11. Buluç A, Gilbert JR (2008) On the representation and multiplication of hypersparse matrices. 22nd IEEE International Symposium on Parallel and Distributed Processing, Miami, Florida
12. Buluç A, Gilbert JR (2008) Challenges and advances in parallel sparse matrix-matrix multiplication. International Conference on Parallel Processing, Portland

Asynchronous Iterations

►Asynchronous Iterative Algorithms

Asynchronous Iterative Algorithms

Giorgos Kollias, Ananth Y. Grama, Zhiyuan Li
Purdue University, West Lafayette, IN, USA

Synonyms

Asynchronous iterations; Asynchronous iterative computations

Definition

In iterative algorithms, a grid of data points are updated iteratively until some convergence criterion is met. The update of each data point depends on the latest updates of its neighboring points. Asynchronous iterative algorithms refer to a class of parallel iterative algorithms that are capable of relaxing strict data dependencies, hence not requiring the latest updates when they are not ready, while still ensuring convergence. Such relaxation may result in the use of inconsistent data which potentially may lead to an increased iteration count and hence increased computational operations. On the other hand, the time spent on waiting for the latest updates performed on remote processors may be reduced. Where waiting time dominates the computation, a parallel program based on an asynchronous algorithm may outperform its synchronous counterparts.

Discussion

Introduction

Scaling application performance to large number of processors must overcome challenges stemming from high communication and synchronization costs. While these costs have improved over the years, a corresponding increase in processor speeds has tempered the impact of these improvements. Indeed, the speed gap between the arithmetic and logical operations and the memory access and message passing operations has a significant impact even on parallel programs executing on a tightly coupled shared-memory multiprocessor implemented on a single semiconductor chip.

System techniques such as prefetching and multithreading use concurrency to hide communication and synchronization costs. Application programmers can complement such techniques by reducing the number of communication and synchronization operations when they implement parallel algorithms. By analyzing the dependencies among computational tasks, the programmer can determine a minimal set of communication and synchronization points that are sufficient for maintaining all control and data dependencies embedded in the algorithm. Furthermore, there often exist several different algorithms to solve the same computational problem. Some may perform more arithmetic operations than others but require less communication and synchronization. A good understanding of the available computing system in terms of the tradeoff between arithmetic operations versus the communication and synchronization cost will help the programmer select the most appropriate algorithm.

Going beyond the implementation techniques mentioned above requires the programmer to find ways to relax the communication and synchronization requirement in specific algorithms. For example, an iterative algorithm may perform a convergence test in order to determine whether to start a new iteration. To perform such a test often requires gathering data which are scattered across different processors, incurring the communication overhead. If the programmer is familiar with the algorithm's convergence behavior, such a convergence test may be skipped until a certain number of iterations have been executed. To further reduce the communication between different processors, algorithm designers have also attempted to find ways to relax the data dependencies implied in conventional parallel algorithms such that the frequency of communication can be substantially reduced. The concept of asynchronous iterative algorithms, which dates back over 3 decades, is developed as a result of such attempts.

With the emergence of parallel systems with tens of thousands of processors and the deep memory hierarchy accessible to each processor, asynchronous iterative algorithms have recently generated a new level of interest.

Motivating Applications

Among the most computation-intensive applications currently solved using large-scale parallel platforms are iterative linear and nonlinear solvers, eigenvalue solvers, and particle simulations. Typical time-dependent simulations based on iterative solvers involve multiple levels at which asynchrony can be exploited. At the lowest level, the kernel operation in these solvers are sparse matrix-vector products and vector operations. In a graph-theoretic sense, a sparse matrix-vector product can be thought of as edge-weighted accumulations at nodes in a graph, where nodes correspond to rows and columns and edges correspond to matrix entries. Indeed, this view of a matrix-vector product forms the basis for parallelization using graph partitioners. Repeated matrix-vector products, say, to compute $A^{(n)}y$, while solving $Ax = b$, require synchronization between accumulations. However, it can be shown that within some tolerance, relaxing strict synchronization still maintains convergence guarantees of many solvers. Note that, however, the actual iteration count may be larger. Similarly, vector operations for computing residuals and intermediate norms can also relax synchronization requirements. At the next higher level, if the problem is nonlinear in nature, a quasi-Newton scheme is generally used. As before, convergence can be guaranteed even when the Jacobian solves are not exact. Finally, in time-dependent explicit schemes, some time-skew can be tolerated, provided global invariants are maintained, in typical simulations.

