



WIKIPEDIA
The Free Encyclopedia

[Main page](#)
[Contents](#)
[Featured content](#)
[Current events](#)
[Random article](#)
[Donate to Wikipedia](#)
[Wikipedia store](#)

Interaction

[Help](#)
[About Wikipedia](#)
[Community portal](#)
[Recent changes](#)
[Contact page](#)


Tools

[What links here](#)
[Related changes](#)
[Upload file](#)
[Special pages](#)
[Permanent link](#)
[Page information](#)
[Cite this page](#)

Print/export

[Create a book](#)
[Download as PDF](#)
[Printable version](#)

Languages

 [Add links](#)

[Create account](#) [Log in](#)

Article [Talk](#)

[Read](#) [Edit](#) [View history](#)

Matrix multiplication algorithm

From Wikipedia, the free encyclopedia

Because [matrix multiplication](#) is such a central operation in many [numerical algorithms](#), much work has been invested in making **matrix multiplication algorithms** efficient. Applications of matrix multiplication in computational problems are found in many fields including [scientific computing](#) and [pattern recognition](#) and in seemingly unrelated problems such counting the paths through a [graph](#).^[1] Many different algorithms have been designed for multiplying matrices on different types of hardware, including [parallel](#) and [distributed](#) systems, where the computational work is spread over multiple processors (perhaps over a network).

Directly applying the mathematical definition of matrix multiplication gives an algorithm that [takes time](#) on the order of n^3 to multiply two $n \times n$ matrices ($\Theta(n^3)$ in [big O notation](#)). Better asymptotic bounds on the time required to multiply matrices have been known since the work of Strassen in the 1960s, but it is still unknown what the optimal time is (i.e., what the [complexity of the problem](#) is).

List of unsolved problems in computer science

What is the fastest algorithm for matrix multiplication?

Contents [\[hide\]](#)

- Iterative algorithm
 - Cache behavior
- Divide and conquer algorithm
 - Non-square matrices
 - Cache behavior
- Sub-cubic algorithms
- Parallel and distributed algorithms
 - Shared-memory parallelism
 - Communication-avoiding and distributed algorithms
 - Algorithms for meshes
- See also
- References
- Further reading

Iterative algorithm [\[edit\]](#)

The [definition of matrix multiplication](#) is that if $C = AB$ for an $n \times m$ matrix A and an $m \times p$ matrix B , then C is an $n \times p$ matrix with entries

$$c_{ij} = \sum_{k=1}^m a_{ik} b_{kj}.$$

From this, a simple algorithm can be constructed which loops over the indices i from 1 through n and j from 1 through p , computing the above using a nested loop:

- Input: matrices A and B
- Let C be a new matrix of the appropriate size
- For i from 1 to n :
 - For j from 1 to p :
 - Let sum = 0
 - For k from 1 to m :
 - Set sum \leftarrow sum + $A_{ik} \times B_{kj}$
 - Set $C_{ij} \leftarrow$ sum
- Return C

This algorithms takes [time](#) $\Theta(nmp)$ (in [asymptotic notation](#)),^[1] or $\Theta(n^3)$ in the case of square matrices, all of shape $n \times n$.^[2]

Cache behavior [\[edit\]](#)

The three loops in iterative matrix multiplication can be arbitrarily swapped with each other without an effect on correctness or asymptotic running time. However, the order can have a considerable impact on practical performance due to the [memory access patterns](#) and [cache](#) use of the algorithm;^[1] which order is best also depends on whether the matrices are stored in [row-major order](#), column-major order, or a mix of both.

In particular, in the idealized case of a [fully associative cache](#) consisting of M cache lines of b bytes each, the above algorithm is sub-optimal for A and B stored in row-major order. When $n > Mb$, every iteration of the inner loop (a simultaneous sweep through a row of A and a column of B) incurs a cache miss when accessing an element of B . This means the algorithm incurs $\Theta(n^3)$ cache misses in the worst case. As of 2010, the speed of memories compared to that of processors is such that the cache misses, rather than the actual calculations, dominate the running time for sizable matrices.^[3]

The optimal variant of the iterative algorithm for A and B in row-major layout is a [tiled](#) version, where the matrix is implicitly divided into square tiles of size \sqrt{M} by \sqrt{M} .^{[3][4]}

