in reality.

To reduce space requirement at no cost of parallelism, we have one observation and one algorithmic trick as follows.

Generalization of "busy-leaves" property: If we view the execution of a recursive algorithm as a depth-first traversal of its invocation tree, each node of which stands for a computing task (nested function call), we define the ***depth*** of a node to be the number of nodes on the path from root of tree to itself. The RWS scheduler specified in Blumofe and Leiserson's paper [21] has an important ***busy-leaves*** property as follows. The busy-leaves property says that from the time a task is spawned to the time it finishes, there is always at least one subtask from the subcomputation rooted at it that is ready. In other words, no leaf task can stall or be preempted. The busy-leaves property holds in Cilk runtime system [10], [11] and we further extend it by Theorem 2.

Theorem 2: If a runtime scheduler stands by the ***busy-leaves*** property, there can be no more than p tasks of the same depth executing or blocking at any time in a p-processor system.

Proof: We prove by induction on the depth of tasks. Since no leaf task can stall or be preempted, there can be no more than p leaf tasks at any time in a p-processor system. Since each leaf task can have only one parent task, it's obvious that their parent tasks can be no more than p either. Recursively, the argument holds for arbitrary depth d. ■

We verified by experiments that Theorem 2 does hold in Intel Cilk Plus runtime [10]. By Theorem 2, it is sufficient to allocate at most p copies of sub-matrix of any depth for reuse among all subtasks. More precisely, $\min\{p, 4^d\}$ copies of sub-matrices of any depth d are sufficient, where the term 4^d indicates that at each depth, at most four out of eight sub-MMs will require temporary space and another four will work right on its parent's space. So, for any depth d, the memory allocator needs to hold at most $\min\{p, 4^d\}$ blocks of size $n/2^d$-by-$n/2^d$ for reuse across requests.

Final SAR algorithm: The pseudo-code of this new algorithm, which we call SAR-MM, is shown in Fig. 4c, with a helper function in Fig. 4b.

Lazy Allocation: To minimize space requirement, or in other words maximize space reuse, we use an algorithmic trick that allocates temporary space in a lazy fashion. That is, a sub-MM will request for temporary space if and only if it runs simultaneously on a different processor from the sub-MM updating the same output region. In Fig. 4c, all top-half and bottom-half sub-MMs updating the same output regions are spawned simultaneously. In Fig. 4b, line 1 show that the top-half and corresponding bottom-half will compete on a mutex lock to determine who will reuse parent's space and who will request temporary space. If the top-half and bottom-half are executed one-after-another either on the same processor or on different processor, they will both reuse parent's space. If some sub-MM does request temporary space for local computation , it has to write its results back to parent atomically on line 13. Though we utilize atomic operations such as mutex lock to coordinate between every pairs of top-half and bottom-half, we feel that it is possible and will be beneficial to have a system's facility like a dataflow "$\rightsquigarrow$" operator [20] for the purpose. Besides the way shown in Fig. 4b, an alternative way to coordinate space reuse is to always let the bottom-half reuse parent's space and top-half check the status ("running" or "finished") of corresponding bottom-half when it is scheduled to run before it decides if it should request temporary space or just reuse bottom-half's space. We have to clarify that Theorem 2 always holds with or without the algorithmic trick of lazy allocation. The trick just further maximizes the space reuse, thus reduces cache misses.

Theorem 3: There is a SAR-MM algorithm that computes general MM of dimension n on a semiring in optimal $O(\log n)$ time, $O(p^{1/3}n^2)$ space, and optimal $O(n^3/B\sqrt{M}+n^2/B)$ cache bounds, assuming $p=o(n)$.

Proof: **Time bound:** Since the algorithm does not

impose any synchronization among the eight (8) sub-MMs derived at each depth, the time bound is not affected, i.e. there are still $O(\log n)$ (atomic) additions sitting on critical path. We have to clarify that the ATOMIC-MADD on line 13 in Fig. 4b is functionally equivalent to a summation of the syncrhonization on line 11 of CO3 algorithm in Fig. 3a and the madd on line 13. We just implement it by atomic operation such as Compare-And-Swap (CAS). Moreover, the trylock on line 1 and unlock on line 17 of Fig. 4b is just an $O(1)$ operation because every pair of top-half and bottom-half do not wait on each other by the operation, but just check and signal the status to each other.