Particle systems exhibit similar behavior to one mentioned above. Spatial hierarchies have been well explored in this context to derive fast methods such as the Fast Multipole Method and Barnes-Hut method. Temporal hierarchies have also been explored, albeit to a lesser extent. Temporal hierarchies represent one form of relaxed synchrony, since the entire system does not evolve in lock steps. Informally, in a synchronous system, the state of a particle is determined by the state of the system in the immediately preceding time step(s) in explicit schemes. Under relaxed models of synchrony, one can constrain the system state to be determined by the prior state of the system in a prescribed time window. So long as system invariants such as energy are maintained, this approach is valid for many applications. For example, in protein folding and molecular docking, the objective is to find minimum energy states. It is shown to be possible to converge to true minimum energy states under such relaxed models of synchrony.

While these are two representative examples from scientific domains, nonscientific algorithms lend themselves to relaxed synchrony just as well. For example, in information retrieval, PageRank algorithms can tolerate relaxed dependencies on iterative computations [35].

Iterative Algorithms

An iterative algorithm is typically organized as a series of steps essentially of the form

$$x(t+1) \leftarrow f(x(t)) \tag{1}$$

where operator $f(\cdot)$ is applied to some data $x(t)$ to produce new data $x(t+1)$. Here integer t counts the number of steps, assuming starting with $x(0)$, and captures the notion of time. Given certain properties on $f(\cdot)$, that it contracts (or pseudo-contracts) and it has a unique fixed point, and so on, an iterative algorithm is guaranteed to produce at least an approximation within a prescribed tolerance to the solution of the fixed-point equation $x = f(x)$, although the exact number of needed steps cannot be known in advance.

In the simplest computation scenario, data x reside in some Memory Entity (ME) and operator $f(\cdot)$ usually consists of both operations to be performed by some Processing Entity (PE) and parameters (i.e., data) hosted by some ME. For example, if the iteration step is a left matrix-vector multiplication $x \leftarrow Ax$, x is considered to be the "data" and the matrix itself with the set of incurred element-by-element multiplications and subsequent additions to be the "operator", where the "operator" part also contains the matrix elements as its parameters.

However, in practice this picture can get complicated:

- Ideally x data and $f(\cdot)$ parameters should be readily available to the $f(\cdot)$ operations. This means that one

would like all data to fit in the register file of a typical PE, or even its cache files (data locality). This is not possible except for very small problems which, most of the time, are not of interest to researchers. Data typically reside either in the main memory or for larger problems in a slow secondary memory. Designing data access strategies which feed the PE at the fastest possible rate, in effect optimizing data flow across the memory hierarchy while preserving the semantics of the algorithm, is important in this aspect. Modern multicore architectures, in which many PEs share some levels of the memory hierarchy and compete for the interconnection paths, introduce new dimensions of complexity in the effective mapping of an iterative algorithm.

- There are cases in which data x are assigned to more than one PEs and $f(\cdot)$ is decomposed. This can be the result either of the sheer volume of the data or the parameters of $f(\cdot)$ or even the computational complexity of $f(\cdot)$ application which performs a large number of operations in each iteration on each data unit. In some rarer cases the data itself or the operator remains distributed by nature throughout the computation. Here, the PEs can be the cores of a single processor, the nodes (each consisting of multiple cores and processors) in a shared-memory machine or similar nodes in networked machines. The latter can be viewed as PEs accessing a network of memory hierarchies of multiple machines, i.e., an extra level of ME interconnection paths.