- Input: matrices A and B
- Let C be a new matrix of the appropriate size
- Pick a tile size $T = \Theta(\sqrt{M})$
- For I from 1 to n in steps of T :
 - For J from 1 to p in steps of T :
 - For K from 1 to m in steps of T :
 - Multiply $A_{I:I+T, K:K+T}$ and $B_{K:K+T, J:J+T}$ into $C_{I:I+T, J:J+T}$; that is:
 - For i from I to $\min(I + T, n)$:
 - For j from J to $\min(J + T, p)$:
 - Let $\text{sum} = 0$
 - For k from K to $\min(K + T, m)$:
 - Set $\text{sum} \leftarrow \text{sum} + A_{ik} \times B_{kj}$
 - Set $C_{ij} \leftarrow \text{sum}$
- Return C

In the idealized cache model, this algorithm incurs only $\Theta(\frac{n^3}{b\sqrt{M}})$ cache misses; the divisor $b\sqrt{M}$ amounts to several orders of magnitude on modern machines, so that the actual calculations dominate the running time, rather than the cache misses.^[3]

Divide and conquer algorithm [\[edit\]](#)

An alternative to the iterative algorithm is the [divide and conquer](#) algorithm for matrix multiplication. This relies on the [block partitioning](#)

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

which works for all square matrices whose dimensions are powers of two, i.e., the shapes are $2^n \times 2^n$ for some n . The matrix product is now

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

which consists of eight multiplications of pairs of submatrices, followed by an addition step. The divide and conquer algorithm computes the smaller multiplications [recursively](#), using the [scalar multiplication](#) $c_{11} = a_{11}b_{11}$ as its base case.

The complexity of this algorithm as a function of n is given by the recurrence^[2]

$$\begin{aligned} T(1) &= \Theta(1); \\ T(n) &= 8T(n/2) + \Theta(n^2), \end{aligned}$$

accounting for the eight recursive calls on matrices of size $n/2$ and $\Theta(n^2)$ to sum the four pairs of resulting matrices element-wise. Application of the [master theorem](#) shows this recursion to have the solution $\Theta(n^3)$, the same as the iterative algorithm.^[2]

Non-square matrices [\[edit\]](#)

A variant of this algorithm that works for matrices of arbitrary shapes and is faster in practice^[3] splits matrices in two instead of four submatrices, as follows.^[5] Let matrix shapes be $n \times m$ for A and $m \times p$ for B . Splitting a

matrix means dividing it into two parts of equal size, or as close to equal sizes as possible in the case of odd dimensions.

- Base case: if $\max(n, m, p)$ is below some threshold, use an **unrolled** version of the iterative algorithm.
- Recursive cases:

- If $\max(n, m, p) = n$, split A horizontally:

$$C = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

- Else, if $\max(n, m, p) = p$, split B vertically:

$$C = A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix}$$

- Otherwise, $\max(n, m, p) = m$. Split A vertically and B horizontally:

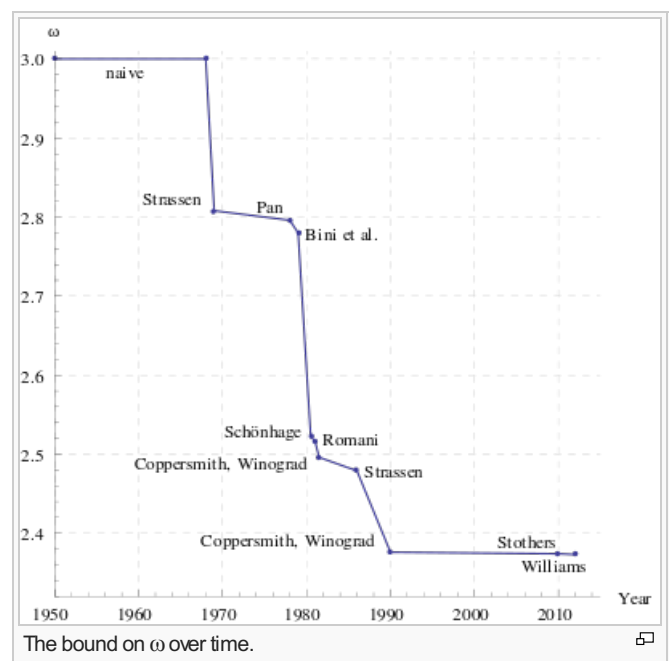
$$C = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2$$

Cache behavior [\[edit\]](#)

The cache miss rate of recursive matrix multiplication is the same as that of a **tiled** iterative version, but unlike that algorithm, the recursive algorithm is **cache-oblivious**.^[5] there is no tuning parameter required to get optimal cache performance, and it behaves well in a **multiprogramming** environment where cache sizes are effectively dynamic due to other processes taking up cache space.^[3] (The simple iterative algorithm is cache-oblivious as well, but much slower in practice if the matrix layout is not adapted to the algorithm.)