Space bound: The recurrences of temporary space requirements are:

$$S(v) = 8S(v/2) + 4(v/2)^2 \quad \text{if } v > n/2^k \tag{15}$$
$$S(v) = pS_1(v) \quad \text{if } v \le n/2^k \tag{16}$$
$$S_1(v) = S_1(v/2) + (v/2)^2 \quad \text{if } v \le n/2^k \tag{17}$$
$$4 \times (8^0 + 8^1 + \ldots + 8^k) = p \tag{18}$$

The term S_1 stands for the space requirement on each processor. Equation (15) says that at upper levels of recursion, every four out of eight sub-MMs spawned at each level may require extra space. Equation (18) calculates the switching depth of k. The term $(n/2)^2$ on the right-hand side of (17) indicates that only one copy of $n/2$-by-$n/2$ temporary matrix is needed for all MMs of size n-by-n on any single processor. The recurrences solve to $k = (1/3)\log_2(7/8p + 1/2)$ and $S(n) = O(p^{1/3}n^2)$. Assuming $p^{1/3} \ll n$, i.e. $p = o(n)$, which usually holds in reality, the total space bound can be asymptotically less than that of the CO3 algorithm.

Cache bound: The recurrences of cache complexity are:

$$Q_1(n) = 8Q_1(n/2) + O(n^2/B) \tag{19}$$
$$Q_1(n) = O(n^2/B) \quad \text{if } (1 + 1/4 + \ldots)n^2 + 2n^2 \le \epsilon_4 M \tag{20}$$

Equation (19) is identical to that of the CO3 algorithm, where the $O(n^2/B)$ term accounts for the possible overheads of merging the top-half and corresponding bottom-half's results by addition. If the top-half reuses bottom-half's space by lazy allocation, the overhead will disappear. So (19) accounts for the worst case. The stop condition of (20) says that if the summation of memory footprint of all writes (output region) and reads (input region) of later recursions of dimension n are less than or equal some constant fraction of cache size M, it will incur no more cache misses than a serial scan. In the equation, the term $(1+1/4+\ldots)n^2$ stands for the summation of all subsequent writes' memory footprint assuming that same-sized memory requests reuse the same memory blocks on the same processor, and $2n^2$ is of all reads. The recurrences solve to the optimal $O(n^3/(B\sqrt{M}) + n^2/B)$ bound. ■

C. Space-Time Adaptive and Reductive (STAR) MM algorithm

The TAR algorithm removes multiplications from critical path without using much more space, while the SAR algorithm reduces space requirement without increasing time bound. A natural idea will be combining the two and yields a near time-optimal and space-optimal ***STAR*** MM algorithm.

The hybrid algorithm works as follows. At upper levels of recursion, we employ the TAR algorithm and switch to the SAR after depth k, where k is a parameter to be determined later. More precisely, before depth k, our new algorithm spawns all eight sub-MMs at each depth in parallel and serializes concurrent writes to the same output region like the TAR-MM algorithm; After depth k, the algorithm keeps spawning all eight sub-MMs at each depth in parallel and reuse memory blocks like the SAR-MM.

Theorem 4: There is a STAR-MM algorithm that computes the general MM of dimension n on a semiring in $O(\sqrt{p}\log n)$ time, optimal $O(n^2)$ space, and optimal $O(n^3/(B\sqrt{M}) + n^2/B)$ cache bounds, assuming $p = o(n^2/\log^2 n)$.

Proof: The time and space recurrences of the STAR MM algorithm are:

$$T_\infty(v) = 2T_\infty(v/2) \quad \text{if } v > n/2^k \tag{21}$$
$$T_\infty(v) = T_\infty(v/2) + O(1) \quad \text{if } v \le n/2^k \tag{22}$$
$$S(v) = pS_1(v) \quad \text{if } v \le n/2^k \tag{23}$$
$$S_1(v) = S_1(v/2) + (v/2)^2 \tag{24}$$

Equations (21) and (22) indicate that concurrent writes to the same output region are serialized / parallelized before / after depth k , respectively. Since there are at most two sub-MMs at each depth updating the same output quadrant, the serialization overhead is $O(1)$ at each level. Since no temporary space is requested before depth k, (23) counts only the space requirement after k, which has the same form as in the SAR algorithm. The recurrences solve to $S(v) = (1/3)p(n/2^k)^2$, and $T_\infty(n) = 2^k\log_2(n/2^k)$. Making $k = (1/2)\log_2 p$, we have $S(v) = (1/3)n^2 = O(n^2)$ and $T_\infty(n) = \sqrt{p}\log_2(n/\sqrt{p}) = O(\sqrt{p}\log n)$.

