V. Models of parallel computers - after PRAM and early models

March 29h, 2010

Content

- dataflow architectures
- systolic architectures
- circuit model
- graph model
- LogP
- LogGP,
- message-passing paradigm
- levels of parallelism

Dataflow architectures

- Aim:
  - make the aspects of a parallel com. explicit at the machine level
    - without artificial constraints limiting the available parallelism in the program
- The idea:
  - The program is represented by a graph of essential data dependences
    - rather than as a fixed collection of explicitly sequenced threads of control.
  - An instruction may execute whenever its data operands are available.
  - The graph may be spread arbitrarily over a collection of processors.
  - Each node specifies op.&address of each of the nodes that need the result
- A processor in a dataflow machine operates a simple circular pipeline.
  - A message (*token)*, cons.of data&an address (*tag)* of its destination node.
  - The tag is compared against those in a matching store.
    - If present, the matching token is extracted & the instruction is issued for exec
    - If not, the token is placed in the store to await its partner.
  - When a result is computed, a new message, or token, containing the result data is sent to each of the destinations specified in the instruction.

Dataflow architectures

- The primary division is whether the graph is
  - *static*, with each node representing a primitive operation, or
  - *dynamic*, in which case a node can represent the invocation of an arbitrary function, itself represented by a graph.
    - In dynamic or *tagged-token* architectures, dynamically expanding the graph on function invocation is achieved by carrying context information in the tag
- The key characteristic of dataflow architectures:
  - the ability to name operations performed anywhere in the machine,
  - the support for synchronization of independent operations, and
  - dynamic scheduling at the machine level.
- The machine provides the ability to name a very large and dynamic set of threads which can be mapped arbitrarily to processors.
  - Opposite - in data parallel archs, the compiler or sequencer maps a large set of “virtual procs” ops onto procs by assigning iterations of a regular loop nest.
- Typically, these machines provided a global address space
- Dataflow architectures experienced a gradual separation of programming model and hardware structure as the approach matured

Systolic Architectures

- Replace a single sequential proc by a regular array of simple PEs and obtain very high throughput with modest memory bandwidth requirements
- The early proposals were driven by the opportunity offered by VLSI to provide inexpensive special purpose chips
- Differ from conventional pipelined function units:
  - the array structure can be non-linear, *e.g*. hexagonal,
  - the pathways between PEs may be multidirectional,
  - each PE may have a small amount of local instruction and data memory.
- Differ from SIMD in that each PE might do a different operation.
- Aspects in common with mes.passing, data parallel,&dataflow models, bu takes on a unique character for a specialized class of problems
  - Alg. represented as a collection of comp. units connected in a regular pattern.
  - Data move at regular “heartbeats” as determined by local state.
- Example: computation of an inner product:
  - at each beat the input data advances to the right,
  - is multiplied by a local weight, and
  - is accumulated into the output sequence as it also advances to the right.

Systolic Architectures

- Practical realizations of these ideas:
  - iWarp: general programmability in the nodes, in order for a variety of algorithms to be realized on the same hardware.
- The network can be configured as a collection of dedicated channels, representing the systolic communication pattern,
  - data can be transferred directly from processor registers to processor registers across a channel.
  - the global knowledge of the communication pattern is exploited to reduce contention and even to avoid deadlock.
- The key characteristic of systolic architectures:
  - ability to integrate highly specialized computation under a simple, regular, and highly localized communication patterns.
- Systolic algs: generally amenable to solutions on generic machines
  - regular, local communication pattern yield good locality
  - the communication bandwidth needed is low, and
  - the synchronization requirements are simple.
  - ⇒ algs have proved effective on the entire spectrum of parallel machines.

Circuit Model

- Model the machine at the circuit level, so that all computational and signal propagation delays can be taken into account.
  - is impossible for a complex supercomputer, because
    - generating and debugging detailed circuit specifications are not much easier than a full-blown implementation
    - a circuit simulator would take eons to run the simulation.
- If the circuit is to be implemented on a dense VLSI chip, would include the effect of wires, in terms of
  - the chip area they consume (cost) and
  - the signal propagation delay between and within the interconnected blocks (time).

