Metadata of the chapter that will be visualized online

Book Title	Encyclopedia of Parallel Computing	
Book Copyright - Year	2011	
Copyright Holder	Springer Science+Business Media LLC	
Title	Computational Sciences and Parallelism	
Author	Role	Professor
	Given Name	**Geoffrey**
	Particle	
	Family Name	**Fox**
	Suffix	
	Phone	1-812-219-4643
	Fax	
	Email	gcf@indiana.edu
Affiliation	Division	School of Informatics and Computing and Pervasive Technology Institute
	Organization	Indiana University
	Street	Lindley Hall
	Postcode	47405
	City	Bloomington
	State	IN
	Country	USA

C

Computational Sciences and Parallelism

Geoffrey Fox
Indiana University, Bloomington, IN, USA

Synonyms

Applications and Parallelism

Problem Architectures

Definition

Here it is asked which applications should run in parallel and correspondingly which areas of computational science will benefit from parallelism. In studying this it will be discovered which applications benefit from particular hardware and software choices. A driving principle is that in parallel programming, one must map problems into software and then into hardware. The architecture differences in source and target of these maps will affect the efficiency and ease of parallelism.

Discussion

Introduction

I have an application – can and should it be implemented on a parallel architecture and if so, how should this be done and what are appropriate target hardware architectures, what is known about clever algorithms and what are recommended software technologies? Fox introduced in [1] a general approach to this question by considering problems and the computer infrastructure on which they are executed as complex systems. Namely each is a collection of entities and connections between them governed by some laws. The entities can be illustrated by mesh points, particles, and data points for problems; cores, networks, and storage locations for hardware; objects, instructions, and messages for software. The processes of deriving numerical models, generating the software to simulate model, compiling the software, generating the machine code, and finally executing the program on particular hardware can be considered as maps between different complex systems. Many performance models and analyses have been developed and these describe the quality of map. It is known that maps are essentially never perfect and describing principles for quantifying this is a goal of this entry. At a high level, it is understood that the architecture of problem and hardware/software must match; given this we have quantitative conditions that the performance of the parts of the hardware must be consistent with the problem. For example, if two mesh points in problem are strongly connected, then bandwidth between components of hardware to which they are mapped must be high. In this discussion, the issues of parallelism are being described and here there are two particularly interesting general results. Firstly a space (the domain of entities) and a time associated with a complex system are usually defined. Time is nature's time for the complex system that describes time dependent simulations. However, for linear algebra, time for that complex system is an iteration count. Note that for the simplest sequential computer hardware there is no space and just a time defined by the control flow. Thus in executing problems on computers one is typically mapping all or part of the space of the problem onto time for the computer and parallel computing corresponding case where both problem and computer have well defined spatial extent. Mapping is usually never 1:1 and reversible, and "information is lost" as one maps one system into another. In particular, one fundamental reason why automatic parallelism can be hard is that the mapping of problem into software has thrown away key information about the space-time structure of original problem. Language designers in this field try to find languages that preserve key information needed for parallelism while hardware designers design computers

David Padua (ed.), *Encyclopedia of Parallel Computing*, DOI 10.1007/978-0-387-09766-4,

that can work around this loss of information. For an example, use of arrays in many data parallel languages from APL, HPF, to Sawzall can be viewed as a way to preserve spatial structure of problems when expressed in these languages. In this article, these issues will not be discussed in depth but rather it will be discussed what is possible with "knowledgeable users" mapping problems to computers or particular programming paradigms.

