THE LOCALITY PRINCIPLE

Peter J. Denning, Naval Postgraduate School, Monterey, California

January 2008
Rev 6/22/08

Abstract: Locality is a universal behavior of all computational processes: They tend to refer repeatedly to subset of their resources over extended time intervals. System designers have exploited this behavior to optimize performance in numerous ways, which include caching, clustering of related objects, search engines, organizations of databases, spam filters, and forensics.

Keywords: locality, virtual memory, caching, thrashing, working sets, execution phases, phase transitions, program behavior

Locality is a universal behavior of all computational processes: They tend to refer repeatedly to subsets of their resources over extended time intervals. System designers have exploited this behavior to optimize performance in numerous ways, which include caching, clustering of related objects, search engines, organization of databases, spam filters, and forensics.

Every executing computation generates references to objects, such as memory pages, disk sectors, database records, and web pages. These references are not uniform: Some objects are referenced more often than others, and references to each object come in bursts. Another way to say this is that execution of a computation consists of a series of phases; phase i has holding time Ti and locality set Li. The locality set is the set of objects referenced in the phase. A particular object is referenced only in the phases in which it is a member of the locality set. Thus, the history of the computation appears as a sequence,

($L1$,$T2$), ($L2$,$T2$), ($L3$,$T3$), ... , (Li,Ti), ...

The locality set of a multithreaded computation at a particular time is the union of the individual thread locality sets at that time.

Knowledge of a computation's locality behavior has several significant benefits:

- The local storage of a processor (cache) needs to contain only the current localities, not the entire object space. The cache provides significant savings in local storage without loss of performance.
- If the phase boundaries are unknown (the usual case), the best predictor of objects the computation will use in the immediate future is its current locality set.
- Objects that tend to be in the same locality sets can be grouped in storage systems so that they can be loaded together (efficiently) into a processor's cache.

There are two aspects of locality: (1) *temporal locality* means that references to the same objects are grouped in time, and (2) *spatial locality* means that objects close to each other tend to be referenced together. These two aspects give the phase-transition definition above.

Locality is among the oldest systems principles in computer science. It was discovered in 1966 during efforts to make early virtual memory systems work well. Working-set memory management was the first exploitation of this principle; it prevented thrashing while maintaining near optimal system throughput, and eventually it enabled virtual memory systems to be reliable, dependable, and transparent. Today the locality principle is being applied to computations that adapt to the neighborhoods in which users are situated, inferring those neighborhoods by observing user actions, and then optimizing performance for users.

The remainder of this article reviews the history of the locality principle and its new applications in context-aware computing.

Models of Locality

A model of locality is a description of a mechanism to generate the locality behavior of a computation without having to run the computation. The earliest notion of locality was a nonuniform distribution of references over a computation's objects. Thus, the objects could be ordered so that their probabilities of use follow the relation

$p1 > p2 > p3 > ... > pk > ...$

When these probabilities are measured, they often follow a Zipf Law, which means that pk is proportional to $1/k$.

This law is called a *static representation* of locality because a single distribution of probabilities holds for all time; there is no differentiation into phases. Empirically, when a computation is modeled this way, the model overestimates the average locality size by factors of 2 or 3.

In contrast are *dynamic representations* that recognize phases and allow for different probability distributions in each phase. Dynamic models tend to estimate average locality size well.

The phase-transition model is a successful dynamic model. It consists of a *macromodel* that generates phase and transition intervals and their holding times, and a *micromodel* that generates references from a locality set associated with the phase. The holding times in the states are random variables, with the average holding in the phase state being much longer than in the transition state. While in the phase state, the model uses a static representation for a

single locality set, such as the distribution above. During the transition phase, the model allows for random references to all objects. (1)

The working set model defines a program's working set at time t, $W(t,T)$, as the set of objects referenced in the time window of length T extending backward from the current time t. It is usually possible to choose the window size T so that it is contained within phases most of time, in which case the working set measures the current locality set. Thus, the working set is a good way to track the localities and phases of a program dynamically. (2)

History

Locality was discovered from efforts to make virtual memory systems work well. Virtual memory was first developed in 1959 on the Atlas system at the University of Manchester (3). Its superior programming environment doubled or tripled programmer productivity. Its automatic caching boosted performance (4). But its performance was finicky: It was sensitive to the choice of replacement algorithm and to the ways compilers grouped code on to pages. Worse, when it was coupled with multiprogramming, it was prone to thrashing, the near-complete collapse of system throughput because of heavy paging. (5) The locality principle guided the design of robust replacement algorithms, compiler code generators, and thrashing-proof systems. It transformed virtual memory from an unpredictable to a robust technology that regulated itself dynamically and optimized throughput without user intervention.