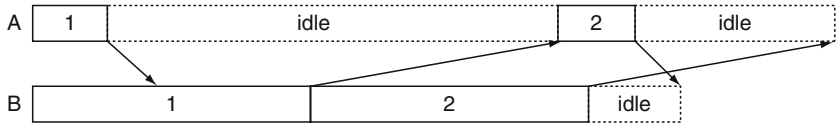
Synchronous Iterations and Their Problems

The decomposition of x and $f(\cdot)$, as mentioned above, necessitates the synchronization of all PEs involved at each step. Typically, each fragment $f_i(\cdot)$ of the decomposed operator may need many, if not all, of the fragments $\{x_j\}$ of the newly computed data x for an iteration step. So, to preserve the exact semantics, it should wait for these data fragments to become available, in effect causing its PE to synchronize with all those PEs executing the operator fragments involved in updating these $\{x_j\}$ in need. Waiting for the availability of data at the end of each iteration make these iterations "synchronous" and this practice preserves the semantics. In networked environments, such synchronous iterations can be coupled with either "synchronous communications" (i.e., synchronous global communication at the end of each step) or "asynchronous communications" for overlapping computation and communication (i.e., asynchronously sending new data fragments as soon as they are locally produced and blocking the receipt of new data fragments at the end of each step) [5].

Synchronization between PEs is crucial to enforcing correctness of the iterative algorithm implementation but it also introduces idle synchronization phases between successive steps. For a moment consider the extreme case of a parallel iterative computation where for some PE the following two conditions happen to hold:

1. It completes its iteration step much faster than the other PEs.
2. Its input communication links from the other PEs are very slow.

It is evident that this PE will suffer a very long idle synchronization phase, i.e., time spent doing nothing (c.f. Fig. 1). Along the same line of thought, one could also devise the most unfavorable instances of computation/communication time assignments for the other PEs which suffer lengthy idle synchronization phases. It is clear that the synchronization penalty for iterative algorithms can be severe.



Asynchronous Iterative Algorithms. Fig. 1 Very long idle periods due to synchronization between A and B executing synchronous iterations. *Arrows* show exchange messages, the horizontal axis is time, *boxes with numbers* denote iteration steps, and *dotted boxes* denote idle time spent in the synchronization. Note that such boxes will be absent in asynchronous iterations

Asynchronous Iterations: Basic Idea and Convergence Issues

The essence of asynchronous iterative algorithms is to reduce synchronization penalty by simply eliminating the synchronization phases in iterative algorithms described above. In other words, each PE is permitted to proceed to its next iteration step without waiting for updating its data from other PEs. Use newly arriving data if available, but otherwise reuse the old data. This approach, however, fails to retain the temporal ordering of the operations in the original algorithm. The established convergence properties are usually altered as a consequence. The convergence behavior of the new asynchronous algorithm is much harder to analyze than the original [20], the main difficulty being the introduction of multiple independent time lines in practice, one for each PE. Even though the convergence analysis uses a global time line, the state space does not involve studying the behavior of the trajectory of only one point (i.e., the shared data at the end of each iteration as in synchronous iterations) but rather of a set of points, one for each PE. An extra complexity is injected by the flexibility of the local operator fragments f_i to be applied to rather arbitrary combinations of data components from the evolution history of their argument lists. Although some assumptions may underly a particular asynchronous algorithm so that certain scenarios of reusing old data components are excluded, a PE is free to use, in a later iteration, data components older than those of the current one. In other words, one can have non-FIFO communication links for local data updates between the PEs.

The most general strategy for establishing convergence of asynchronous iterations for a certain problem is to move along the lines of the Asynchronous Convergence Theorem (ACT) [13]. One tries to construct a sequence of boxes (cartesian products of intervals where data components are known to lie) with the property that the image of each box under $f(\cdot)$ will be contained in the next box in this sequence, given that the original (synchronous) iterative algorithm converges. This nested box structure ensures that as soon as all local states enter such a box, no communication scenario can make any of the states escape to an outer box (deconverge), since updating variables through communication is essentially a coordinate exchange. On the other hand the synchronous convergence assumption guarantees that local computation also cannot produce any state escaping the current box, its only effect being a possible inclusion of some of the local data components in the respective intervals defining some smaller (nested) box. The argument here is that if communications are nothing but state transitions parallel to box facets and computations just drive some state coordinates in the facet of some smaller (nested) box, then their combination will continuously drive the set of local states to successively smaller boxes and ultimately toward convergence within a tolerance.