We can consider the cache bound as follows. Recall that from the paragraph of "Memory Model" in Sect. II we count only serial cache bound in this paper. If executing STAR algorithm on a single processor, i.e. $p = 1$, it reduces to SAR; If we adjust the switching depth k on the single processor, it will become some middle ground between the TAR and SAR. In either case, its cache bound stays optimal with a constant in big-Oh between that of the TAR and SAR. ■

From the proof of Theorem 4, we can see that the total extra space requirement of STAR algorithm is just a third of the output matrix size, i.e. $(1/3)n^2$, with an $O(\sqrt{p})$ factor increase in the time bound. Since the CO2 algorithm uses at least $3n^2$ space to hold the input and output matrices, this extra temporary space is just minimum. If we assume that $p = o(n^2/\log^2 n)$, the time bound stays sublinear.

D. Discussions

The main difference of our SAR-MM algorithm from CO3 is that SAR-MM requires a memory allocator to guarantee the reuse of the same sized memory blocks across different requests on the same processor, plus a novel "lazy allocation"

strategy. The reuse of memory blocks across requests removes un-necessary cold cache misses in CO3 algorithm. The generalization of "busy-leaves" property bounds total space. The "lazy allocation" trick further reduces space and cache requirements.

Though our STAR algorithm takes processor count p as a parameter to bound total space requirement, unlike classic processor-aware approach, we do not partition statically problem space to processor grid. Static partitioning strategy has several concerns as discussed in Sect. I. By contrast, STAR algorithm enjoys the full benefits of *dynamic load balance* as classic processor-oblivious approach, which is a key to a satisfactory speedup especially when there is no good static partitioning algorithm for a problem. By a sublinear time bound, i.e. a shorter critical-path length of computational DAG, our STAR algorithm has at least two ***advantages*** over classic cache-oblivious parallel counterpart (e.g. the CO2 algorithm in Fig. 3b). Firstly, it means more work available for dynamic load balance along critical path; Secondly, it incurs asymptotically less parallel cache misses according to (1).

IV. STAR ALGORITHM FOR STRASSEN-LIKE FAST MATRIX MULTIPLICATION ALGORITHM

This section extends the STAR technique to Strassen-like fast MM algorithms. Given square matrices A, B, and C, the Strassen algorithm [22] recursively divides each matrix into four equally sized quadrants as shown in (2). It then computes each quadrant of C as follows:

$$\begin{array}{lll} S_1 = A_{00} \oplus A_{11} & S_2 = A_{10} \oplus A_{11} & S_3 = A_{00} \\ S_4 = A_{11} & S_5 = A_{00} \oplus A_{01} & S_6 = A_{10} \ominus A_{00} \\ S_7 = A_{01} \ominus A_{11} & & \\ T_1 = B_{00} \oplus B_{11} & T_2 = B_{00} & T_3 = B_{01} \ominus B_{11} \\ T_4 = B_{10} \ominus B_{00} & T_5 = B_{11} & T_6 = B_{00} \oplus B_{01} \\ T_7 = B_{10} \oplus B_{11} & & \end{array}$$

$$P_r = S_r \otimes T_r, \qquad 1 \le r \le 7$$

$$\begin{array}{ll} C_{00} = P_1 \oplus P_4 \ominus P_5 \oplus P_7 & C_{01} = P_3 \oplus P_5 \\ C_{10} = P_2 \oplus P_4 & C_{11} = P_1 \oplus P_3 \ominus P_2 \oplus P_6 \end{array}$$

The $\ominus$ notation dnotes the inverse operation of $\oplus$.

