IX. Parallel Algorithms Design (1)

April 13th, 2009

Content

- Concurrency in parallel programs,
- Approaches to achieve concurrency,
- Basic layers of software concurrency;
- Tasks, processes and processors,
- Design steps
- Decomposition
  - simple examples
  - classification

Exploiting concurrency

- The key to parallel computing is exploitable concurrency.
- Concurrency exists in a computational problem when the problem can be decomposed into subproblems that can safely execute at the same time.
- The concurrency must be exploitable:
  - Structure the code to expose and later exploit the concurrency and permit the subproblems to actually run concurrently;
  - Most large computational problems contain exploitable concurrency.
  - A programmer works with exploitable concurrency by creating a parallel alg.& implementing the alg. using a parallel programming environment.
- Example, suppose part of a computation involves computing the summation of a large set of values.
  - If multiple processors are available, instead of adding the values together sequentially, the set can be partitioned and the summations of the subsets computed simultaneously, each on a different processor.
  - Using multiple procs to compute in parallel => solution sooner.
  - If each proc.has its own memory, partitioning the data between procs may allow larger probls to be handled than could be handled on a single proc.

Problems when applying in par.comp.

- Often, the concurrent tasks making up the problem include dependencies that must be identified and correctly managed.
  - The order in which the tasks execute may change the answers of the computations in nondeterministic ways.
  - Example: in the parallel summation described earlier, a partial sum cannot be combined with others until its own computation has completed.
    - The algorithm imposes a partial order on the tasks (that is, they must complete before the sums can be combined).
    - The numerical value of the summations may change slightly depending on the order of the operations within the sums because floating point arithmetic is non associative.
  - A good parallel programmer must take care to ensure that nondeterministic issues such as these do not affect the quality of the final answer.
    - Creating safe parallel programs can take considerable effort from the programmer.
- Even when a parallel program is "correct", it may fail to deliver the anticipated performance improvement from exploiting concurrency.
  - Care must be taken to ensure that the overhead incurred by managing the concurrency does not overwhelm the program runtime.
  - Partitioning the work among the processors in a balanced way is often not as easy as the summation example suggests.
- The effectiveness of a parallel algorithm depends on how well it maps onto the underlying parallel computer,
  - a parallel alg could be very effective on one parallel arch. & a disaster on another.

Concurrency in Parallel Programs vs Operating Systems

- Modern OSs use multiple procs to increase the throughput of the syst
- The fundamental concepts for safely handling concurrency are the same in parallel programs and operating systems
- Important differences.
    - Why finding concurrency
        - For an operating system, the problem is not finding concurrency
            - the concurrency is inherent in the way the OS functions in managing a collection of concurrently executing processes & providing synchronization mechanisms so resources can be safely shared
        - An OS must support concurrency in a robust and secure way
            - Processes should not be able to interfere with each other,
            - The entire system should not crash if something goes wrong with one process.
        - In a parallel program, finding and exploiting concurrency is a challenge
            - isolating processes from each other is not the critical concern it is with an OS
    - Performance goals are different as well.
        - In an OS, is related to throughput or response time & it may be acceptable to sacrifice efficiency to maintain robustness&fairness in resource allocation
        - In a parallel program, is to minimize the running time of a single program.

What is Concurrency

- Two events are concurrent if they occur within the same time interval.
- Two or more tasks executing over the same time interval are execute concurrently.
- Example 1: Two tasks may occur concurrently within the same second but with each task executing within different fractions of the second.
  - The first task may execute for the first tenth of the second and pause,
  - The second task may execute for the next tenth of the second and pause, the first task may start again executing in the third tenth of a second, and so on.
  - Each task may alternate executing.
  - Length of a sec is so short that it appears both tasks are executing simultaneously.
- Example 2: Two programs performing some task within the same hour continuously make progress of the task during that hour, although they may or may not be executing at the same exact instant.
  - The two programs are executing concurrently for that hour.
- Concurrent tasks can execute in a single or multiprocessing environment.
  - In a single processing environment, concurrent tasks exist at the same time and execute within the same time period by context switching.
  - In a multiprocessor environment, if enough processors are free, concurrent tasks may execute at the same instant over the same time period.
- The determining factor for what makes an acceptable time period for concurrency is relative to the application.