Simple Example

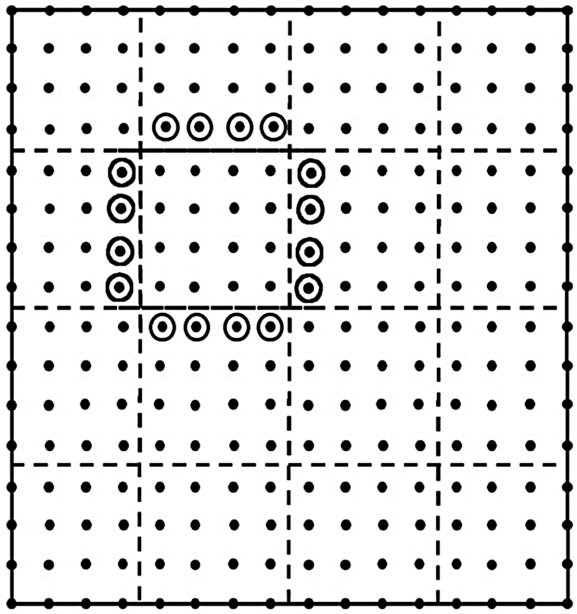
The simple case of a problem whose complex system spatial structure is represented as a 2D mesh is considered. This comes in material science when one considers local forces between a regular array of particles or in the finite difference approach to solving Laplace or Poisson's equation in two dimensions. There are many important subtleties such as adaptive meshes and hierarchical multigrid methods but in the simplest formulation such problems are set up as a regular grid of field values where the basic iterative update links nearest neighbors in two dimensions.

If the points are labeled by an index pair (i, j), then Jacobi's method (not state of the art but chosen as simplicity allows a clear discussion) can be written

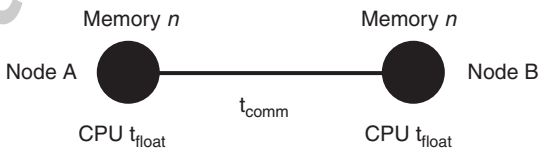
$$\phi(i,j) \text{ is replaced by } (\phi_{\text{Left}}+\phi_{\text{Right}}+\phi_{\text{Up}}+\phi_{\text{Down}})/4 \quad (1)$$

where $\phi_{\text{Left}} = \phi(i, j-1)$ and similarly for ϕ_{Right}, ϕ_{Up}, and ϕ_{Down}.

Such problems would usually be implemented on a parallel architecture by a technique that is often called "domain decomposition" or "data parallelism" although these terms are not very precise. Parallelism is naturally found for such problems by dividing the domain up into parts and assigning each part to a different processors as seen in Fig. 1. Here the problem represented as a 16×16 mesh is solved on a 4×4 mesh of processors. For problems coming from nature this geometric view is intuitive as say in a weather simulation, the atmosphere over California evolves independently from that over Indiana and so can be simulated on separate processors. This is only true for short time extrapolations – eventually information flows between these sites and their dynamics are mixed. Of course it is the communication of data between the processors (either directly in a distributed memory or implicitly in a shared memory) that implements this eventual mixing.



Computational Sciences and Parallelism. Fig. 1 Communication structure for 2D complex system example. The dots are the 256 points in the problem. Shown by dashed lines is the division into 16 processors. The circled points are the halo or ghost grid points communicated to processor they surround



Computational Sciences and Parallelism. Fig. 2 Parameters determining performance of loosely synchronous problems

Such block data decompositions typically lead to a SPMD (Single Program Multiple Data) structure with each processor executing the same code but on different data points and with differing boundary conditions. In this type of problem, processors at the edge of the (4×4) mesh do not see quite the same communication and compute complexity as the "general case" of an inside processor shown in Fig. 1. For the local nearest neighbor structure of Eq. 1, one needs to communicate the ring of halo points shown in figure. As computation grows like the number of points (grain size) n in

each processor and communication like the number on edge (proportional to $\sqrt{n}$), the time "wasted" communicating decreases as a fraction of the total as the grain size n increases. Further one can usually "block" the communication to transmit all the needed points in a few messages as latency can be an important part of communication overhead.

Note that this type of data decomposition implies the so-called "owner's-compute" rule. Here each data point is imagined as being owned by the processor to which the decomposition assigns it. The owner of a given data-point is then responsible for performing the computation that "updates" its corresponding data values. This produces a common scenario where parallel program consists of a loop over iterations divided into compute-communicate phases:

- *Communicate*: At the start of each iteration, first communicate any outside data values needed to update the data values at points owned by this processor.
- *Compute*: Perform update of data values with each processor operating without need to further synchronize with other machines.

This general structure is preserved even in many complex physical simulations with fixed albeit irregular decompositions. Dynamic decompositions introduce a further step where data values are migrated between processors to ensure load balance but this is usually still followed by similar communicate-compute phases. The communication phase naturally synchronizes the operation of the parallel processors and provides an efficient barrier point which naturally scales. The above discussion uses a terminology natural for distributed memory hardware or message passing programming models. With a shared memory model like OpenMP, communication would be implicit and the "communication phase" above would be implemented as a barrier synchronization.