Sub-cubic algorithms [\[edit\]](#)

Algorithms exist that provide better running times than the straightforward ones. The first to be discovered was **Strassen's algorithm**, devised by **Volker Strassen** in 1969 and often referred to as "fast matrix multiplication". It is based on a way of multiplying two 2×2 -matrices which requires only 7 multiplications (instead of the usual 8), at the expense of several additional addition and subtraction operations. Applying this recursively gives an algorithm with a multiplicative cost of $O(n^{\log_2 7}) \approx O(n^{2.807})$. Strassen's algorithm is more complex, and the **numerical stability** is reduced compared to the naïve algorithm,^[6] but it is faster in cases where $n > 100$ or so^[1] and appears in several libraries, such as **BLAS**.^[7] It is very useful for large matrices over exact domains such as finite fields, where numerical stability is not an issue.



The current $O(n^k)$ algorithm with the lowest known exponent k is a generalization of the **Coppersmith–Winograd algorithm** that has an asymptotic complexity of $O(n^{2.3728639})$, by François Le Gall.^[8] The Le Gall algorithm, and the Coppersmith–Winograd algorithm on which it is based, are similar to Strassen's algorithm: a way is devised for multiplying two $k \times k$ -matrices with fewer than k^3 multiplications, and this technique is applied recursively. However, the constant coefficient hidden by the **Big O notation** is so large that these algorithms are only worthwhile for matrices that are too large to handle on present-day computers.^{[9][10]}

Since any algorithm for multiplying two $n \times n$ -matrices has to process all $2n^2$ -entries, there is an asymptotic lower bound of $\Omega(n^2)$ operations. Raz proves a lower bound of $\Omega(n^2 \log(n))$ for bounded coefficient arithmetic circuits over the real or complex numbers.^[11]

Cohn *et al.* put methods such as the Strassen and Coppersmith–Winograd algorithms in an entirely different **group-theoretic** context, by utilising triples of subsets of finite groups which satisfy a disjointness property called

the [triple product property \(TPP\)](#). They show that if families of [wreath products](#) of [Abelian groups](#) with symmetric groups realise families of subset triples with a simultaneous version of the TPP, then there are matrix multiplication algorithms with essentially quadratic complexity.^{[12][13]} Most researchers believe that this is indeed the case.^[10] However, Alon, Shpilka and [Chris Umans](#) have recently shown that some of these conjectures implying fast matrix multiplication are incompatible with another plausible conjecture, the [sunflower conjecture](#).^[14]

[Freivalds' algorithm](#) is a simple Monte Carlo algorithm that, given matrices A , B and C , verifies in $\Theta(n^2)$ time if $AB = C$.

Parallel and distributed algorithms [\[edit\]](#)

Shared-memory parallelism [\[edit\]](#)

The [divide and conquer algorithm](#) sketched earlier can be [parallelized](#) in two ways for [shared-memory multiprocessors](#). These are based on the fact that the eight recursive matrix multiplications in

$$\begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

can be performed independently of each other, as can the four summations (although the algorithm needs to "join" the multiplications before doing the summations). Exploiting the full parallelism of the problem, one obtains an algorithm that can be expressed in [fork–join style pseudocode](#).^[15]

Procedure multiply(C, A, B):

- Base case: if $n = 1$, set $c_{11} \leftarrow a_{11} \times b_{11}$ (or multiply a small block matrix).
- Otherwise, allocate space for a new matrix T of shape $n \times n$, then:
 - Partition A into $A_{11}, A_{12}, A_{21}, A_{22}$.
 - Partition B into $B_{11}, B_{12}, B_{21}, B_{22}$.
 - Partition C into $C_{11}, C_{12}, C_{21}, C_{22}$.
 - Partition T into $T_{11}, T_{12}, T_{21}, T_{22}$.
 - Parallel execution:
 - *Fork* multiply(C_{11}, A_{11}, B_{11}).
 - *Fork* multiply(C_{12}, A_{11}, B_{12}).
 - *Fork* multiply(C_{21}, A_{21}, B_{11}).
 - *Fork* multiply(C_{22}, A_{21}, B_{12}).
 - *Fork* multiply(T_{11}, A_{12}, B_{21}).
 - *Fork* multiply(T_{12}, A_{12}, B_{22}).
 - *Fork* multiply(T_{21}, A_{22}, B_{21}).
 - *Fork* multiply(T_{22}, A_{22}, B_{22}).
 - *Join* (wait for parallel forks to complete).
 - add(C, T).
 - Deallocate T .