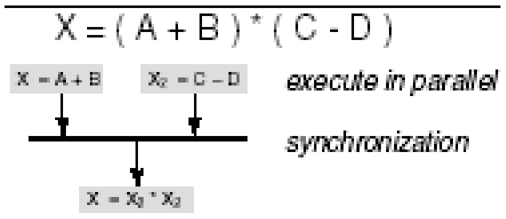
Why concurrency?

- Used to allow a program to do more work over the same time period or time interval.
  - The program is broken down in such a way that some of the tasks can be executed concurrently
- Simplifying the programming solution
  - Sometimes it makes more sense to think of the solution to the problem as a set of concurrently executed tasks.
  - For instance, the solution to the problem of losing weight is best thought of as concurrently executed tasks: diet and exercise.
- The concurrency of both processes is the natural form of the solution.
- Sometimes concurrency is used to make software faster or get done with its work sooner
- Sometimes concurrency is used to make software do more work over the same interval where speed is secondary to capacity.
  - For instance, some web sites want customers to stay logged on as long as possible.
  - So it's not how fast they can get the customers on and off of the site that is the concern—it's how many customers the site can support concurrently.
  - So the goal of the software design is to handle as many connections as possible for as long a time period as possible
- Concurrency can be used to make the software simpler.
  - Often, one long, complicated sequence of operations can be implemented easier as a series of small, concurrently executing operations.
- Whether concurrency is used to make the software faster, handle larger loads, or simplify the programming solution, the main object is software improvement using concurrency to make the software better.

Two Basic Approaches to Achieving Concurrency

- Parallel programming and distributed programming
  - They are two different programming paradigms that sometimes intersect.
- Parallel programming techniques assign the work a program has to do to two or more processors within a single physical or a single virtual computer
  - A program that contains parallelism executes on the same physical or virtual computer.
  - The parallelism within a program may be divided into processes or threads.
  - Multithreading is restricted to parallelism.
- Distributed programming techniques assign the work a program has to do to two or more processes— where the processes may or may not exist on the same computer.
  - Parts of a distr.prog often run on different comps or at least in different processes.
  - Distributed programs can only be divided into processes.
- Parallel programs are sometimes distributed, as is the case with PVM programming.
- Distributed programming is sometimes used to implement parallelism, as is case of MPI
  - However, not all distributed programs involve parallelism.
  - The parts of a distrib.progr may execute at different instances&over different time periods
  - For instance, a software calendar program might be divided into two parts:
    - One part provides the user with a calendar &a method for recording important appointments
    - The other part provides the user with a set of alarms for each different type of appointment.
    - The user schedules the appointments using part of the software, and the other part of the software executes separately at a different time.
    - The alarms and the scheduling component together make a single application, but they are divided into two separately executing parts.
- In parallelism, the concurrently executing parts are all components of the same progr.
- In distributed programs, the parts are usually implemented as separate programs.

The Basic Layers of Software Concurrency



1. Instruction level
   - when multiple parts of a single instruction can be executed simultaneously.
   - Figure shows how a single instruction can be decomposed for simultaneous execution
     - the component (A + B) can be executed at the same time as (C – D).
   - This kind of parallelism is normally supported by compiler directives and is not under the direct control of a C++ programmer.
2. Routine (function/procedure) level
   - The structure of a program may be distributed along function lines:
     - The total work involved in a software solution is divided between a number of functions.
   - If these functions are assigned to threads, then each function can execute on a different processor and
   - If enough processors are available, each function can execute simultaneously.
3. Object level
4. Application level

The Basic Layers of Software Concurrency

1. Instruction level
2. Routine (function/procedure) level
    - The structure of a program may be distributed along function lines:
        - The total work involved in a software solution is divided between a number of functions.
    - If these functions are assigned to threads, then each function can execute on a different processor and
    - If enough processors are available, each function can execute simultaneously.
3. Object level
4. Application level

The Basic Layers of Software Concurrency

1. Instruction level
2. Routine (function/procedure) level
3. Object level
   - The idea is to have a distribution between objects.
   - Each object can be assigned to a different thread, or process.
   - Using the CORBA standard, each object may be assigned to a different computer on the network or different computer on a different network.
   - Objects residing in different threads or processes may execute their methods concurrently.
4. Application level
   - Two or more applications can cooperatively work together to solve some problem.
   - Although the application may have originally been designed separately and for different purposes, the principles of code reuse often allow applications to cooperate.
   - In these circumstances two separate applications work together as a single distributed application.