It follows that since a synchronous iteration is just a special case in this richer framework, its asynchronous counterpart will typically fail to converge for all asynchronous scenarios of minimal assumptions. Two broad classes of asynchronous scenarios are usually identified:

Totally asynchronous The only constraint here is that, in the local computations, data components are ultimately updated. Thus, no component ever becomes arbitrarily old as the global time line progresses, potentially indefinitely, assuming that any local computation marks a global clock tick. Under such a constraint, ACT can be used to prove, among other things, that the totally asynchronous execution of the classical iteration $x \leftarrow Ax + b$, where x and b are vectors and A a matrix distributed row-wise, converges to the correct solution provided that the spectral radius of the modulus matrix is less than unity ($\rho(|A|) < 1$). This is a very interesting result from a practical standpoint as it directly applies to classical stationary iterative algorithms (e.g., Jacobi) with nonnegative iteration matrix, those commonly used for benchmarking the asynchronous model implementations versus the synchronous ones.

Partially asynchronous The constraint here is stricter: each PE must perform at least one local iteration within the next B global clock ticks (for B a fixed integer) and it must not use components that are computed B ticks prior or older. Moreover, for the locally computed components, the most recent must always be used. Depending on how restricted the value of B may be, partially asynchronous algorithms can be classified in two types, Type I and Type II. The Type I is guaranteed to

converge for arbitrarily chosen B. An interesting case of Type I has the form $x \leftarrow Ax$ where x is a vector and A is a column-stochastic matrix. This case arises in computing stationary distributions of Markov chains [24] like the PageRank computation. Another interesting case is found in gossip-like algorithms for the distributed computation of statistical quantities, where $A = A(t)$ is a time-varying row-stochastic matrix [16]. For Type II partially asynchronous algorithms, B is restricted to be a function of the structure and the parameters of the linear operator itself to ensure convergence. Examples include gradient and gradient projection algorithms, e.g., those used in solving optimization and constrained optimization problems respectively. In these examples, unfortunately, the step parameter in the linearized operator must vary as inversely proportional to B in order to attain convergence. This implies that, although choosing large steps could accelerate the computation, the corresponding B may be too small to enforce in practice.

Implementation Issues

Termination Detection

In a synchronous iterative algorithm, global convergence can be easily decided. It can consist of a local convergence detection (e.g., to make sure a certain norm is below a local threshold in a PE) which triggers global convergence tests in all subsequent steps. The global tests can be performed by some PE acting as the monitor to check, at the end of each step, if the error (determined by substituting the current shared data x in the fixed point equation) gets below a global threshold. At that point, the monitor can signal all PEs to terminate. However, in an asynchronous setting, local convergence detection at some PE does not necessarily mean that the computation is near the global convergence, due to the fact that such a PE may compute too fast and not receive much input from its possibly slow communication in-links. Thus, in effect, there could also be the case in which each PE computes more or less in isolation from the others an approximation to a solution to the less-constrained fixed point problem concerning its respective local operator fragment $f_i(\cdot)$ but not $f(\cdot)$. Hence, local convergence detection may trigger the global convergence tests too early. Indeed, experiments have shown long phases in which PEs continually enter and exit their "locally converged" status. With the introduction of an extra waiting period for the local convergence status to stabilize, one can avoid the premature trigger of the global convergence detection procedure which may be either centralized or distributed. Taking centralized global detection for example, there is a monitor PE which decides global convergence and notifies all iterating PEs accordingly. A practical strategy is to introduce two integer "persistence parameters" (a localPersistence and a globalPersistence). If local convergence is preserved for more than localPersistence iterations, the PE will notify the monitor. As soon as the monitor finds out that all the PEs remain in locally convergence status for more than globalPersistence of their respective checking cycles, it signals global convergence to all the PEs.

Another solution, with a simple proof of correctness, is to embed a conservative iteration number in messages, defined as the smallest iteration number of all those messages used to construct the argument list of the current local computation, incremented by one [9]. This is strongly reminiscent of the ideas in the ACT and practically the iteration number is the box counter in that context, where nested boxes are marked by successively greater numbers of such. In this way, one can precompute some bound for the target iteration number for a given tolerance, with simple local checks and minimal communication overhead. A more elaborate scheme for asynchronous computation termination detection in a distributed setting can be found in literature [1].