Procedure add(C, T) adds T into C , element-wise:

- Base case: if $n = 1$, set $c_{11} \leftarrow c_{11} + t_{11}$ (or do a short loop, perhaps unrolled).
- Otherwise:
 - Partition C into $C_{11}, C_{12}, C_{21}, C_{22}$.
 - Partition T into $T_{11}, T_{12}, T_{21}, T_{22}$.
 - In parallel:
 - *Fork* add(C_{11}, T_{11}).
 - *Fork* add(C_{12}, T_{12}).
 - *Fork* add(C_{21}, T_{21}).
 - *Fork* add(C_{22}, T_{22}).
 - *Join*.

Here, *fork* is a keyword that signal a computation may be run in parallel with the rest of the function call, while *join* waits for all previously "forked" computations to complete. *partition* achieves its goal by pointer manipulation only.

This algorithm has a [critical path length](#) of $\Theta(\log^2 n)$ steps, meaning it takes that much time on an ideal machine with an infinite number of processors; therefore, it has a maximum possible [speedup](#) of $\Theta(n^3/\log^2 n)$ on any real computer. The algorithm isn't practical due to the communication cost inherent in moving data to and from the temporary matrix T , but a more practical variant achieves $\Theta(n^2)$ speedup, without using a

temporary.^[15]

Communication-avoiding and distributed algorithms ^[edit]

On modern architectures with hierarchical memory, the cost of loading and storing input matrix elements tends to dominate the cost of arithmetic. On a single machine this is the amount of data transferred between RAM and cache, while on a distributed memory multi-node machine it is the amount transferred between nodes; in either case it is called the *communication bandwidth*. The naïve algorithm using three nested loops uses $\Omega(n^3)$ communication bandwidth.

Cannon's algorithm, also known as the *2D algorithm*, partitions each input matrix into a block matrix whose elements are submatrices of size $\sqrt{M/3}$ by $\sqrt{M/3}$, where M is the size of fast memory.^[16] The naïve algorithm is then used over the block matrices, computing products of submatrices entirely in fast memory. This reduces communication bandwidth to $O(n^3/\sqrt{M})$, which is asymptotically optimal (for algorithms performing $\Omega(n^3)$ computation).^{[17][18]}

In a distributed setting with p processors arranged in a \sqrt{p} by \sqrt{p} 2D mesh, one submatrix of the result can be assigned to each processor, and the product can be computed with each processor transmitting $O(n^2/\sqrt{p})$ words, which is asymptotically optimal assuming that each node stores the minimum $O(n^2/p)$ elements.^[18] This can be improved by the *3D algorithm*, which arranges the processors in a 3D cube mesh, assigning every product of two input submatrices to a single processor. The result submatrices are then generated by performing a reduction over each row.^[19] This algorithm transmits $O(n^2/p^{2/3})$ words per processor, which is asymptotically optimal.^[18] However, this requires replicating each input matrix element $p^{1/3}$ times, and so requires a factor of $p^{1/3}$ more memory than is needed to store the inputs. This algorithm can be combined with Strassen to further reduce runtime.^[19] "2.5D" algorithms provide a continuous tradeoff between memory usage and communication bandwidth.^[20] On modern distributed computing environments such as **MapReduce**, specialized multiplication algorithms have been developed.^[21]

Algorithms for meshes ^[edit]

There are a variety of algorithms for multiplication on **meshes**. For multiplication of two $n \times n$ on a standard two-dimensional mesh using the 2D **Cannon's algorithm**, one can complete the multiplication in $3n-2$ steps although this is reduced to half this number for repeated computations.^[22] The standard array is inefficient because the data from the two matrices does not arrive simultaneously and it must be padded with zeroes.