Performance Model

Our current Poisson equation example can be used to illustrate some simple techniques that allow estimates of the performance of many parallel programs.

As shown in Fig. 5, the node of a parallel machine is characterized by a parameter t_{float}, which is time taken for a single floating point operation. t_{float} is of course not very well defined as depends on the effectiveness of cache, possible use of fused multiply-add and other issues. This implies that this measure will have some application dependence reflecting the goodness of the match of the problem to the node architecture. We let n be the grain size – the number of data points owned by a typical processor. Communication performance – whether through a shared or distributed memory architecture – can be parameterized as

$$\begin{aligned}&\text{Time to communicate } N_{\text{comm}} \text{ words}\\ &= t_{\text{latency}} + N_{\text{comm}} \cdot t_{\text{comm}} \qquad (2)\end{aligned}$$

This equation ignores issues like bus or switch contention but is a reasonable model in many cases. Latencies t_{latency} can be around 1 μs on high performance systems but is measured in milliseconds in a geographically distributed grid. t_{comm} is time to communicate a single word and for large enough messages, the latency term can be ignored which will be done in the following.

Parallel performance is dependent on load balancing and communication and both can be discussed but here it is focused on communication with problem of Fig. 1 generalized to N_{proc} processors arranged in an $\sqrt{N_{\text{proc}}}$ by $\sqrt{N_{\text{proc}}}$ grid with a total of N mesh points and the grain size $n = N/N_{\text{proc}}$. Let $T(N_{\text{proc}})$ be the execution time on N_{proc} processors and two contributions are found to this ignoring small load imbalances from edge processors. There is a calculation time expressed as $n \cdot t_{\text{calc}}$ with $t_{\text{calc}} = 4t_{\text{float}}$ as the time to execute the basic update Eq. 1. In addition the parallel program has communication overhead, which adds to $T(N_{\text{proc}})$ a term $4\sqrt{n} \cdot t_{\text{comm}}$. Now the speed up formula is found:

$$\begin{aligned}S(N_{\text{proc}}) &= T(1)/T(N_{\text{proc}})\\ &= N_{\text{proc}}/(1 + t_{\text{comm}}/(\sqrt{n} \cdot t_{\text{float}})) \qquad (3)\end{aligned}$$

It is noted that this analysis ignores the possibility available on some computers of overlapping communication and computation which is straightforwardly included. The above formalism can be generalized most conveniently using the notation that

$$S(N_{\text{proc}}) = \varepsilon \cdot N_{\text{proc}} = N_{\text{proc}}/(1 + f), \qquad (4)$$

which defines efficiency ε and overhead f. Note that it is preferred to discuss overhead rather than speed-up or efficiency as one typically gets simpler models for f as the effects of parallelism are additive to f but for example occur in the denominator of Eq. 3 for speedup

and efficiency. The communication part f_{comm} of the overhead f is given by combining Eqs. 3 and 4 as

$$f_{\text{comm}} = t_{\text{comm}}/(\sqrt{n} \cdot t_{\text{float}}) \quad (5)$$

Note that in many instances, f_{comm} can be thought of as simply the ratio of parallel communication to parallel computation. This equation can be generalized to essentially all problems we will later term loosely synchronous. Then in each coupled communicate-compute phases of such problems, one finds that the overhead takes the form:

$$f_{\text{comm}} = constant \cdot t_{\text{comm}}/(n^{1/d} \cdot t_{\text{float}}) \quad (6)$$

Here d is an appropriate (complexity or information) dimension, which is equal to the geometric dimension for partial differential based equations or other geometrically local algorithms such as particle dynamics. A particularly important case in practice is the 3D value $d = 3$ when $n^{-1/d}$ is just surface/volume in three dimensions. However Eq. 6 describes many non geometrically local problems with for example the value $d = 2$ for the best decompositions for full matrix linear algebra and $d = 1$ for long range interaction problems. The Fast Fourier Transform FFT finds $n^{1/d}$ in Eq. 6 replaced by $\ln(n)$ corresponding to $d = \infty$.