Atlas included the first demand-paged virtual memory, which automated the process of transferring pages between random access memory (RAM) and disk. The designers grappled with two performance problems, either one of which could scuttle the system: One was translating addresses to locations, the other replacing pages already loaded into RAM. They quickly found a solution to the translation problem using page tables stored in RAM and address caches in the central processing unit (CPU). The replacement problem was much more difficult.

Because the disk access time was about 10,000 times slower than the CPU instruction cycle, each page fault added a significant delay to a job's completion time. Therefore, minimizing page faults was critical to system performance. The ideal page to replace from main memory is the one that will not be used again for the longest time. But next-use time cannot be known with certainty. The Atlas system used a "learning algorithm" that hypothesized a loop cycle for each page, measured each page's period, and estimated which page was not needed for the longest time.

The learning algorithm was controversial. It performed well on programs with well-defined loops and poorly on many other programs. In numerous experimental studies well into the 1960s, researchers sought to determine what replacement rules might work best over the widest possible range of programs. Belady's study in 1966 (6) was the most comprehensive and scientific. Eventually, it became apparent that the volatility resulted from variations in compiling methods: The way in which a compiler grouped code blocks onto pages strongly affected the program's performance under a given replacement strategy (7).

The major computer makers were drawn to multiprogrammed virtual memory because of its superior programming environment. RCA, General Electric, Burroughs, and Univac all included virtual memory in their operating systems of the mid-1960s. Because a bad replacement algorithm could cost a million dollars of lost machine time over the life of a system, they all had a keen interest in finding good replacement algorithms.

Imagine their chagrin when by 1966 these companies reported a new, unexplained, catastrophic problem they called thrashing. It was a sudden collapse of throughput as the multiprogramming level rose. It had nothing to do with the choice of replacement policy. A thrashing system spent most of its time resolving page faults and little running the CPU. Thrashing was far more damaging than a poor replacement algorithm. It scared the daylights out of the computer makers.

IBM avoided these uncertainties by excluding virtual memory from its OS360 in 1964. Instead, it sponsored at its Watson laboratory one of the most comprehensive experimental systems projects of all time. Led by Bob Nelson, Les Belady, and David Sayre, the project team built the first virtual-machine operating system and used it to study the performance of virtual memory. (The term "virtual memory" seems to have come from this project.) By 1966 they had tested every replacement policy that anyone had ever proposed and a few more they invented. Many of their tests involved the use bits built into page tables. By periodically scanning and resetting the bits, the replacement algorithm distinguishes recently referenced pages from others. Belady concluded that policies that favor recently used pages performed better than other policies; least recently used (LRU) replacement was consistently the best performer among those tested (6). He said that this resulted from reference clustering locality behavior. His colleagues verified that many programs exhibited locality behavior (8).

At MIT Project MAC in 1966, Peter Denning hypothesized that the controversies over replacement algorithms could be settled by defining an intrinsic memory demand: "This process needs *p* pages at time *t*." Intrinsic demand was the first notion of "working set". Individual replacement algorithms could then be rated by their abilities to detect working sets. Inspired by Belady's concept of locality, Denning formally defined a process's working set as the set of pages used during a fixed-length sampling window in the immediate past (2). A working set could be measured by periodically reading and resetting the use bits in a page table.

This method solved thrashing because if every process is guaranteed its working set, then no process can overload the paging disk and system throughput can be maintained (5). Thrashing is impossible for a working-set policy. Experiments with real operating systems confirmed that this policy gives high efficiency and prevents thrashing (9).

The working-set idea worked because the pages observed in the backward window were highly likely to be used again in the immediate future. Was this assumption justified? The idea that reference behavior could be described as a sequence of phases and locality sets seemed natural because programmers planned overlays using diagrams that showed subsets and time phases (Fig. 1). But what was strikingly interesting was that programs showed this behavior even when it was not explicitly preplanned. Each program had its own distinctive usage pattern, like a voiceprint (Fig. 2).