The result is even faster on a two-layered cross-wired mesh, where only $2n-1$ steps are needed.^[23] The performance improves further for repeated computations leading to 100% efficiency.^[24] The cross-wired mesh array may be seen as a special case of a non-planar (i.e. multilayered) processing structure.^[25]

See also ^[edit]

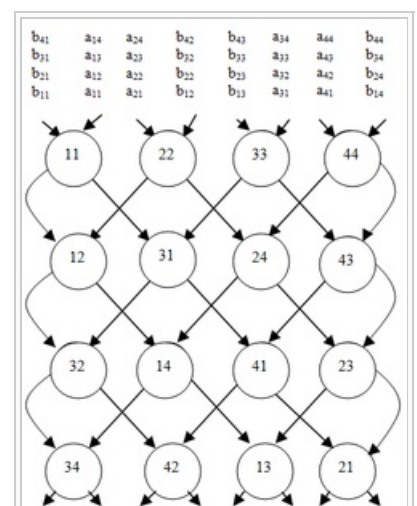
- Computational complexity of mathematical operations
- CYK algorithm, §Valiant's algorithm
- Matrix chain multiplication
- Sparse matrix-vector multiplication

References ^[edit]















- ↑ *a b c d* Skiena, Steven (2008). *The Algorithm Design Manual*. Springer. pp. 45–46, 401–403. doi:10.1007/978-1-84800-070-4_4 .
- ↑ *a b c* Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.
- ↑ *a b c d e* Amarasinghe, Saman; Leiserson, Charles (2010). "6.172 Performance Engineering of Software Systems, Lecture 8" . MIT OpenCourseWare. Massachusetts Institute of Technology. Retrieved 27 January 2015.

$\sqrt{\frac{M}{3}}$	$\{$	A_{11}	A_{12}	\cdots	A_{1s}
		A_{21}	A_{22}	\cdots	A_{2s}
		\vdots	\vdots	\ddots	\vdots
		A_{s1}	A_{s2}	\cdots	A_{ss}
		B_{11}	B_{12}	\cdots	B_{1s}
		B_{21}	B_{22}	\cdots	B_{2s}
		\vdots	\vdots	\ddots	\vdots
		B_{s1}	B_{s2}	\cdots	B_{ss}
		C_{11}	C_{12}	\cdots	C_{1s}
		C_{21}	C_{22}	\cdots	C_{2s}
		\vdots	\vdots	\ddots	\vdots
		C_{s1}	C_{s2}	\cdots	C_{ss}



Block matrix multiplication. In the 2D ^[26] algorithm, each processor is responsible for one submatrix of C . In the 3D algorithm, every pair of submatrices from A and B that is multiplied is assigned to one processor.



Matrix multiplication completed in $2n-1$ ^[27] steps for two $n \times n$ matrices on a cross-