  Note: in modern VLSI design wire delays and area are beginning to overshadow switching or gate delays and the area occupied by devices, respectively.

  E.g. For the hypercube architecture, interprocessor wire delays can dominate the intraprocessor delays,
  - ⇒ communication step time much larger than that of the mesh- and torus-based architectures.

Circuit Model

- Determine bounds on area and wire-length parameters based on network properties,
  - without having to resort to specification & layout with VLSI design tools.
- Examples:
  - in 2D VLSI implementation, the bisection width of a network yields a lower bound on its layout area in an asymptotic sense.
    - bisection width is B => smallest dimension of the chip should be at least Bw, where w is the minimum wire width
  - In the case of 2D meshes, the area lower bound will be linear in the number p of processors.
    - Such an architecture is said to be *scalable* in the VLSI layout sense.
  - Hypercube: the area required is a quadratic function of p and the architecture is not scalable.
- Power consumption of digital circuits is another limiting factor:
  - Power dissipation in modern microprocessors
    - grows ~ linearly with the product of die area & clock frequency (both rising)
    - today stands at a few tens of watts in high-performance designs.
  - Disposing of the heat generated by 1 M procs is a great challenge.

Graph Models

- A distributed-memory arch.: characterized primarily by the network
- The network is usually represented as a graph
  - vertices corresponding to processor–memory nodes and
  - edges corresponding to communication links.
    - If communication links are unidirectional, then directed edges are used.
    - Undirected edges imply bidirectional communication
- Parameters of an interconnection network include
  1. ***Network diameter*:**
     - the longest of the shortest paths between various pairs of nodes,
     - should be relatively small if network latency is to be minimized.
  2. ***Bisection (band)width*:**
     - smallest no. links need to be cut in order to : netw into 2 subnetws of 1/2 size.
     - important when nodes communicate with each other in a random fashion
     - a small value limits the rate of data transfer between the two halves of the netw
  3. ***Vertex or node degree*:**
     - No. communication ports required of each node,
     - Constant independent of network size if arch.is readily scalable to larger sizes.
     - Influence the cost of each node

Bulk-synchronous parallel (BSP) model -1990

- attempts to hide the communication latency altogether through a specific parallel programming style
  - thus making the network topology irrelevant
- Synchronization of processors occurs once every *L* time steps, where *L* is a periodicity parameter.
- A parallel computation consists of a sequence of *supersteps.*
  - In a given superstep, each processor performs a task consisting of local computation steps, message transmissions, and message receptions from other processors
  - Data received in messages will not be used in the current superstep but rather beginning with the next superstep.
  - After each period of *L* time units, a global check is made to see if the current superstep has been completed.
    - If so, then the processors move on to executing the next superstep.
    - Otherwise, the next period of *L* time units is allocated to the unfinished superstep.