Challenges in the design process

- There are 3 basic challenges to writing parallel or distributed progr:
  1. Identifying the natural parallelism that occurs within the context of a problem domain.
  2. Dividing the software appropriately into two or more tasks that can be performed at the same time to accomplish the required parallelism.
  3. Coordinating those tasks so that the software correctly and efficiently does what it is supposed to do.
- The design process include three issues (DCS):
  (a) D: decomposition,
  (b) C: communication, and
  (c) S: synchronization

- Obstacles to concurrency:
  - Data race
  - Deadlock detection
  - Partial failure
  - Latency
  - Deadlock
  - Communication failures
  - Termination detection
  - Lack of global state
  - Multiple clock problem
  - Protocol mismatch
  - Localized errors
  - Lack of centralized resource allocation

Decomposition

- Is the process of dividing up the problem and the solution into parts.
- The parts are grouped
  - into logical areas (i.e., searching, sorting, calculating, input, output, etc.).
  - by logical resource (i.e., file, communication, printer, database, etc.).
- WBS (work breakdown structure) determines which piece of software does what.
  - Primary issue of concurrent programming: identify a natural WBS for the solution at hand.
  - There is no simple or cookbook approach to identifying the WBS.
  - The process of modeling uncovers the WBS of a software solution.
    - The better the model is understood & developed the more natural WBS will be
- The question of how to break up an application into concurrently executing parts should be answered during the design phase and should be obvious in the model of the solution.
  - If the model of the problem and the solution don't imply or suggest parallelism and distribution then try a sequential solution.
  - If the sequential solution fails, that failure may give clues to how to approach the parallelism.

Communication & Synchronization

- Once the software solution is decomposed into a number of concurrently executing parts, those parts will usually do some amount of **communicating.**
  - Issues have to be considered when designing parallel or distributed systems:
    - How will this communication be performed if the parts are in different processes or different computers?
    - Do the different parts need to share any memory?
    - How will one part of the software know when the other part is done?
    - Which part starts first?
    - How will one component know if another component has failed?
- Components of software are working on the same probl => they must be **syncr.**
  - Some component has to determine when a solution has been reached.
  - The components' order of execution must be coordinated.
  - Issues:
    - Do all of the parts start at the same time or does some work while others wait?
    - What two or more components need access to the same resource?
    - Who gets it first?
    - If some of the parts finish their work long before others, should they be assigned new work?
    - Who assigns the new work in such cases?
- DCS (decomposition, communication, and synchronization) is the minimum that must be considered when approaching parallel or distributed programming
  - In addition to considering DCS, the location of DCS is also important.
  - There are several layers of concurrency in application development: DCS is applied a little differently in each layer.

Task, Processes and Processors

- A *task* is an arbitrarily defined piece of the work done by the program.
  - It is the smallest unit of concurrency that the parallel program can exploit;
    - an individual task is executed by only one processor, and concurrency is exploited across tasks.
  - Example: in Raytrace a ray or a group of rays, and in Data Mining it may be checking a single transaction for the occurrence of an itemset.
  - What exactly constitutes a task is not prescribed by the underlying sequential program; it is a choice of the parallelizing agent, though it usually matches some natural granularity of work in the sequential program structure.
  - If the amount of work a task performs is small, it is called a *fine-grained* task; otherwise, it is called *coarse-grained*.
- A *process* (referred to interchangeably hereafter as a *thread*) is an abstract entity that performs tasks.
  - A parallel program is composed of multiple cooperating processes, each of which performs a subset of the tasks in the program.
  - Tasks are assigned to processes by some *assignment* mechanism.
  - Example: in data mining, the assignment may be determined by which portions of the database are assigned to each process, and by how the itemsets within a candidate list are assigned to processes to look up the database.
  - Processes may need to communicate and synchronize with one another to perform their assigned tasks.
- The way processes perform their assigned tasks is by executing them on the physical *processors* in the machine.

Processes vs. processors

- Processors are physical resources
- Processes provide a convenient way of abstracting or *virtualizing* a multiprocessor.
- Write parallel programs in terms of processes not physical processors;
- Mapping processes to processors is a subsequent step.
- The number of processes does not *have* to be the same as the number of processors available to the program.
  - If there are more processes, they are multiplexed onto the available processors;
  - If there are fewer processes, then some processors will remain idle.
- The correct OS definition of a process is that of an address space and one or more threads of control that share that address space.
  - Thus, processes and threads are distinguished in that definition.
  - In what follows: assume that a process has only one thread of control.