4. ^{^ a b} Lam, Monica S.; Rothberg, Edward E.; Volf, Michael E. (1991). *The Cache Performance and Optimizations of Blocked Algorithms*. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS).
5. ^{^ a b} Prokop, Harald (1999). *Cache-Oblivious Algorithms*  (PDF) (Master's). MIT.
6. [^] Miller, Webb (1975), "Computational complexity and numerical stability", *SIAM News* **4**: 97–107, [CiteSeerX: 10.1.1.148.9947](#) 
7. [^] Press, William H.; Flannery, Brian P.; Teukolsky, Saul A.; Vetterling, William T. (2007). *Numerical Recipes: The Art of Scientific Computing* (3rd ed.). Cambridge University Press. p. 108. ISBN 978-0-521-88068-8.
8. [^] Le Gall, François (2014), "Powers of tensors and fast matrix multiplication", *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC 2014)*, [arXiv:1401.7714](#) . The original algorithm was presented by Don Coppersmith and Shmuel Winograd in 1990, has an asymptotic complexity of $O(n^{2.376})$. It was improved in 2013 to $O(n^{2.3729})$ by Virginia Vassilevska Williams, giving a time only slightly worse than Le Gall's improvement: Williams, Virginia Vassilevska. "Multiplying matrices faster than Coppersmith-Winograd"  (PDF).
9. [^] Iliopoulos, Costas S. (1989), "Worst-case complexity bounds on algorithms for computing the canonical structure of finite abelian groups and the Hermite and Smith normal forms of an integer matrix"  (PDF), *SIAM Journal on Computing* **18** (4): 658–669, doi:[10.1137/0218045](#) , MR [1004789](#) , "The Coppersmith–Winograd algorithm is not practical, due to the very large hidden constant in the upper bound on the number of multiplications required."
10. ^{^ a b} Robinson, Sara (2005). "Toward an Optimal Algorithm for Matrix Multiplication"  (PDF). *SIAM News* **38** (9).
11. [^] Ran Raz. On the complexity of matrix product. In Proceedings of the thirty-fourth annual ACM symposium on Theory of computing. ACM Press, 2002. doi:[10.1145/509907.509932](#) .
12. [^] Henry Cohn, Robert Kleinberg, Balázs Szegedy, and Chris Umans. Group-theoretic Algorithms for Matrix Multiplication. [arXiv:math.GR/0511460](#). *Proceedings of the 46th Annual Symposium on Foundations of Computer Science*, 23–25 October 2005, Pittsburgh, PA, IEEE Computer Society, pp. 379–388.
13. [^] Henry Cohn, Chris Umans. A Group-theoretic Approach to Fast Matrix Multiplication. [arXiv:math.GR/0307321](#). *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, 11–14 October 2003, Cambridge, MA, IEEE Computer Society, pp. 438–449.
14. [^] Alon, Shpilka, Umans, [On Sunflowers and Matrix Multiplication](#) 
15. ^{^ a b} Randall, Keith H. (1998). *Cilk: Efficient Multithreaded Computing*  (PDF) (Ph.D.). Massachusetts Institute of Technology. pp. 54–57.
16. [^] Lynn Elliot Cannon, [A cellular computer to implement the Kalman Filter Algorithm](#) , Technical report, Ph.D. Thesis, Montana State University, 14 July 1969.
17. [^] Hong, J. W.; Kung, H. T. (1981). "I/O complexity: The red-blue pebble game". *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*: 326–333.
18. ^{^ a b c} Irony, Dror; Toledo, Sivan; Tiskin, Alexander (September 2004). "Communication lower bounds for distributed-memory matrix multiplication". *J. Parallel Distrib. Comput.* **64** (9): 1017–1026. doi:[10.1016/j.jpdc.2004.03.021](#) .
19. ^{^ a b} Agarwal, R.C.; Balle, S. M.; Gustavson, F. G.; Joshi, M.; Palkar, P. (September 1995). "A three-dimensional approach to parallel matrix multiplication". *IBM J. Res. Dev.* **39** (5): 575–582. doi:[10.1147/rd.395.0575](#) .
20. [^] Solomonik, Edgar; Demmel, James (2011). "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms". *Proceedings of the 17th international conference on*

Parallel processing. Part II: 90–109.

21. [^] Bosagh Zadeh, Reza; Carlsson, Gunnar. "Dimension Independent Matrix Square Using MapReduce"  (PDF). Retrieved 12 July 2014.
22. [^] Bae, S.E., T.-W. Shinn, T. Takaoka (2014) A faster parallel algorithm for matrix multiplication on a mesh array. *Procedia Computer Science* 29: 2230-2240
23. [^] Kak, S. (1988) A two-layered mesh array for matrix multiplication. *Parallel Computing* 6: 383-385
24. [^] Kak, S. (2014) Efficiency of matrix multiplication on the cross-wired mesh array. <http://arxiv.org/abs/1411.3273> 
25. [^] Kak, S. (1988) Multilayered array computing. *Information Sciences* 45: 347-365

Further reading [\[edit\]](#)

- Buttari, Alfredo; Langou, Julien; Kurzak, Jakub; [Dongarra, Jack](#) (2009). "A class of parallel tiled linear algebra algorithms for multicore architectures". *Parallel Computing* **35**: 38–53. doi:10.1016/j.parco.2008.10.002 [↗](#). arXiv:0709.1272.
- Goto, Kazushige; Van de Geijn, Robert A. (2008). "Anatomy of high-performance matrix multiplication". *ACM Transactions on Mathematical Software* **34** (3). doi:10.1145/1356052.1356053 [↗](#). CiteSeerX: 10.1.1.140.3583 [↗](#).

v t e	Numerical linear algebra	[hide]
Key concepts	Floating point · Numerical stability	
Problems	Matrix multiplication (algorithms) · Matrix decompositions · Linear equations · Sparse problems	
Hardware	CPU cache · TLB · Cache-oblivious algorithm · SIMD · Multiprocessing	
Software	BLAS · Specialized libraries · General purpose software	

Categories: [Matrix multiplication algorithms](#)