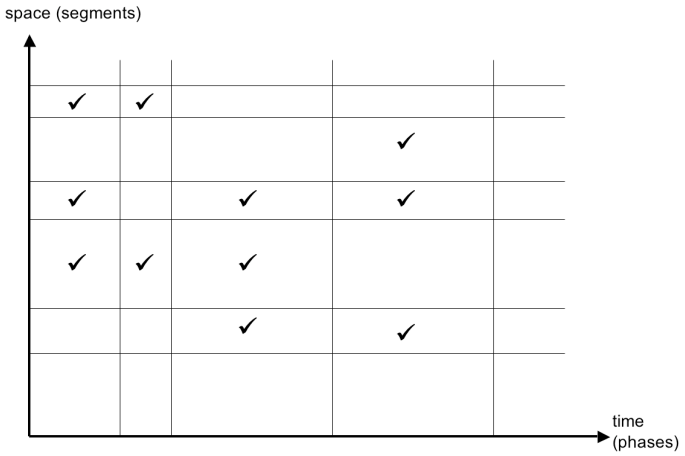

Figure 1. Locality sequence behavior diagrammed by programmer during overlay planning.

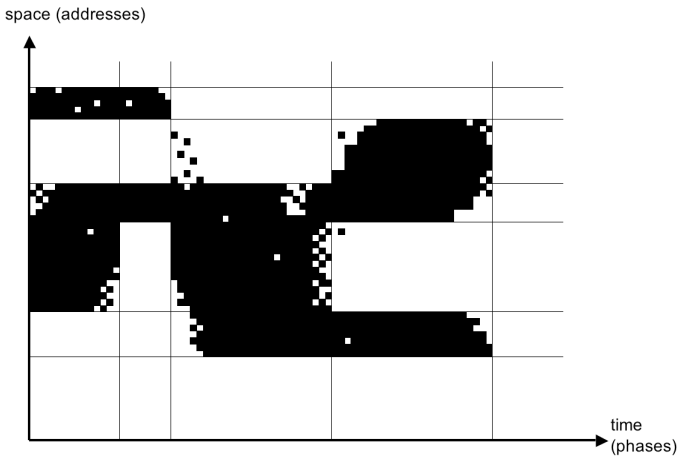

Figure 2. Locality sequence behavior observed by sampling use bits during program execution. Programs exhibit phases and localities naturally, even when overlays are not preplanned.

Two effects could make this happen: (1) temporal clustering caused by looping and executing within modules with private data, and (2) spatial clustering caused by related values being grouped into arrays, sequences, modules, and other data structures. Both these reasons followed from the human practice of "divide and conquer" -- breaking a large problem into parts and working separately on each part. The locality bit maps captured someone's problem-solving method in action. The working set measures an approximation of a program's intrinsic locality sequence.

A distance function gives a single measure of temporal and spatial locality. $D(x,t)$ measures the distance from the execution point of a process to an object x at time t. Distances can be temporal, such as the time since prior reference or access time within a network; spatial, measuring hops in a network or address separation in a sequence; or cost, which measures any nondecreasing accumulation of cost since prior reference. Object x is in the locality set at time t if the distance is withins a threshold: $D(x,t) \leq T$.

By 1980 the locality principle was understood as a package of three ideas (1):

1. Computational processes pass through a sequence of phases.
2. The locality sets of phases can be inferred by applying a distance function to a program's address trace observed during a backward window.
3. Memory management is optimal when it guarantees each program that its locality sets will be present in high-speed memory.

Adoption of Locality Principle

The locality principle was adopted almost immediately by operating systems, databases, and hardware architects. It was soon adopted into ever-widening circles:

- In virtual memory to organize caches for address translation and to design the replacement algorithms
- In data caches for CPUs, originally as mainframes and now as microchips
- In buffers between main memory and secondary memory devices
- In buffers between computers and networks
- In video boards to accelerate graphics displays
- In modules that implement the information-hiding principle
- In accounting and event logs in that monitor activities within a system
- In alias lists that associate longer names or addresses with short nicknames
- In the "most recently used" object lists of applications
- In Web browsers to hold recent web pages
- In file systems, to organize indexes (e.g., B-trees) for fastest retrieval of file blocks
- In database systems, to manage record-flows between levels of memory
- In search engines to find the most relevant responses to queries
- In classification systems that cluster related data elements into similarity classes
- In spam filters, which infer which categories of email are in the user's locality space and which are not
- In "spread spectrum" video streaming that bypasses network congestion and reduces the apparent distance to the video server
- In "edge servers" to hold recent web pages accessed by anyone in an organization or geographical region
- In the field of computer forensics to infer criminal motives and intent by correlating event records in many caches