The Four Steps and Their Goals

The job of creating a parallel program from a sequential one consists of 4 steps:

1. **Decomposition** of the computation into tasks,
2. **Assignment** of tasks to processes,
3. **Orchestration** of the data access, communic.&synchronization among processes,
4. **Mapping** or binding of processes to processors.

Decomposition + assignment = **partitioning**

- since they divide the work done by the program among the cooperating processes.

Step	**Arch.dependent?**	**Major Performance Goals**
Decomposition	Mostly no	Expose enough concurrency, but not too much
Assignment	Mostly no	Balance workload
		Reduce communication volume
Orchestration	Yes	Reduce non-inherent communic. via data locality
		Reduce cost of comm/synch as seen by processor
		Reduce serialization of shared resources
		Schedule tasks to satisfy dependences early
Mapping	Yes	Put related processes on the same processor if necessary
		Exploit locality in network topology

Decomposition

- Means breaking up the computation into a collection of tasks.
- For example, tracing a single ray in Raytrace may be a task
- Tasks may become available dynamically as the program executes,
  - no.tasks available at a time may vary over the execution of the program
- The max.no. tasks available at a time provides an upper bound on the no.processes (&processors) that can be used effectively at that time.
- The major goal in decomposition is to *expose enough concurrency* to keep the processes busy at all times,
  - yet not so much that the overhead of managing the tasks becomes substantial compared to the useful work done.
- Limited concurrency is the most fundamental limitation on the speedup achievable through parallelism
  - Not just the fundamental concurrency in the underlying problem but also how much of this concurrency is exposed in the decomposition.
  - The impact of available concurrency is codified in the *Amdahl's Law*
- Concurrency profile:
  - given a decomposition and a problem size, a *concurrency profile* depicts how many operations (or tasks) are available to be performed concurrently in the application at a given time.
  - Is a function of the problem, the decomposition and the problem size
  - Is independent of the number of processors.
  - It is also independent of the assignment or orchestration.
  - May be easy to analyze or they may be quite irregular.

Decomposition - example

- Consider a simple example program with two phases.
  - 1st phase: an operation is performed independently on all points of a 2-d n –by n grid.
  - 2nd phase, the sum of the n^2 grid point values is computed.
- Solution 1:
  - If we have p procs, we can assign n^2/p points to each proc
    - Complete the first phase in parallel in time n^2/p.
  - In 2nd phase, each processor can add each of its assigned vals into a global sum var.
  - What is the problem with this assignment, and how can we expose more concurrency?
  - The problem is that the accumulations into the global sum must be done one at a time, or *serialized*, to avoid corrupting the sum value by having 2 procs try to modify it simultaneously.
  - Thus, the second phase is effectively serial and takes n^2 time regardless of p . T
  - The total time is $n^2/p+ n^2$, compared to a sequential time of $2n^2$, so S is at best 2 even if p increase
- Solution 2:
  - Instead of summing each value directly into the global sum, serializing all the summing, we divide the second phase into two phases.
    - In the new 2nd phase (fully parallel), a process sums its assigned values independently into a private sum.
    - Then, in the third phase (serialized), processes sum their private sums into the global sum.
  - The total parallel time is $n^2/p+ n^2/p+p$, and the speedup is improved.
  - The speedup limit is almost linear in the number of processors used.

Assignment

- Specify the mechanism by which tasks will be distributed among processes
- For example, which process will count occurrences of which item sets and in which parts of the database in Data Mining
- Primary performance goals are:
  - to *balance the workload* among processes, referred to as *load balancing*
    - the workload includes: computation, input/output & data access or communication,
  - to *reduce interprocess communication*, and
  - to *reduce the runtime overheads of managing the assignment*.
- Programs are often structured in phases, and candidate tasks for decomposition within a phase are often easily identified as seen in the case studies.
  - The appropriate assignment of tasks is often discernible either by inspection of the code or from a higher-level understanding of the application.
  - And where this is not so, well-known heuristic techniques are often applicable.
- Classification:
  - *static or predetermined assignment*;
    - If the assignment is completely determined at the beginning of the program—or just after reading and analyzing the input—and does not change thereafter
  - *dynamic assignment*.
    - if the assignment of work to processes is determined at runtime as the program executes—perhaps to react to load imbalances