Asynchronous Communication

The essence of asynchronous algorithms is not to let local computations be blocked by communications with other PEs. This nonblocking feature is easier to implement on a shared memory system than on a distributed memory system. This is because on a shared memory system, computation is performed either by multiple threads or multiple processes which share the physical address space. In either case, data communication can be implemented by using shared buffers. When a PE sends out a piece of data, it locks the data buffer until the write completes. If another PE tries to receive the latest data but finds the data buffer locked, instead

of blocking itself, the PE can simply continue iterating the computation with its old, local copy of the data. On the other hand, to make sure the receive operation retrieves the data in the buffer in its entirety, the buffer must also be locked until the receive is complete. The sending PE hence may also find itself locked out of the buffer. Conceptually, one can give the send operation the privilege to preempt the receive operation, and the latter may erase the partial data just retrieved and let the send operation deposit the most up to date data. However, it may be easier to just let the send operation wait for the lock, especially because the expected waiting will be much shorter than what is typically experienced in a distributed memory system.

On distributed memory systems, the communication protocols inevitably perform certain blocking operations. For example, on some communication layer, a send operation will be considered incomplete until the receiver acknowledges the safe receipt of the data. Similarly, a receive operation on a certain layer will be considered incomplete until the arriving data become locally available. From this point of view, a communication operation in effect synchronizes the communicating partners, which is not quite compatible with the asynchronous model. For example, the so-called nonblocking send and probe operations assume the existence a hardware device, namely, the network interface card that is independent of the PE. However, in order to avoid expensive copy operations between the application and the network layers, the send buffer is usually shared by both layers, which means that before the PE can reuse the buffer, e.g., while preparing for the next send in the application layer, it must wait until the buffer is emptied by the network layer. (Note that the application layer cannot force the network layer to undo its ongoing send operation.) This practically destroys the asynchronous semantics. To overcome this difficulty, computation and communication must be decoupled, e.g., implemented as two separate software modules, such that the blocking due to communication does not impede the computation. Mechanisms must be set up to facilitate fast data exchange between these two modules, e.g., by using another data buffer to copy data between them. Now, the computation module will act like a PE sending data in a shared memory system while the communication module acts like a PE receiving data. One must also note that multiple probes by a receiving PE are necessary since multiple messages with the same "envelope" might have arrived during a local iteration. This again has a negative impact on the performance of an asynchronous algorithm, because the receiving PE must find the most recent of these messages and discard the rest.

The issues listed above have been discussed in the context of the MPI library [10]. Communication libraries such as Jace [7] implemented in Java or its C++ port, CRAC [18], address some of the shortcomings of "general-purpose" message passing libraries. Internally they use separate threads for the communication and computation activities with synchronized queues of buffers for the messages and automatic overwriting on the older ones.

Recent Developments and Potential Future Directions

Two-Stage Iterative Methods and Flexible Communications

There exist cases in which an algorithm can be restructured so as to introduce new opportunities for parallelization and thus yield more places to inject asynchronous semantics. Notable examples are two-stage iterative methods for solving linear systems of equations of the form $Ax = b$. Matrix A is written as $A = M - N$ with M being a block diagonal part. With such splitting, the iteration reads as $Mx(t+1) \leftarrow (Nx(t) + b)$, and its computation can be decomposed row-wise and distributed to the PEs, i.e., a block Jacobi iteration. However, to get a new data fragment $x_i(t+1)$ at each iteration step at some PE, a new linear system of equations must be solved, which can be done by a new splitting local M_i part, resulting in a nested iteration. In a synchronous version, new data fragments exchange only at the end of each iteration to coordinate execution. The asynchronous model applied in this context [22] not only relaxes the timing in $x_i(t+1)$ exchanges in the synchronous model, but it also introduces asynchronous exchanges even during the nested iterations (toward $x_i(t+1)$) and their immediate use in the computation and not at the end of the respective computational phases. This idea was initially used for solving systems of nonlinear equations [25].