- In the field of network science by defining hierarchies of self-similar locality structures within complex power-law networks

Modern Model of Locality: Context Awareness

As the uses of locality expanded into more areas, our understanding of locality has evolved. Locality is the consequence of a more basic principle: Everything we do, we do in a context. Context awareness embraces four key ideas:

- **An observer**
- **Neighborhoods:** One or more sets of objects that are most relevant to the observer at any given time
- **Inference:** A method of identifying the most relevant objects by monitoring the observer's actions and interactions and other information about the observer contained in the environment
- **Optimal actions:** An expectation that the observer will complete work in the shortest time if neighborhood objects are ready accessible in nearby caches

These four ideas can be recognized in the original definition. The observer is the execution point of the computational process; the neighborhood is the current locality set; the distance function is the inference mechanism; the optimal action is to guarantee that the current locality is present in a processor's cache. Let us examine the generalizations of these ideas.

The observer is the agent who is trying to accomplish tasks with the help of software, and who places expectations on its function and performance (Fig. 3). In most cases, the observer is the user who interacts with software. In some cases, such as a program that computes a mathematical model, the observer can be built into the software itself.

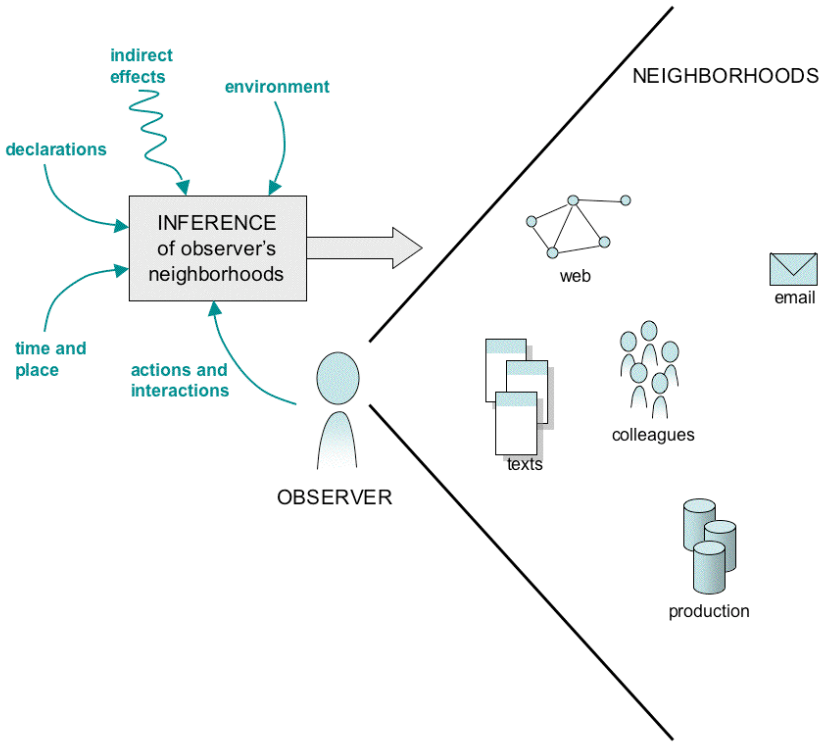

Figure 3. The modern view of locality is a means of inferring the context of an observer using software, so that the software can dynamically adapt its actions to produce optimal behavior for the observer.

A neighborhood is a group of objects related to the observer by some metric. Newer examples of neighborhoods include email correspondents, non-spam email, colleagues, teammates, objects used in a project, favorite objects, user's web, items of production, texts, and directories. Some neighborhoods can be known by explicit declarations; for example a user's file directory, address book, or web pages. But most neighborhoods can only be inferred by monitoring the event sequences of an observer's actions and interactions.