Orchestration

- The arch.&progr.model play a large role, as well as the programming language itself.
- Orchestration uses the available mechanisms to accomplish the goals:
  - To execute their assigned tasks, processes need mechanisms to name and access data, to exchange data with other processes, and to synchronize with one another.
- The choices made in orchestration are much more dependent on
  - the programming model, and
  - the efficiencies with which the primitives of the progr.model & communic.abstraction are supported,

  than the choices made in the previous steps.
- Some questions in orchestration include:
  - how to organize data structures and schedule tasks to exploit locality,
  - whether to communicate implicitly or explicitly and in small or large messages, and
  - how exactly to organize and express interprocess communication and synchronization.
- Orchestration also includes scheduling the tasks assigned to a process temporally, i.e. deciding the order in which they are executed.
- The programming language is important because
  - this is the step in which the program is actually written and
  - some of the above tradeoffs in orchestration are influenced by available lang. mechanisms & costs
- The major performance goals in orchestration are:
  - *reducing the cost of the communication and synchronization* as seen by the processors,
  - *preserving locality of data reference*,
  - *scheduling tasks* so that those on which many other tasks depend are completed early, and
  - *reducing the overheads of parallelism management*.
- The job of architects: provide appropriate primitives that simplify orchestration.

Mapping

- The program may control the mapping of processes to processors, but if not the operating system will take care of it, providing a parallel execution.
- Mapping tends to be fairly specific to the system or programming environment.
- *Space sharing*
  - the processors in the machine are partitioned into fixed subsets, possibly the entire machine, and only a single program runs at a time in a subset.
  - The program
    - can bind or *pin* processes to processors to ensure that they do not migrate during the execution, or
    - can control which proc.a process runs on so as to preserve locality of communication in the netw topology
  - Strict spaces-sharing schemes, together with some simple mechanisms for time-sharing a subset among multiple applications, have so far been typical of large-scale multiprocessors.
- At the other extreme: the OS may dynamically control which process runs where and when— without allowing the user any control over the mapping—to achieve better resource sharing and utilization.
  - Each processor may use the usual multi-programmed scheduling criteria to manage processes from the same or from different programs, and processes may be moved around among processors as the scheduler dictates.
  - The operating system may extend the uniprocessor scheduling criteria to include multiprocessor-specific issues.
- In fact, most modern systems fall somewhere between the above two extremes:
  - The user may ask the system to preserve certain properties,
  - giving the user program some control over the mapping,
  - the OS is allowed to change the mapping dynamically for effective resource management.

Parallelizing Computation versus Data

- *Computation- centric* view:
  - The view of the parallelization process is usually centered on computation, or work, rather than on data.
  - It is the computation that is decomposed and assigned.
  - due to the communication abstraction or performance considerations, we may be responsible for decomposing and assigning data to processes as well.
- In many important classes of probls the decomposition of work & data are so strongly related that they are difficult or even unnecessary to distinguish.
  - Example: data mining,
    - where we can view the database as being decomposed and assigned.
    - the question of assigning the itemsets to processes
    - The process that is assigned a portion of data will then be responsible for the computation associated with that portion, a so-called *owner computes* arrangement.
  - Several language systs, including the High Performance Fortran standard**,** allow the programmer to specify the decomposition and assignment of data structures;
    - the assignment of computation then follows the assignment of data in an owner computes manner.
- The distinction between computation and data is stronger in many other applications
  - Example: Raytrace case