LogP model - 1996

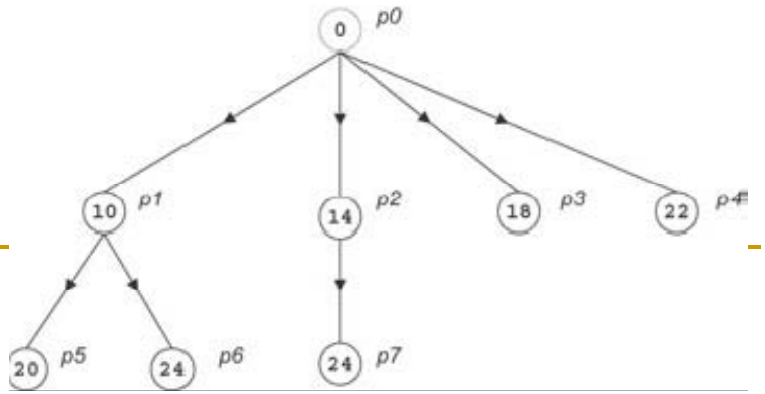
- The development of efficient parallel algorithms suffers from the proliferation of available interconnection networks:
  - for algorithm design must be done virtually from scratch for each new architecture.
  - ? abstract away the effects of the interconnection topology (as PRAM for global-mem. mach.) in order to free the alg. designer from a lot of machine-specific details.
- Models that replace the topological information reflected in the interconnection graph with a small number of parameters do exist
  - have been shown to capture the effect of interconnection topology fairly accurately.
- An example of such abstract models: LogP model (1996).
- The model specifies the performance characteristics of the interconnection network, but does not describe the structure of the network.
- The model has four basic parameters:
  1. *L*: an upper bound on the *latency*, or delay, incurred in sending a message from its source processor to its target processor.
  2. *o*: the *overhead:* length of time that a processor is engaged in the transmission or reception of each message - during this time the proc.cannot perform other ops.
  3. *g*: the *gap* between messages: the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor.
  4. *P*: the number of processors.

LogP model

- The processors communicate by point-to-point short messages.
- Assumes a unit time for local operations and calls it a processor cycle.
- The parameters *L*, *o*, and *g* are measured as multiples of the processor cycle.
- It is assumed that the network has a finite capacity
  - at most *L*/*g* mess can be in transit from any processor or to any processor at any time.
  - If a processor attempts to transmit a message that would exceed this limit, it stalls unti the message can be sent without exceeding the capacity limit.
- The model is asynchronous:
  - processors work asynchronously, and the latency experienced by any message is unpredictable but is bounded above by *L* in the absence of stalls.
- Because of variations in latency, the messages directed to a given target processor may not arrive in the same order as they are sent.
- Parameters are not equally important in all situations; often it is possible to ignore one or more parameters and work with a simpler model.
  - Algs. that communicate data infrequently: ignore the bandwidth and capacity limits.
  - If messages are sent in long streams pipelined through the network (transmission time is dominated by the inter-message gaps) the latency may be disregarded.
  - In some MPPs the overhead dominates the gap, so *g* can be eliminated.

Example 1/LogP: optimal broadcasting a single data unit from one processor to P - 1 others

- Idea: all processors that have received the data unit transmit it as quickly as possible, while ensuring that no processor receives more than one message.
- The source of the broadcast begins transmitting the data unit at time 0.
- The first data unit enters the network at time *o*, takes *L* cycles to arrive at the destination, and is received by the processor at time *L* + 2*o*.
- Meanwhile the source will initiate transmission to other procs at time *g*, 2*g*,…,
- Assuming *g* ≥ *o*, each of which acts as the root of a smaller broadcast tree.
- The optimal broadcast tree for *p* processors is unbalanced with the fan-out at each node determined by the relative values of *L*, *o*, and *g*.
- Figure : optimal broadcast tree for *P* = 8, *L* = 6, *g* = 4, and *o* = 2.
  - No./node: time at which it has received the data unit and can begin sending it on.
  - Proc. overhead of successive transmissions overlaps delivery of previous messages.
  - Procs may experience idle cycles at the end of the algorithm while the last few messages are in transit.

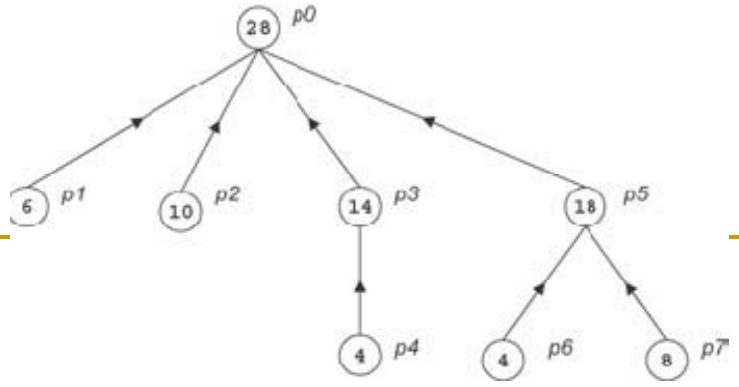


Ex. 2: sum of as many values as possible within time T

- Pattern of communication among the procs again forms a tree (~Ex. 2).
- Each processor has the task of summing a set of the elements and then (except for the root processor) transmitting the result to its parent.
  - Elements to be summed by a proc. consist of original inputs stored in its memory, together with partial results received from its children in the communication tree.