Lemma 5: A straightforward parallelization of Strassen MM algorithm has an $O(\log n)$ time, $O(n^{\log_2 7})$ work, $O(n^{\log_2 7})$ space, and an $O(n^{\log_2 7}/B)$ serial cache bound.

Proof: The time (T_∞), space (S) and cache (Q_1) recurrences of a straightforward Strassen parallelization is as follows:

$$T_\infty(n) = T_\infty(n/2) + O(1) \qquad S(n) = 7S(n/2) + 17(n/2)^2$$
$$Q_1(n) = 7Q_1(n/2) + O(n^2/B)$$

The time recurrence says that at each depth of dimension n, the Strassen algorithm spawns simultaneously seven (7) subtasks of dimension $n/2$ and only one of them sits on the critical path. The $O(1)$ term in the time recurrence accounts for the overheads of constant number of matrix additions and subtractions (for computing S's, T's, and C's) at each depth. The space recurrence says that at each depth of dimension n, a straightforward parallelization requires at most 17 copies of $n/2$-by-$n/2$ temporary matrices to hold the intermediate results of all P_r's and some of the S_r's and T_r's. If an S_r or T_r corresponds directly to a quadrant of input matrices A or B with no $\oplus$ or $\ominus$ operations such as S_3 or T_2, the algorithm doesn't allocate temporary space for it. Since all seven subtasks of multiplication of the same recursion level execute simultaneously, all temporary matrices have to be ready before subtasks can be launched as the CO3 algorithm. The cache recurrence is analogous with a similar stop condition to that of the CO3 algorithm ((10)), i.e. $Q_1(1) = O(1)$, which indicates that the straightforward parallelization keeps allocating new space for new subtasks, which is always assumed to incur cold cache misses. The recurrences solve to $T_\infty(n) = O(\log n)$, $S(n) = O(n^{\log_2 7})$, and $Q_1(n) = O(n^{\log_2 7}/B)$. That is to say, the amount of space requirement, as well as cache misses, is proportional to total work. ■

A. The SAR algorithm for Strassen

Lemma 6: There is a SAR-STRASSEN algorithm that has optimal $O(\log n)$ time, $O(n^{\log_2 7})$ work, $O(pn^2)$ space, and optimal $O(n^2/B + n^{\log_2 7}/(BM^{1/2\log_2 7-1})$ cache bound.

Proof: Observing that on any processor three copies of temporary matrices of size $n/2^k$-by-$n/2^k$ is sufficient for all subtasks at the depth k, we have the following improved space recurrences with no change made to the time recurrences (of Lemma 5), which solve to $S(n) = pn^2$.

$$S(n) = pS_1(n) \qquad S_1(n) = S_1(n/2) + 3(n/2)^2$$

The three copies of temporary matrices are used to hold the input and output matrices of $P_r = S_r \otimes T_r$ at each depth, with S_r and T_r computed on the fly from A and B respectively. The final quadrants of C can be computed by reusing the space of C and P's. Analogous to the SAR-MM algorithm, the only change to cache recurrences of Lemma 5 is the stop condition.

$$Q_1(n) = 7Q_1(n/2) + O(n^2/B)$$
$$Q_1(n) = O(n^2/B) \quad \text{if } (1 + 1/4 + \ldots)3n^2 + 3n^2 \le \epsilon M$$

In the stop condition, the term $(1 + 1/4 + \ldots)3n^2$ accounts for the summation of temporary space reused for the same-sized memory requests on the same processor and $3n^2$ accounts for input and output quadrants of A, B, and C. The cache recurrences solve to the optimal $O(n^2/B + n^{\log_2 7}/(BM^{1/2\log_2 7-1})$. Again, this is because the SAR algorithm enforces the reuse of temporary storage of every depth on each processor. ■

B. The STAR algorithm for Strassen

Theorem 7: There is a STAR-STRASSEN algorithm that has an $O(p^{1/2}\log n)$ time, $O(p^{0.09}n^{\log_2 7})$ work, optimal $O(n^2)$ space, and $O(p^{0.09} \cdot n^{\log_2 7}/(BM^{1/2\log_2 7-1}) + p^{1/2} \cdot n^2/B)$ serial cache bound.