From Eq. 4, it can be found that $S(N_{\text{proc}})$ increases linearly with N_{proc} as long as N_{proc} is increased with fixed f_{comm} which implies fixed grain size n, while t_{comm} and t_{float} are naturally fixed. This is scaled speedup where the problem size $N = n \cdot N_{\text{proc}}$ also increases linearly with N_{proc}. The continuing success of parallel computing even on very large machines can be considered as a consequence of equations like Eq. 6 as the formula for f_{comm} only depends on local node parameters and not on the number of processors. Thus as we scale up the number of processors keeping the node hardware and application grain size n fixed, we will get scaling performance – speedup proportional to N_{proc}. Note this implies that total problem size increases proportional to N_{proc} – the defining characteristic of scaled speedup.

Complex Applications Are Better for Parallelism

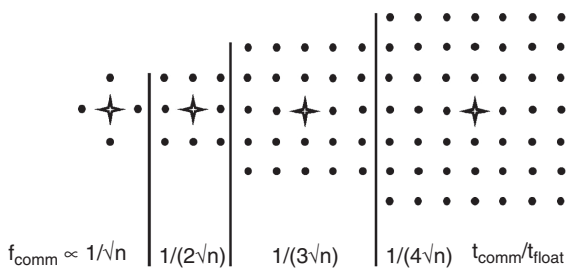
The simple problem described above is perhaps the one where the parallel issues are most obvious; however it is not the one where good parallel performance is easiest to obtain as the small computational complexity of the update Eq. 1 makes the communication overhead relatively more important. There is a fortunate general rule that as one increases the complexity of a problem, the computation needed grows faster than the communication overhead and this will be illustrated below. Jacobi iteration does have perhaps the smallest communication for problems of this class. However it has one of largest ratios of communication to computation and correspondingly high parallel overhead. Note one sees the same effect on a hierarchical (cache) memory machine, where problems such as Jacobi Iteration for simple equations can perform poorly as the number of operations performed on each word fetched into cache is proportional to number of links per entity and this is small (four in the 2-D mesh considered above) for this problem class.

As an illustration of the effect varying computational complexity, it can be seen in Fig. 3 how the above analysis is altered as one changes the update formula of Eq. 1. The size of the stencil parameterized can now be systematically increased by an integer l and it can be found how f_{comm} changes. In the case where points are particles the value of l corresponds to the range of their mutual force and in the case of discretization of partial differential equations l measures the order of the approximation.

The communication overhead is found to decrease systematically as shown in Fig. 3 as the range of the force increases. The general 2D result is:

$$f_{\text{comm}} \propto t_{\text{comm}}/(l \cdot \sqrt{n} \cdot t_{\text{float}}) \quad (7)$$

This is valid for l which is large compared to 1 but smaller than the length scale corresponding to region



Computational Sciences and Parallelism. Fig. 3 Communication structure as a function of stencil size. The figure shows 4 stencils with from left to right, range $l = 1, 1, 2, 3$

stored in each processor. In the interesting limit of an infinite range ($l \to \infty$) force, the analysis needs to be redone and one finds the result that is independent of the geometric dimension

$$f_{\text{comm}} \propto t_{\text{comm}}/(n \cdot t_{\text{float}}) \tag{8}$$

which is of the general form of Eq. 6 with complexity dimension $d = 1$. This is the best-understood case where the geometric and complexity dimensions are different. The overhead formula of Eq. 8 corresponds to the computationally intense $O(N^2)$ algorithms for evolving N-body problems. The amount of computation is so large that the ratio of communication to computation is extremely small.