Inference can be any reasonable method that measures the content of neighborhoods. Newer inference methods include Google's counting of incoming hyperlinks to a web page, patterns generated by connectionist networks after being presented with many examples, and Bayesian spam filters.

Optimal actions are performed by the software on behalf of the observer. These actions can come either from inside the software with which the observer is interacting, or from outside that software, in the run-time system.

The matrix below shows four quadrants that correspond to the four combinations of inference data collection and locus of action just mentioned. Examples of software are named in each quadrant and are summarized below. "Inside" and "outside" are relative to the context-aware software.

		ORIGIN OF DATA FOR INFERENCE	
		Inside	**Outside**
LOCUS OF ADAPTIVE ACTION	**Inside**	Amazon.com, Bayesian spam filter	Semantic web Google
	Outside	Linkers and loaders	Working sets, Ethernet load control

- **Amazon.com, Bayesian spam filters.** Amazon collects data about user purchasing histories and recommends other purchases, by the user or others, that resemble the user's previous purchases. Bayesian spam filters gather data about which emails the user considers relevant and then block irrelevant emails. (Data collection inside, optimal actions inside.)
- **Semantic web, Google.** Semantic web is a set of declarations of structural relationships that constitute context of objects and their connections. Programs read and act on it. Google gathers data from the Web and uses it to rank pages that seem to be most relevant to a keyword query posed by user. (Data collection outside, optimal actions inside.)
- **Linkers and loaders.** These workhorse systems gather library modules mentioned by a source program and link them together into a self-contained executable module. The libraries are neighborhoods of the source program. (Data collection inside, optimal action outside.)
- **Working sets, Ethernet load controls.** Virtual memory systems measure working sets and guarantee programs enough space to contain them, which thereby prevents thrashing. Ethernet prevents the contention-resolving protocol from getting overloaded by making competing transactions wait longer for retries if load is heavy (10). (Data collection outside, optimal action outside.)

In summary, the modern principle of locality is that observers operate in one or more neighborhoods that can be inferred from dynamic action sequences and static structural declarations. Systems can optimize the observer's productivity by adapting to the observer's neighborhoods, which they can estimate by distance metrics or other inferences.

Future Uses of Locality Principle

Locality principles are certain to remain at the forefront of systems design, analysis, and performance, because locality flows from human cognitive and coordinative behavior. The mind focuses on a small part of the sensory field and can work most quickly on the objects of its attention. People organize their social and intellectual systems into neighborhoods of related objects, and they gather the most useful objects of each neighborhood close around them to minimize the time and work of using them. These behaviors are transferred into computational systems they design and into the expectations users have about how their systems should interact with them.

Here are seven modern areas that offer challenging research problems that locality may be instrumental in solving.

Architecture

Computer architects have heavily exploited the locality principle to boost the performance of chips and systems. Putting cache memory near the CPU, either on board the same chip or on a neighboring chip, has enabled modern CPUs to pass the 1 GHz speed mark. Locality within threaded instruction sequences is being exploited by a new generation of multicore processor chips. The "system on a chip" concept places neighboring functions on the same chip to decrease delays of communicating between components significantly. Animated sequences of pictures can be compressed by locality: by detecting the common neighborhood behind a sequence, transmitting it once, and then transmitting the differences for each picture. Architects will continue to examine locality carefully to find new ways to speed up chips, communications, and systems.

Caching

The locality principle is useful wherever there is an advantage in reducing the apparent distance from a process to the objects it can access. Objects in the neighborhood of the process are kept in a local cache with fast access time. The performance acceleration of a cache generally justifies the modest investment in the cache storage. Novel forms of caching have sprung up in the Internet. One prominent example is edge servers that store copies of Web objects near their users. Another example is the clustered databases built by search engines (like Google) to retrieve relevant objects instantly from the same neighborhoods as the asker. Similar capabilities are available in MacOS Windows to speed up finding relevant objects.

Bayesian Inference

A growing number of inference systems exploit Bayes's principle of conditional probability to compute the most likely internal (hidden) states of a system given observable data about the system. Spam filters, for example, use it to infer the email user's mental rules for classifying certain objects as spam. Connectionist networks use it for learning: Their internal states abstract from desired input-output pairs shown to the network; the network gradually acquires a capability for new action. Bayesian inference is an exploitation of locality because it infers a neighborhood given observations of what a user or process is doing.