Asynchronous Tiled Iterations (*or* Parallelizing Data Locality)

As mentioned in Sect. 3, the PEs executing the operator and the MEs hosting its parameters and data must have fast interconnection paths. Restructuring the computation so as to maximize reuse of cached data across iterations have been studied in the past [27]. These are tiling techniques [34] applied to chunks of iterations (during which convergence is not tested) and coupled with strategies for breaking the so induced dependencies. In this way data locality is considerably increased but opportunities for parallelization are confined only within the current tile data and not the whole data set as is the general case, e.g., in iterative stencil computations. Furthermore, additional synchronization barriers, scaling with the number of the tiles in number, are introduced. In a very recent work [23], the asynchronous execution of tiled chunks is proposed for regaining the parallelization degree of nontiled iterations: each PE is assigned a set of tiles (its sub-grid) and performs the corresponding loops without synchronizing with the other PEs. Only the convergence test at the end of such a phase enforces synchronization. So on the one hand, locality is preserved since each PE traverses its current tile data only and on the other hand all available PEs execute concurrently in a similar fashion without synchronizing, resulting in a large degree of parallelization.

When to Use Asynchronous Iterations?

Asynchronism can enter an iteration in both natural and artificial ways. In naturally occurring asynchronous iterations, PEs are either asynchronous by default (or it is unacceptable to synchronize) computation over sensor networks or distributed routing over data networks being such examples.

However, the asynchronous execution of a parallelized algorithm enters artificially, in the sense that most of the times it comes as a variation of the synchronous parallel port of a sequential one. Typically, there is a need to accelerate the sequential algorithm and as a first step it is parallelized, albeit in synchronous mode in order to preserve semantics. Next, in the presence of large synchronization penalties (when PEs are heterogeneous both in terms of computation and communication as in some Grid installations) or extra flexibility needs (such as asynchronous computation starts, dynamic changes in data or topology, non-FIFO communication channels), asynchronous implementations are evaluated. Note that in all those cases of practical interest, networked PEs are implied. The interesting aspect of [23, 36] is that it broadens the applicability of the asynchronous paradigm in shared memory setups for yet another purpose, which is to preserve locality but without losing parallelism itself as a performance boosting strategy.

Related Entries

►Memory Models
►Synchronization

Bibliographic Notes and Further Reading

The asynchronous computation model, as an alternative to the synchronous one, has a life span of almost 4 decades. It started with its formal description in the pioneering work of Chazan and Miranker [17] and its first experimental investigations by Baudet [8] back in the 1970s. Perhaps the most extensive and systematic treatment of the subject is contained in a book by Bertsekas and Tsitsiklis in the 1980s [14], particularly in its closing three chapters. During the last 2 decades an extensive literature has been accumulated [4, 11, 12, 15, 19, 21, 26, 29, 30, 32, 33]. Most of these works explore theoretical extensions and variations of the asynchronous model coupled with very specific applications. However in the most recent ones, focus has shifted to more practical, implementation-level aspects [28, 31, 36], since the asynchronous model seems appropriate for the realization of highly heterogeneous, Internet-scale computations [2, 3, 6].

Bibliography

1. Bahi JM, Contassot-Vivier S, Couturier R, Vernier F (2005) A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. IEEE Trans Parallel Distrib Syst 16(1):4–13
2. Bahi JM, Contassot-Vivier S, Couturier R (2002) Asynchronism for iterative algorithms in a global computing environment. In: 16th Annual International Symposium on high performance computing systems and applications (HPCS'02). IEEE, Moncton, Canada, pp 90–97