Application Architectures

The analysis above can be applied to many SPMD problems and addresses the matching of "spatial" structure of applications and computers. This drives needed linkage of individual computers in a parallel system in terms of topology and performance of network. However this only works if we can match the temporal structure and this aspect is more qualitative and perhaps controversial. The simplest ideas here underlied the early SIMD (Single Instruction Multiple Data) machines that were popular some 20 years ago. These are suitable for problems where each point of the complex system evolves with the same rule (mapping into machine instruction) at each time. There are many such problems including for example the Laplace solver discussed above. However many related problems do not fit this structure – called *synchronous* in [1] – with the simplest reason being heterogeneity in system requiring different computational approaches at different points. A huge number of scientific problems fit a more general classification – *loosely synchronous*. Here SPMD applications can be seen which have the compute-communication stages described above but now the compute phases are different on different processors. One uses load balancing methods to ensure that the computational work on each node is balanced but not on each machine instruction but rather in a coarse grain fashion at every iteration or time-step – whatever defines the temporal evolution of the complete system. Loosely synchronous problems fit naturally MIMD machines with the communication stages at macroscopic "time-steps" of the application. This communication ensures the overall correct synchronization of the parallel application. Thus overhead formulae like Eqs. 5 and 6 describe both communication and synchronization overhead. As this overhead only depends on local parameters of the application, it is understood why loosely synchronously can get good scalable performance on the largest supercomputers. Such applications need no expensive global synchronization steps. Essentially all linear algebra, particle dynamics and differential equation solvers fall in the loosely synchronous class. Note synchronous problems are still around but they are run on MIMD (Multiple Instruction Multiple Data) machines with the SPMD model.

A third class of problems – termed *asynchronous* – consists of asynchronously interacting objects and is often people's view of a typical parallel problem. It probably does describe the concurrent threads in a modern operating system and some important applications such as event driven simulations and areas like search in computer games and graph algorithms. Shared memory is natural for asynchronous problems due to low latency often needed to perform dynamic synchronization. It wasn't clear in the past but now it appears this category is not very common in large scale parallel problems of importance. The surprise of some at the practical success of parallel computing can perhaps be understood from people thinking about asynchronous problems whereas its loosely synchronous and *pleasingly parallel* problems that dominate. The latter class is the simplest algorithmically with disconnected parallel components. However the importance of this category has probably grown since the original 1988 analysis [2] when it was estimated as 20% of all parallel computing. Both Grids and clouds are very natural for this class which does not need high performance communication between different nodes. Parameter searches and many data analysis applications of independent observations fall into this class.

From the start, we have seen a fifth class – termed metaproblems – which refer to the coarse grain linkage of different "atomic" problems. Here synchronous, loosely synchronous, asynchronous and pleasingly parallel are the atomic classes. Metaproblems are very common and expected to grow in importance. One often uses a two level programming model in this case with

Computational Sciences and Parallelism. Table 1 Application classification

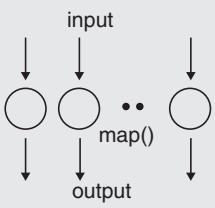
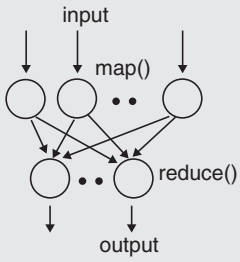
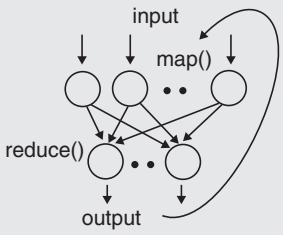
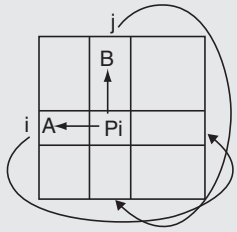
#	Class	Description	Machine Architecture
1	**Synchronous**	The problem class can be implemented with instruction level Lockstep Operation as in SIMD architectures	**SIMD**
2	**Loosely Synchronous (or BSP Bulk Synchronous Processing)**	These problems exhibit iterative Compute-Communication stages with independent compute (map) operations for each CPU that are synchronized with a communication step. This problem class covers many successful MPI applications including partial differential equation solution and particle dynamics applications.	**MIMD on MPP (Massively Parallel Processor)**
3	**Asynchronous**	Illustrated by Compute Chess and Integer Programming; Combinatorial Search often supported by dynamic threads. This is rarely important in scientific computing but at heart of operating systems and concurrency in consumer applications such as Microsoft Word.	**Shared Memory**
4	**Pleasingly Parallel**	Each component is independent. In 1988, Fox estimated this at 20% of the total number of applications [2] but that percentage has grown with the use of Grids and data analysis applications including for example the Large Hadron Collider analysis for particle physics.	**Grids** moving to **Clouds**
5	**Metaproblems**	These are coarse grain (asynchronous or dataflow) combinations of classes (1–4) and (6). This area has also grown in importance and is well supported by Grids and described by workflow of Section 3.5.	**Grids of Clusters**
6	**MapReduce++**	It describes file(database) to file(database) operations which has three subcategories given below and in Table 2. (6a) Pleasingly Parallel Map Only – similar to category 4 (6b) Map followed by reductions (6c) Iterative "Map followed by reductions" – Extension of Current Technologies that supports much linear algebra and data mining	**Data-intensive Clouds** (a) Master-Worker or MapReduce (b) MapReduce (c) Twister