Proof: We construct the STAR-STRASSEN by employing the TAR-MM algorithm (Theorem 1) at upper levels of recursion and switching to the SAR-STRASSEN algorithm (Lemma 6) after depth k, where k is a parameter to be determined later.

The recurrences of the total work, time and space bounds of the hybrid algorithm become:

$$\begin{aligned}
T_\infty(v) &= 2T_\infty(v/2) + O(1) && \text{if } v > n/2^k \\
T_1(v) &= 8T_1(v) \\
S(v) &= S(v/2) \\
Q_1(v) &= 8Q_1(v/2) \\
T_\infty(v) &= O(\log_2 v) && \text{if } v \le n/2^k \\
T_1(v) &= O(v^{\log_2 7}) \\
S(v) &= O(pv^2) \\
Q_1(v) &= O(v^{\log_2 7}/(BM^{1/2\log_2 7-1}) + v^2/B)
\end{aligned}$$

Before the depth reaches k, it inherits the same recurrences of the TAR-MM algorithm and switches to those of SAR-STRASSEN after depth k. Making $k = (1/2)\log_2 p$, we have the total work bound of $O(p^{1/2(3-\log_2 7)}n^{\log_2 7} \approx O(p^{0.09}n^{\log_2 7})$, a factor of $O(p^{0.09})$ larger than the original Strassen, the time bound of $T_\infty(n) = O(p^{1/2}\log_2(n/\sqrt{p})) = O(p^{1/2}\log_2 n)$, a factor of $O(p^{1/2})$ longer, the space bound of $S(n) = O(n^2)$, a factor of $O(n^{\log_2 7-2}) \approx O(n^{0.8})$ improvement, and the serial cache bound of $O(p^{3/2-1/2\log_2 7}\cdot n^{\log_2 7}/(BM^{1/2\log_2 7-1}) + p^{1/2}\cdot n^2/B)$, where $3/2 - 1/2\log_2 7 \approx 0.09$. Notice that the constant hidden in the big-Oh of space bound is just 1. That is, besides the $3n^2$ space for the input and output matrices A, B, and C, the STAR-STRASSEN algorithm just requires a total of $1n^2$ extra temporary storage. ■

Though a factor of $O(p^{0.09})$ increase in work and cache bound seems small in theory, in practice the increase can matter. Since the expected running time of a processor-oblivious algorithm on a p-processor system under an RWS scheduler is $T_1/p + O(T_\infty)$ [21], the T_1/p term will dominate given sufficient parallelism as in the case of our STAR-STRASSEN whose $T_\infty = O(p^{1/2}\log_2 n)$.

Theorem 8: There is an alternative STAR-STRASSEN algorithm that has an optimal $O(n^{\log_2 7})$ work, optimal $O(\log n)$ time, near-optimal $O(n^{\log_2 7}/(BM^{1/2\log_2 7-1}) + p^{1/2\log_2 7-1}n^2/B)$ cache and an $O(p^{1/2\log_2 7}n^2)$ space bound.

Proof: An alternative way to construct a STAR-STRASSEN algorithm is to employ the straightforward parallel Strassen algorithm (Lemma 5) for the top k levels of recursion before switching to the SAR-STRASSEN algorithm (Lemma 6). Obviously the work and time bound will stay asymptotically optimal because both upper and lower levels are pure Strassen with no control or data dependency inserted on critical path. The recurrences for space and cache are as follows.

$$\begin{aligned}
S(v) &= 7S(v/2) + O(v^2) && \text{if } v \ge n/2^k \\
Q_1(v) &= 7Q_1(v/2) + O(v^2/B) \\
S(v) &= O(pv^2) && \text{if } v < n/2^k \\
Q_1(v) &= O(v^{\log_2 7}/(BM^{1/2\log_2 7-1}) + v^2/B)
\end{aligned}$$

Solving the recurrences, we have $S(n) = O(p^{1/2\log_2 7}n^2)$ and $Q_1(n) = O(n^{\log_2 7}/(BM^{1/2\log_2 7-1}) + p^{1/2\log_2 7-1}n^2/B)$. ■

The alternative STAR-STRASSEN in Theorem 8 has asymptotically optimal work, time, and near-optimal cache bound, so it may perform better in practice.