3. Bahi JM, Contassot-Vivier S, Couturier R (2003) Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational Grid. In: 17th International Parallel and Distributed Processing Symposium (IPDPS'03), p 40. IEEE, Nice, France
4. Bahi JM, Contassot-Vivier S, Couturier R (2004) Performance comparison of parallel programming environments for implementing AIAC algorithms. In: 18th International Parallel and Distributed Processing Symposium (IPDPS'04). IEEE, Santa Fe, USA
5. Bahi JM, Contassot-Vivier S, Couturier R (2007) Parallel iterative algorithms: from sequential to Grid computing. Chapman & Hall/CRC, Boca Raton, FL
6. Bahi JM, Domas S, Mazouzi K (2004) Combination of Java and asynchronism for the Grid: a comparative study based on a parallel power method. In: 18th International Parallel and Distributed Processing Symposium (IPDPS '04), pp 158a, 8. IEEE, Santa Fe, USA, April 2004
7. Bahi JM, Domas S, Mazouzi K (2004). Jace: a Java environment for distributed asynchronous iterative computations. In: 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (EUROMICRO-PDP'04), pp 350–357. IEEE, Coruna, Spain
8. Baudet GM (1978) Asynchronous iterative methods for multiprocessors. JACM 25(2):226–244
9. El Baz D (1996) A method of terminating asynchronous iterative algorithms on message passing systems. Parallel Algor Appl 9:153–158
10. El Baz D (2007) Communication study and implementation analysis of parallel asynchronous iterative algorithms on message passing architectures. In: Parallel, distributed and network-based processing, 2007. PDP '07. 15th EUROMICRO International Conference, pp 77–83. Weimar, Germany
11. El Baz D, Gazen D, Jarraya M, Spiteri P, Miellou JC (1998) Flexible communication for parallel asynchronous methods with application to a nonlinear optimization problem. D'Hollander E, Joubert G et al (eds). In: Advances in Parallel Computing: Fundamentals, Application, and New Directions. North Holland, vol 12, pp 429–436
12. El Baz D, Spiteri P, Miellou JC, Gazen D (1996) Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems. J Parallel Distrib Comput 38(1):1–15
13. Bertsekas DP (1983) Distributed asynchronous computation of fixed points. Math Program 27(1):107–120
14. Bertsekas DP, Tsitsiklis JN (1989) Parallel and distributed computation. Prentice-Hall, Englewood Cliffs, NJ
15. Blathras K, Szyld DB, Shi Y (1999) Timing models and local stopping criteria for asynchronous iterative algorithms. J Parallel Distrib Comput 58(3):446–465
16. Blondel VD, Hendrickx JM, Olshevsky A, Tsitsiklis JN (2006) Convergence in multiagent coordination, consensus, and flocking. In: Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC'05. 44th IEEE Conference on, pp 2996–3000
17. Chazan D, Miranker WL (1969) Chaotic relaxation. J Linear Algebra Appl 2:199–222
18. Couturier R, Domas S (2007) CRAC: a Grid environment to solve scientific applications with asynchronous iterative algorithms. In: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, p 1–8
19. Elsner L, Koltracht I, Neumann M (1990) On the convergence of asynchronous paracontractions with application to tomographic reconstruction from incomplete data. Linear Algebra Appl 130:65–82
20. Frommer A, Szyld DB (2000) On asynchronous iterations. J Comput Appl Math 123(1–2):201–216
21. Frommer A, Schwandt H, Szyld DB (1997) Asynchronous weighted additive schwarz methods. ETNA 5:48–61
22. Frommer A, Szyld DB (1994) Asynchronous two-stage iterative methods. Numer Math 69(2):141–153
23. Liu L, Li Z (2010) Improving parallelism and locality with asynchronous algorithms. In: 15th ACM SIGPLAN Symposium on principles and practice of parallel programming (PPoPP), pp 213–222, Bangalore, India
24. Lubachevsky B, Mitra D (1986) A chaotic, asynhronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius. JACM 33(1):130–150
25. Miellou JC, El Baz D, Spiteri P (1998) A new class of asynchronous iterative algorithms with order intervals. Mathematics of Computation, 67(221):237–255
26. Moga AC, Dubois M (1996) Performance of asynchronous linear iterations with random delays. In: Proceedings of the 10th International Parallel Processing Symposium (IPPS '96), pp 625–629
27. Song Y, Li Z (1999) New tiling techniques to improve cache temporal locality. ACM SIGPLAN Notices ACM SIGPLAN Conf Program Lang Design Implement 34(5):215–228
28. Spiteri P, Chau M (2002) Parallel asynchronous Richardson method for the solution of obstacle problem. In: Proceedings of 16th Annual International Symposium on High Performance Computing Systems and Applications, Moncton, Canada, pp 133–138
29. Strikwerda JC (2002) A probabilistic analysis of asynchronous iteration. Linear Algebra Appl 349(1–3):125–154
30. Su Y, Bhaya A, Kaszkurewicz E, Kozyakin VS (1998) Further results on convergence of asynchronous linear iterations. Linear Algebra Appl 281(1–3):11–24
31. Szyld DB (2001) Perspectives on asynchronous computations for fluid flow problems. First MIT Conference on Computational Fluid and Solid Mechanics, pp 977–980
32. Szyld DB, Xu JJ (2000) Convergence of some asynchronous nonlinear multisplitting methods. Num Algor 25(1–4):347–361
33. Uresin A, Dubois M (1996) Effects of asynchronism on the convergence rate of iterative algorithms. J Parallel Distrib Comput 34(1):66–81
34. Wolfe M (1989) More iteration space tiling. In: Proceedings of the 1989 ACM/IEEE conference on Supercomputing, p 664. ACM, Reno, NV