The MapReduce++ category has three subdivisions (a) "map only" applications similar to pleasingly parallel category; (b) The classic MapReduce with file to file operations consisting of parallel maps followed by parallel reduce operations; (c) captures the extended MapReduce introduced in [4–10]. Note this category has the same complex system structure as loosely synchronous or pleasingly parallel problems but is distinguished by the reading and writing of data. This comparison is made clearer in Table 2. Note nearly all early work on parallel computing discussed computing with data on memory. MapReduce and languages like Sawzall [11] and Pig-Latin [12] emphasize the parallel processing of data on disks – a field that until recently was only covered by database community

the metaproblem linkage specified by workflow and the component problems with traditional parallel languages and runtimes. Grids or Clouds are suitable for metaproblems as coarse grain decomposition does not usually require stringent performance.

These five categories are summarized in Table 1 which also introduces a new category *MapReduce++* which has recently grown in importance to described data analysis. Nearly all the early work on parallel computing focused on simulation as opposed to data

Computational Sciences and Parallelism. Table 2 Comparison of MapReduce++ subcategories and Loosely Synchronous category

Map-only	Classic MapReduce	Iterative MapReduce	Loosely Synchronous
			
• Document conversion (e.g. PDF->HTML) • Brute force searches in cryptography • Parametric sweeps • Gene assembly • Much data analysis of independent samples	• High Energy Physics (HEP) Histograms • Distributed search • Distributed sort • Information retrieval • Calculation of Pairwise Distances for sequences (BLAST)	• Expectation maximization algorithms • Linear Algebra • Datamining including • Clustering • K-means • Multidimensional Scaling (MDS)	• Many MPI scientific applications utilizing wide variety of communication constructs including local interactions • Solving differential equations and • Particle dynamics with short range forces
← **Domain of MapReduce and Iterative Extensions** →			**MPI**

analysis (or what some call data intensive applications). Data analysis has exploded in importance recently [3] correspondingly to growth in number of instruments, sensors and human (the web) sources of data.

Summary

Problems are set up as computational or numerical systems and these can be considered as a "space" of linked entities evolving in time. The spatial structure (which is critical for performance) and the temporal structure which is critical to understand the class of software and computer needed were discussed. These were termed "basic complex systems" and characterized by their possibly dynamic spatial (geometric) and temporal structure. The difference between the structure of the original problem and that of computational system derived from it have been noted. Much of the past experience can be summarized in parallelizing applications by the conclusion:

Synchronous and Loosely Synchronous problems perform well on large parallel machines as long as the problem is large enough. For a given machine, there is a typical sub-domain size (i.e. the grain size or size of that part of the problem stored on each node) above which one can expect to get good performance. There will be a roughly constant ratio of parallel speedup to N_{proc} if one scales the problem with fixed sub-domain size and total size proportional to N_{proc}. This conclusion has been enriched by study of grids and clouds with an emphasize on pleasingly parallel and MapReduce++ style problems often with a data intensive focus. These also parallelize well.

Bibliographic Notes and Further Reading

The approach followed here was developed in [1, 13] with further details in [2, 14]. The extension to include data intensive applications was given in [8, 15]. There are many good discussions of speedup including Gustafson's seminal work [16] and the lack of it – Amdahl's law [17]. The recent spate of papers on MapReduce [4, 7] and its applications and extensions [4–12, 15] allow one to extend the discussion of parallelism from simulation (which implicitly dominated the early work) to data analysis [3].