V. EXPERIMENTS

We have implemented the TAR, SAR, and STAR algorithms for general matrix multiplication of data type "double", and compared their performance with classic CO2 and CO3 algorithms on a 24-core shared-memory machine (Fig. 7).

In order for a *Fair* performance comparison, we require that all competing algorithms call the same kernal function for serial base-case computation in the same round of comparison. Moreover, we mandate the same base-case size for serial computation in all competing algorithms. Thus, the only difference among competing algorithms is how they partition and schedule tasks. We compute speedup by "$(\text{running\_time}_{\text{peer alg.}}/\text{running\_time}_{\text{STAR}} - 1) \times 100\%$".

Brief Summary: Our TAR algorithm performs consistently the fastest even with an $O(n^3)$ additional overheads of daxpy; Other new algorithms are generally faster than CO2 and CO3, especially when problem dimension is reasonably large. Interestly, with a relative faster kernel, i.e. MKL's dgemm and daxpy, CO2 is faster than CO3. While with a slower manually implemented kernel, CO3 becomes faster. More experimenting data can be presented in a full version.

VI. CONCLUSION AND RELATED WORKS

Concluding Remarks: In this paper, we reviewed and analyzed classic matrix multiplication algorithms, CO2 and CO3, in modern processor-oblivious runtime. The CO2 algorithm has provably best work, space, and serial cache bounds, while its longer critical-path length may incur more parallel cache missees in a parallel setting. On the contrary, people used to over-estimated CO3 algorithm's space and cache requirements. We show that by the busy-leaves property [12], we can derive Theorem 2, which is verified in popular Cilk runtime, such that there are no more than p subtasks of the same depth in a p-processor system. Moreover, by the Last-In First-Out memory allocation strategy, which seems to be true in modern heap and TLS (Thread-Local Storage) allocators, memory blocks are largely reused in the CO3 algorithm. By the above two properties, the classic CO3 algorithm does not perform too bad compared with CO2 algorithm in modern processor-oblivious runtime such as Intel Cilk Plus. Interestingly, when employed a manually implemented (slower) kernel for base-case computation, CO3 algorithm can even be faster than CO2 probably due to its shorter critical-path length. To further

reduce space requirement or in other words maximize space reuse, thus minimize un-necessary cold cache misses, we propose a "lazy allocation" strategy for our SAR and STAR algorithms. Though we utilize atomic operations to check subtask's status for lazy allocation, we believe that it's possible for runtime system to provide such facility, which can be our future work. We also show how to possibly extend our approach to Strassen-like fast algorithms.

Related Works:

Various optimizations, tradeoffs among work, space, time, communication bounds, on the general MM on a semiring or Strassen-like fast algorithm has been studied for decades, including at least [23], [24], [25], [9], [26], [27], [28], [29]. The basic idea behind these prior works is to switch manually back and forth between a serial algorithm to save and reuse space and a parallel algorithm to increase parallelism. Our approach differs in the following ways. Firstly, our approach is dynamic load-balance, which is arguably more flexible and adaptive on shared-memory system; Moreover, by generalizing the "busy-leaves" property of runtime schedulers, our technique upper-bounds space requirement to be asymptotically optimal without tuning; By a "Last-In First-Out" memory allocator and lazy allocation strategy, we bound cache efficiency to be asymptotically optimal without tuning; By having a sublinear critical-path length, we reduce asymptotically parallel cache misses.

Smith et al. [29] noticed divots in the performance curves of Intel MKL's dgemm when multiplying a matrix of size m-by-k with a matrix of size k-by-n (a rank-k update), where both m and n are fixed to 14400 and k is very slightly larger than a multiple of 240. We find the divots of Intel MKL's dgemm in square matrix multiplication where problem dimension is powers of two.

REFERENCES

[1] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, vol. 39, pp. 575–582, Sep. 1995.

[2] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, Bozeman, MT, USA, 1969, aAI7010025.

[3] A. Aggarwal, A. K. Chandra, and M. Snir, "Communication complexity of prams," *Theor. Comput. Sci.*, vol. 71, no. 1, pp. 3–28, Mar. 1990.