35. Kollias G, Gallopoulos E, Szyld DB (2006) Asynchronous iterative computations with Web information retrieval structures: The PageRank case. In: Joubert GR, Nagel WE, et al (Eds). Parallel Computing: Current and Future issues of High-End computing, NIC Series. John von Neumann-Institut für Computing, Jülich, Germany, vol 33, pp 309–316
36. Kollias G, Gallopoulos E (2007) Asynchronous Computation of PageRank computation in an interactive multithreading environment. In: Frommer A, Mahoney MW, Szyld DB (eds) Web Information Retrieval and Linear Algebra Algorithms, Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl, Germany, ISSN: 1862–4405

Asynchronous Iterative Computations

►Asynchronous Iterative Algorithms

ATLAS (Automatically Tuned Linear Algebra Software)

R. Clint Whaley
University of Texas at San Antonio, San Antonio, TX, USA

Synonyms

Numerical libraries

Definition

ATLAS [20–22, 24, 25] is an ongoing research project that uses empirical tuning to optimize dense linear algebra software. The fruits of this research are embodied in an empirical tuning framework available as an open source/free software package (also referred to as "ATLAS"), which can be downloaded from the ATLAS homepage [23]. ATLAS generates optimized libraries which are also often collectively referred to as "ATLAS," "ATLAS libraries," or more precisely, "ATLAS-tuned libraries." In particular, ATLAS provides a full implementation of the BLAS [6, 7, 10, 12] (Basic Linear Algebra Subprograms) API, and a subset of optimized LAPACK [1] (Linear Algebra PACKage) routines. Because dense linear algebra is rich in operand reuse, many routines can run tens or hundreds of times faster when tuned for the hardware than when written naively. Unfortunately, highly tuned codes are usually not performance portable (i.e., a code transformation that helps performance on architecture A may reduce performance on architecture B).

The BLAS API provides basic building block linear algebra operations, and was designed to help ease the performance portability problem. The idea was to design an API that provides the basic computational needs for most dense linear algebra algorithms, so that when this API has been tuned for the hardware, all higher-level codes that rely on it for computation automatically get the associated speedup. Thus, the job of optimizing a vast library such as LAPACK can be largely handled by optimizing the much smaller code base involved in supporting the BLAS API. The BLAS are split into three "levels" based on how much cache reuse they enjoy, and thus how computationally efficient they can be made to be. In order of efficiency, the BLAS levels are: Level 3 BLAS [6], which involve matrix–matrix operations that can run near machine peak, Level 2 BLAS [7, 8] which involve matrix–vector operations and Level 1 BLAS [10, 12], which involve vector–vector operations. The Level 1 and 2 BLAS have the same order of memory references as floating point operations (FLOPS), and so will run at roughly the speed of memory for out-of-cache operation.

The BLAS were extremely successful as an API, allowing dense linear algebra to run at near-peak rates of execution on many architectures. However, with hardware changing at the frantic pace dictated by Moore's Law, it was an almost impossible task for hand tuners to keep BLAS libraries up-to-date on even those fortunate families of machines that enjoyed their attentions. Even worse, many architectures did not have anyone willing and able to provide tuned BLAS, which left investigators with codes that literally ran orders of magnitude slower than they should, representing huge missed opportunities for research. Even on systems where a vendor provided BLAS implementations, license issues often prevented their use (e.g., SUN provided an optimized BLAS for the SPARC, but only licensed its use for their own compilers, which left researchers using other languages such as High Performance Fortran without BLAS; this was one of the original motivations to build ATLAS).