Bibliography

1. Fox GC, Williams RD, Messina PC (1994) Parallel computing works!. Morgan Kaufmann, San Francisco. http://www.old-npac.org/copywrite/pcw/node278.html#SECTION 00144000000000000000
2. Fox GC (1988) What have we learnt from using real parallel machines to solve real problems. In: Fox GC (ed) Third conference on hypercube concurrent computers and applications, vol. 2. ACM, New York, pp 897–955
3. Gray J, Hey T, Tansley S, Tolle K (2010) The fourth paradigm: data-intensive scientific discovery. Accessed 21 Oct 2010. Available from: http://research.microsoft.com/en-us/collaboration/fourthparadigm/
4. Ekanayake J, Li H, Zhang B, Gunarathne T, Bae S, Qiu J, Fox G (2010) Twister: a runtime for iterative MapReduce. In: Proceedings of the first international workshop on MapReduce and its applications of ACM HPDC 2010 conference. ACM, Chicago, 20–25 Jun 2010. http://grids.ucs.indiana.edu/ptliupages/publications/hpdc-camera-ready-submission.pdf
5. Yingyi B, Howe B, Balazinska M, Ernst MD (2010) HaLoop: efficient iterative data processing on large clusters. In: The 36th international conference on very large data bases, VLDB Endowment, vol 3, Singapore, 13–17 Sept 2010. http://www.ics.uci.edu/~yingyib/papers/HaLoop_camera_ready.pdf
6. Zhang B, Ruan Y, Tak-Lon W, Qiu J, Hughes A, Fox G (2010) Applying twister to scientific applications. In: CloudCom 2010. IUPUI Conference Center, Indianapolis, 30 Nov–3 Dec 2010. http://grids.ucs.indiana.edu/ptliupages/publications/PID1510523.pdf
7. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
8. Ekanayake J (2010) Architecture and performance of runtime environments for data intensive scalable computing. Ph. D. thesis, School of Informatics and Computing, Indiana University, Bloomington, Dec 2010. http://grids.ucs.indiana.edu/ptliupages/publications/thesis_jaliya_v24.pdf
9. Malewicz G, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, Czajkowski G (2010) Pregel: a system for large-scale graph processing. In: International conference on management of data, Indianapolis, pp 135–146
10. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010) Spark: cluster computing with working sets. In: Second USENIX workshop on hot topics in cloud computing (HotCloud '10), Boston, 22 Jun 2010. http://www.cs.berkeley.edu/~franklin/Papers/hotcloud.pdf
11. Pike R, Dorward S, Griesemer R, Quinlan S (2005) Interpreting the data: parallel analysis with sawzall. Sci Program J (Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure) 13(4):227–298. http://iospress.metapress.com/content/99VJKGKAE3JKVU9T
12. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A (2008) Pig Latin: a not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD international conference on management of data. ACM, Vancouver, pp 1099–1110. http://portal.acm.org/citation.cfm?id=1376726
13. Dongarra J, Foster I, Fox G, Gropp W, Kennedy K, Torczon L, White A (2002) The sourcebook of parallel computing. Morgan Kaufmann, San Francisco. ISBN:978-1558608719
14. Fox GC, Coddington P (2000) Parallel computers and complex systems. In: Bossomaier TRJ, Green DG (eds) Complex systems: from biology to computation. Cambridge University Press, pp 272–287. http://cs.adelaide.edu.au/~paulc/papers/sccs-370b/abs-0370b.html AU5
15. Ekanayake J, Gunarathne T, Qiu J, Fox G, Beason S, Choi JY, Ruan Y, Bae SH, Li H (2010) Applicability of DryadLINQ to scientific applications. Community Grids Laboratory, Indiana University, 30 Jan 2010. http://grids.ucs.indiana.edu/ptliupages/publications/DryadReport.pdf
16. Gustafson JL (1988) Reevaluating Amdahl's law. Commun ACM 31(5):532–533. doi:10.1145/42411.42415
17. Wikipedia (2010) Amdahl's law. Accessed 28 Dec 2010. Available from: http://en.wikipedia.org/wiki/Amdahl's_law

Author Query Form

Encyclopedia of Parallel Computing

Chapter No. 00274

Query Refs.	Details Required	Author's response
AU1	Please provide the section "Synonyms" if applicable.	
AU2	Please check if edit to the sentence starting "For an example. . ." is okay.	
AU3	Please provide text citation for Fig. 2.	
AU4	Please provide the section "Related Entries" if applicable.	
AU5	Please provide the publisher location for the reference Fox and Coddington (2000) if applicable.	