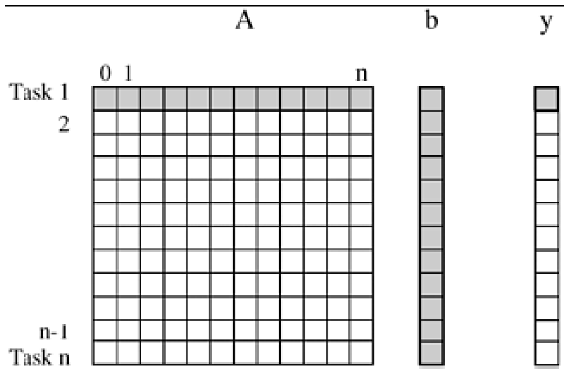
Tasks in decomposition phase

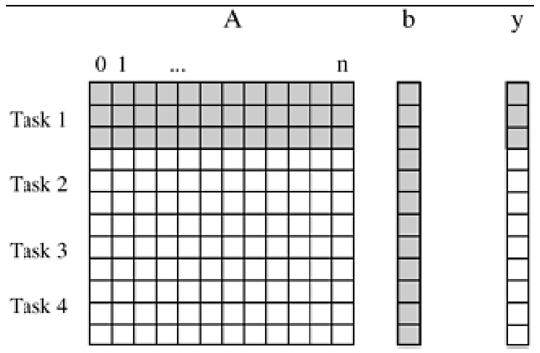
- Decomposition:
  - The process of dividing a computation into smaller parts, some or all of which may potentially be executed in parallel,
- Tasks are programmer-defined units of computation into which the main computation is subdivided by means of decomposition.
- Simultaneous execution of multiple tasks is the key to reducing the time required to solve the entire problem.
- Tasks can be of arbitrary size, but once defined, they are regarded as indivisible units of computation.
- The tasks into which a problem is decomposed may not all be of the same size.
- Examples that follows:

1. Dense matrix-vector multiplication
2. Sparse matrix-vector multiplication

Ex 1: Dense matrix-vector multiplication - decomposition

- Consider the multiplication of a dense *n* x *n* matrix *A* with a vector *b* to yield another vector *y*.
  - The *i*th element y[i] of the product vector is the dot-product of the *i*th row of *A* with the input vector *b*; i.e. *y*[*i*]=sum(*A*[*i*,*j*]b[*j*],*j*=1..*n*).
  - The computation of each *y*[*i*] can be regarded as a task.
    - all tasks are independent & can be performed all together or in any sequence.
  - Alternatively:
    - the computation could be decomposed into fewer, say 4, tasks where each task computes roughly *n*/4 of the entries of vector y.





Granularity and task interaction

- The number and size of tasks into which a problem is decomposed determines the granularity of the decomposition.
  - A decomposition into a large number of small tasks is called *finegrained*
  - A decomposition into a small number of large tasks is called *coarse-grained*.
- Example:
  - First decomposition for matrix-vector multiplication is fine-grained
  - Second dec.is a coarse-grained decomposition of the same problem into four tasks
- *Maximum degree of concurrency*:
  - Max.no. tasks executed simultaneously in a parallel progr. at any given time.
  - It is usually less than the total no. tasks due to dependencies among the tasks.
  - Ex.: for task-dependency graphs that are trees, equal to the no. leaves in the tree.
- Average degree of concurrency,
  - the average number of tasks that can run concurrently over the entire duration of execution of the program.
- Example:
  - First decomposition of matrix-vector multiplication has a fairly small granularity and a large degree of concurrency.
  - Second decomposition for the same problem has a larger granularity and a smaller degree of concurrency.

Task-dependency graph

- Some tasks may use data produced by other tasks and thus may need to wait for these tasks to finish execution.
- It is an abstraction used to express such dependencies among tasks and their relative order of execution
  - A task-dependency graph is a directed acyclic graph in which the nodes represent tasks and the directed edges indicate the dependencies amongst them
  - The task corresponding to a node can be executed when all tasks connected to this node by incoming edges have completed.
  - Task-dependency graphs can be disconnected & the edge-set of a task-dependency graph can be empty.
    - The case for matrix-vector x, where each task computes a subset of the entries of the product vector.
- The nodes with no incoming edges are *start nodes* and the nodes with no outgoing edges are *finish nodes*.
- Degree of concurrency depends on the shape of the task-dependency graph
- A feature of a task-dependency graph that determines the average degree of concurrency for a given granularity is its *critical path*.
  - The longest directed path between any pair of start and finish nodes: the critical path.
  - The sum of the weights of nodes along this path is the critical path length, where the weight of a node is the size or the amount of work associated with the corresp task.
  - Ratio of the total amount of work to the critical-path length: average degree of conc.
  - A shorter critical path favors a higher degree of concurrency.