1. Determine the optimal schedule of communication events;
2. Determine the distribution of the initial inputs.

- If $T \leq L + 2o$,
  - the optimal solution is to sum $T + 1$ values on a single proc
- Otherwise
  - the last step performed by the root processor (at time $T - 1$) is to add a value it has computed locally to a value it just received from another processor.
  - Remote proc must have sent the value at time $T - 1 - L - 2o$, and we assume recursively that it forms the root of an optimal summation tree with this time bound.
  - … (see the textbook)
- Fig: communication tree for optimal summing for $T = 28, P = 8, L = 5, g = 4, o = 2$.



Message-Passing Paradigm

Principles

- The paradigm is one of the oldest and most widely used approaches for programming parallel computers.
  - its roots can be traced back in the early days of parallel proc.
  - its wide-spread adopted
- There are two key attributes that characterize the paradigm.
  1. Assumes a partitioned address space and
  2. It supports only explicit parallelization.
- Each data element must belong to one of the partitions of the space
  - data must be explicitly partitioned and placed
  - encourages locality of access
- All interactions (read-only, read/write) require cooperation of 2 processes:
  1. the process that has the data and
  2. the process that wants to access the data.
- Advantage of explicit two-way interactions:
  - the programmer is fully aware of all the costs of nonlocal interactions, and is more likely to think about algorithms (and mappings) that minimize interactions.
  - paradigm can be efficiently implemented on a wide variety of architectures.
- Disadvantage:
  - For dynamic and/or unstructured interactions the complexity of the code written for this type of paradigm can be very high for this reason.

Programming issues

- Parallelism is coded explicitly by the programmer:
  - The programmer is responsible:
    - for analyzing the underlying serial algorithm/application and
    - Identifying ways by which he/she can decompose the computations and extract concurrency.
  - Programming using the paradigm tends to be hard and intellectually demanding.
  - Properly written message-passing programs can often achieve very high performance and scale to a very large number of processes.
- Progs are written using the asynchronous or loosely synchronous paradigms.
  - In the asynchronous paradigm:
    - all concurrent tasks execute asynchronously.
    - such programs can be harder to reason about & can have non-deterministic behavior
  - Loosely synchronous programs:
    - tasks or subsets of tasks synchronize to perform interactions.
    - between these interactions, tasks execute completely asynchronously.
    - Since the interaction happens synchronously, it is still quite easy to reason about the program
- Paradigm supports execution of a different program on each of the p processes.
  - provides the ultimate flexibility in parallel programming,
  - makes the job of writing parallel programs effectively unscalable.
  - ⇒ most programs are written using the single program multiple data (SPMD) approach.
    - In SPMD programs the code executed by different processes is identical except for a small nc processes (e.g., the "root" process).

Building Blocks: Send and Receive Operations

- In their simplest form, the prototypes of these operations are
  - send(void *sendbuf, int nelems, int dest)
  - receive(void *recvbuf, int nelems, int source)
    - The sendbuf points to a buffer that stores the data to be sent,
    - recvbuf points to a buffer that stores the data to be received,
    - nelems is the number of data units to be sent and received,
    - dest is the identifier of the process that receives the data, and
    - source is the identifier of the process that sends the data.
- Performance ramifications of how these functions are implemented.
- Example:

P0	P1
a = 100;	receive(&a, 1, 0)
send(&a, 1, 1);	printf("%d\n", a);

- Most platforms have additional hardw support for send&recv. messages:
  - Asynchronous message transfer using network interface hardware.
    - allow the transfer from buffer memory to desired location without CPU intervention.
  - DMA (direct memory access)
    - allows copying of data from one memory location to another without CPU support
    - If send returns before the communication operation has been accomplished, P1 might receive the value 0 in a instead of 100!