[4] E. Dekel, D. Nassimi, and S. Sahni, "Parallel matrix and graph algorithms," *SIAM Journal on Computing*, vol. 10, pp. 657–675, 1981.

[5] S. L. Johnsson, "Minimizing the communication time for matrix multiplication on multiprocessors," *Parallel Computing*, vol. 19, pp. 1235–1257, 1993.

[6] D. Irony, S. Toledo, and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication," *J. Parallel Distrib. Comput.*, vol. 64, no. 9, pp. 1017–1026, Sep. 2004.

[7] R. Chowdhury and V. Ramachandran, "Cache-efficient Dynamic Programming Algorithms for Multicores," in *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2008, pp. 207–216.

[8] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing communication in numerical linear algebra," *SIAM J. Matrix Analysis Applications*, vol. 32, no. 3, pp. 866–901, 2011.

[9] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 90–109.

[10] *Intel Cilk Plus Language Specification*, Intel Corporation, 2010, document Number: 324396-001US. Available from http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf.

[11] C. E. Leiserson, "The Cilk++ concurrency platform," *Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, March 2010.

[12] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, California, Jul. 1995, pp. 207–216.

[13] J. JáJá, *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[14] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri, "Low depth cache-oblivious algorithms," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 189–199.

[15] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *SPAA '98*, Jun. 1998, pp. 119–129.

[16] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proc. of the 12th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA 2000)*, 2000, pp. 1–12.

[17] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper, "Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures," in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009, pp. 91–100.

[18] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," *ACM Trans. Algorithms*, vol. 8, no. 1, pp. 4:1–4:22, Jan. 2012.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[20] D. Dinh, H. V. Simhadri, and Y. Tang, "Extending the nested parallel model to the nested dataflow model with provably efficient schedulers," in *SPAA'16*, Pacific Grove, CA, USA, 11 – 13 2016.

[21] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *JACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.

[22] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 14, no. 3, pp. 354–356, 1969.

[23] A. R. Benson and G. Ballard, "A framework for practical parallel fast matrix multiplication," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 42–53.

[24] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Communication-optimal parallel algorithm for strassen's matrix multiplication," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '12. New York, NY, USA: ACM, 2012, pp. 193–204.

[25] F. W. McColl and A. Tiskin, "Memory-efficient matrix multiplication in the bsp model," *Algorithmica*, vol. 24, no. 3, pp. 287–297, 1999.

[26] B. Kumar, C. Huang, P. Sadayappan, and R. W. Johnson, "A tensor product formulation of strassen's matrix multiplication algorithm with memory reduction," *Scientific Programming*, vol. 4, no. 4, pp. 275–289, 1995.

[27] B. Boyer, J.-G. Dumas, C. Pernet, and W. Zhou, "Memory efficient scheduling of strassen-winograd's matrix multiplication algorithm," in *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*, ser. ISSAC '09. New York, NY, USA: ACM, 2009, pp. 55–62.

[28] J. Huang, T. M. Smith, G. M. Henry, and R. A. van de Geijn, "Implementing strassen's algorithm with blis," *CoRR*, 2016.

[29] T. M. Smith, R. v. d. Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. V. Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1049–1059.

TAR-MM(C, A, B)

1 // $C \leftarrow A \times B$
2 **if** (sizeof(C) $\leq$ BASE_SIZE)
3 // Request space from the program-managed memory pool
4 $D \leftarrow$ GET-STORAGE(sizeof(C))
5 BASE-KERNEL(D, A, B)
6 // Write the intermediate results in D to C atomically
7 ATOMIC-MADD(C, D)
8 // Return storage to the memory pool
9 free(D)
10 **return**
11 // Run all 8 sub-MMs concurrently
12 TAR-MM(C_{00}, A_{00}, B_{00}) $\|$ TAR-MM(C_{01}, A_{00}, B_{01})
13 $\|$ TAR-MM(C_{10}, A_{10}, B_{00}) $\|$ TAR-MM(C_{11}, A_{10}, B_{01})
14 $\|$ TAR-MM(C_{00}, A_{01}, B_{10}) $\|$ TAR-MM(C_{01}, A_{01}, B_{11})
15 $\|$ TAR-MM(C_{10}, A_{11}, B_{10}) $\|$ TAR-MM(C_{11}, A_{11}, B_{11})
16 **return**