Forensics

The burgeoning field of computer forensics owes much of its success to the ubiquity of caches. They are literally everywhere in an operating systems and applications. By recovering evidence from these caches, forensics experts can reconstruct (infer) an amazing amount of a criminal's motives and intent (11). Criminals who erase data files are still not safe, because experts use advanced signal-processing methods to recover the faint magnetic traces of the most recent files from the disk (12). Learning to draw valid inferences from data in

a computer's caches, and from correlated data in caches in other computers with which the subject has communicated, is a challenging research problem.

Web-Based Business Processes

Web-based business systems allow buyers and sellers to engage in transactions using web interfaces to sophisticated database systems. Amazon.com illustrates how a system can infer "book interest neighborhoods" of customers and (successfully) recommend additional sales. Many businesses employ customer relationship management systems that infer "customer interest neighborhoods" and allow the company to provide better, more personalized service. Database, network, server, memory, and other caches optimize the performance of these systems (13).

Context Aware Software

More software designers are coming to believe that most software failures can be traced to the inability of software to be aware of and act on the context in which it operates. More and more modern software uses inferred context to be consistently more reliable, dependable, usable, safe, and secure.

Network Science

Many scientists have begun to apply statistical mechanics to large random networks, typically finding that the distribution of node connections is power law with degree -2 to -3 (14,15). These networks are self-similar, which means that if all neighborhoods (nodes within a maximum distance of each other) are collapsed to single nodes, then the resulting network has the same power distribution as the original (16). The idea that localities are natural in complex systems is not new; in 1976, Madison and Batson (17) reported that program localities have self-similar sub-localities; and in 1977, P. J. Courtois (18) applied it to cluster similar states of complex systems to simplify their performance analyses. The locality principle may offer new understandings of the structure of complex networks.

Researchers looking for challenging problems can find many in these areas and can exploit the principle of locality to solve them.

BIBLIOGRAPHY

1. P. J. Denning, Working sets past and present, *IEEE Trans. Softw. Eng.* **SE-6**(1): 64-84, 1980.
2. P. J. Denning, The working set model for program behavior, *ACM Commun.* **11**(5), 323-333, 1968.
3. T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, One-level storage system, *IRE Trans.* **EC-11**: 223-235, 1962.
4. M. V. Wilkes, Slave memories and dynamic storage allocation, *IEEE Trans. Comp.* **EC-14**(4): 270-271, 1965.
5. P. J. Denning, Thrashing: Its causes and prevention. *Proc. AFIPS Fall Joint Computer Conference 33*. Thompson, 1968, pp. 915-922.

6. L. A. Belady, A study of replacement algorithms for virtual storage computers, *IBM Systems J.* **5(**2): 78-101, 1966.
7. D. Ferrari, Improving locality by critical working sets. *ACM Commun.* **17** (11): 614-620, 1974.
8. B. Brawn and F. G. Gustavson, Program behavior in a paging environment, *Proc. AFIPS Fall Joint Computer Conference 33*. Thompson, 1968, pp. 1019-1032.
9. J. Rodriguez-Rosell and J. P. Dupuy, The design, implementation, and evaluation of a working set dispatcher, *ACM Commun.* **16**(4): 247-253, 1973.
10. R. M. Metcalfe and D. Boggs, Ethernet: Distributed packet switching for local networks, *ACM Commun.* **19**(7): 395-404, 1976.
11. D. Farmer and W. Venema, *Forensic Discovery,* Reading, MA: Addison Wesley, 2004.
12. B. Carrier, *File System Forensic Analysis,* Reading, MA: Addison Wesley, 2005.
13. D. Menasce and V. Almeida, *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning.* Englewood Cliffs, NJ: Prentice-Hall, 2000.
14. A. L. Barabasi, *Linked: The New Science of Networks,* Perseus Books, 2002.
15. P. J. Denning, Network Laws, *ACM Commun.* **47** (11): 15-20, 2004.
16. C. Song, S. Havlin, and H. Makse, Self-Similarity of Complex Networks, *Nature* **433** (1): 392-395, 2005.
17. A. W. Madison and A. Batson, Characteristics of program localities, *ACM Commun.* **19**(5): 285-294, 1976.
18. P. J. Courtois, *Decomposability.* New York: Academic Press, 1977.