Limitation factors

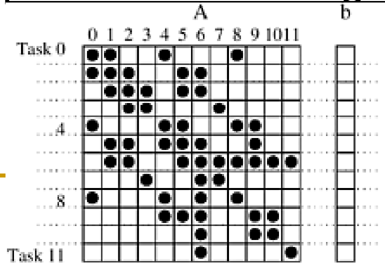
- Usually, there is an inherent bound on how fine-grained a decomposition a problem permits.
  - For instance, there are n^2 multiplications and additions in matrix-vector multiplication considered in the example and the problem cannot be decomposed into more than $O(n^2)$ tasks even by using the most fine-grained decomposition.
- The interaction among tasks running on different physical processors.
  - The tasks that a problem is decomposed into often share input, output, or intermediate data.
  - The dependencies in a task-dependency graph usually result from the fact that the output of one task is the input for another.
  - Example: in the database query example, tasks share intermediate data; the table generated by one task is often used by another task as input.

Task-interaction graph

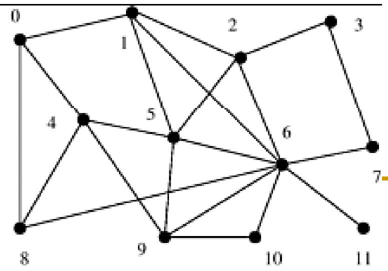
- Depending on the definition of the tasks and the parallel programming paradigm, there may be interactions among tasks that appear to be independent in a task-dependency graph.
  - Example, in the decomposition for matrix-vector multiplication, although all tasks are independent, they all need access to the entire input vector *b*.
    - Since originally there is only one copy of *b*, tasks may have to send and receive messages for all of them to access the entire vector in the distributed-memory paradigm.
- Task-interaction graph:
  - Captures the pattern of interaction among tasks
  - The nodes in a task-interaction graph represent tasks and the edges connect tasks that interact with each other.
  - The nodes and edges of a task-interaction graph can be assigned weights proportional to the amount of computation a task performs and the amount of interaction that occurs along an edge, if this information is known.
  - The edges in a task-interaction graph are usually undirected, but directed edges can be used to indicate the direction of flow of data, if it is unidirectional.
  - The edge-set of a task-interaction graph is usually a superset of the edge-set of the task-dependency graph.
  - Example: in the database query example, the task-interaction graph is the same as the task-dependency graph.

Ex. 2: Sparse matrix-vector multiplication

- Consider: ?the product *y* = *Ab* of a sparse *n* x *n* matrix *A* with a dense *n* vector *b*.
- A matrix is considered sparse when a significant no. entries in it are zero and the locations of the non-zero entries do not conform to a predefined structure or pattern.
- Arithmetic operations involving sparse matrices can often be optimized significantly by avoiding computations involving the zeros.
- While computing the *i*th entry *y*[*i*]=sum(*A*[*i,j*]*b*[*j*],*j*=1..*n*) of the product vector, we need to compute the products *A*[*i, j*] x *b*[*j*] for only those values of j for which *A*[*i, j*] not equal with 0.
  - For example, *y*[0] = *A*[0, 0].*b*[0] + *A*[0, 1].*b*[1] + *A*[0, 4].*b*[4] + *A*[0, 8].*b*[8].
- One possible way of decomposing this computation is to partition the output vector y and have each task compute an entry in it (Fig. (a))
  - Assigning the computation of the element *y*[*i*] of the output vector to Task *i*,
  - Task *i* is the "owner" of row *A*[*i*, *] of the matrix and the element *b*[*i*] of the input vector.
  - The computation of *y*[*i*] requires access to many elements of *b* that are owned by other tasks
  - So Task *i* must get these elements from the appropriate locations.
    - In the message-passing paradigm, with the ownership of *b*[*i*],Task *i* also inherits the responsibility of sending *b*[*i*] to all the other tasks that need it for their computation.
    - For example: Task 4 must send *b*[4] to Tasks 0, 5, 8, and 9 and must get *b*[0], *b*[5], *b*[8], and *b*[9] to perform its own computation.
- Task-interaction graph is shown in Fig.(b).



(a)



(b)

Classification of Decomposition Techniques

1. recursive decomposition,
2. data-decomposition,
3. exploratory decomposition, and
4. speculative decomposition

- The recursive- and data-decomposition techniques are relatively general purpose as they can be used to decompose a wide variety of problems.
- Speculative- and exploratory-decomposition techniques are more of a special purpose nature because they apply to specific classes of problems.