(a) TAR-MM algorithm

HLP($Parent$, A, B, d)

1 **if** ($parent$.trylock())
2 // work right on parent's storage
3 $D \leftarrow parent$
4 **else**
5 // request space for depth d
6 $D \leftarrow$ GET-STORAGE(sizeof($n/2^d$))
7 **if** (sizeof($n/2^d$) $\leq$ BASE_SIZE)
8 BASE-KERNEL(D, A, B)
9 **else**
10 SAR-MM(D, A, B, d)
11 **if** ($D \neq parent$)
12 // Update D to $parent$ atomically
13 ATOMIC-MADD($parent$, D)
14 // Return storage to the memory pool
15 free(D)
16 **else**
17 $parent$.unlock()
18 **return**

(b) The helper function request temporary storage from the program-managed memory pool *iff* parent's storage is occupied. If computation is on a local temporary storage, the helper function will write back results to parent by atomic addition.

SAR-MM(C, A, B, d)

1 // Computes SAR-MM at recursion level d
2 // Run all 8 sub-MMs concurrently
3 HLP($C_{00}, A_{00}, B_{00}, d+1$) $\|$ HLP($C_{01}, A_{00}, B_{01}, d+1$)
4 $\|$ HLP($C_{10}, A_{10}, B_{00}, d+1$) $\|$ HLP($C_{11}, A_{10}, B_{01}, d+1$)
5 $\|$ HLP($C_{00}, A_{01}, B_{10}, d+1$) $\|$ HLP($C_{01}, A_{01}, B_{11}, d+1$)
6 $\|$ HLP($C_{10}, A_{11}, B_{10}, d+1$) $\|$ HLP($C_{11}, A_{11}, B_{11}, d+1$)
7 **return**

(c) SAR-MM algorithm

Fig. 4: TAR-MM and SAR-MM algorithms.

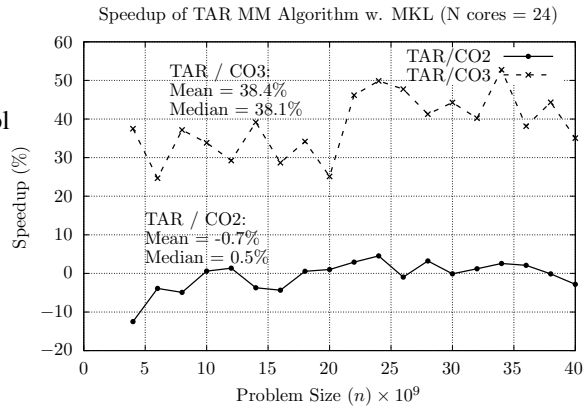

Fig. 5: TAR-MM's speedup over CO2 and CO3 with MKL kernel

with MKL kernel			
Mean/Median Spdp (%)	TAR	SAR	STAR
CO2	$-0.7/0.5$	$-2.0/-1.0$	$-2.6/-1.8$
CO3	38.4/38.1	36.6/36.5	35.8/34.0
with manual kernel			
Mean/Median Spdp (%)	TAR	SAR	STAR
CO2	11.8/9.2	9.3/6.8	10.5/8.0
CO3	1.6/1.5	$-0.6/-0.5$	0.5/0.5

Fig. 6: Mean and Median Speedup of TAR, SAR, and STAR algorithms over CO2 and CO3 with MKL kernel and with manually implemented kernel. All numbers shown in cell are in percentage. For an instance, In above rows of "with MKL kernel", the cell in intersection of TAR and CO2 reads "$-0.7/0.5$", which means with MKL kernel, the mean speedup of TAR algorithm over CO2 is 0.7% slower, while the median is 0.5% faster.

Fig. 7: Experiementing Machine

Name	24-core machine
OS	CentOS 7 x86_64
Compiler	ICC 19.0.3
CPU type	Intel Xeon E5-2670 v3
Clock Freq	2.30 GHz
# sockets	2
# cores / socket	12
Dual Precision	
FLOPs / cycle	16
Hyper-Threading	disabled
L1 dcache / core	32 KB
L2 cache / core	256 KB
L3 cache (shared)	30 MB
memory	132 GB