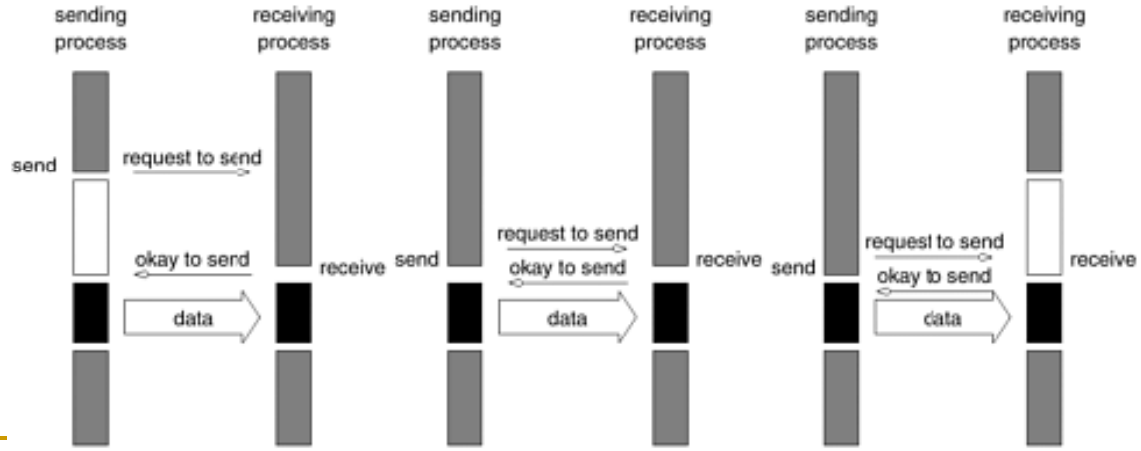
Blocking Message Passing Operations

- The sending operation blocks until it can guarantee that the semantics will not be violated on return irrespective of what happens in the program subsequently.
- There are two mechanisms by which this can be achieved:
  1. Blocking Non-Buffered Send/Receive.
  2. Blocking Buffered Send/Receive.

1. **Blocking Non-Buffered Send/Receive**
   - The send operation does not return until the matching receive has been encountered at the receiving process.
   - Then the message is sent and the send operation returns upon completion of the communication operation.
   - Involves a handshake between the sending and receiving processes.
     - The sending process sends a request to communicate to the receiving process
     - When the receiving process encounters the target receive, it responds to the request.
     - The sending process upon receiving this response initiates a transfer operation.
   - Since there are no buffers used at either sending or receiving ends, this is also referred to as a non-buffered blocking operation.

Idling Overheads in Blocking Non-Buffered Send/Recv

- 3 scenarios:
  - (a) the send is reached before the receive is posted,
  - (b) the send and receive are posted around the same time,
  - (c) the receive is posted before the send is reached.
  - In cases (a) and (c): there is considerable idling at the sending and receiving process.
  - ⇒ A blocking non-buffered protocol is suitable when the send and receive are posted at roughly the same time.
    - In an asynchronous environment, this may be impossible to predict.



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

Deadlocks in Blocking Non-Buffered Send/Recv

- Consider the following simple exchange of messages that can lead to a deadlock:

```
P0                  P1
send(&a, 1, 1); send(&a, 1, 0);
receive(&b, 1, 1);      receive(&b, 1, 0);
```

- The send at P0 waits for the matching receive at P1
- The send at process P1 waits for the corresponding receive at P0, resulting in an infinite wait.
- Deadlocks are very easy in blocking protocols and care must be taken to break cyclic waits of the nature outlined.
- In the above example, this can be corrected by replacing the operation sequence of one of the processes by a receive and a send as opposed to the other way around.
- This often makes the code more cumbersome and buggy.

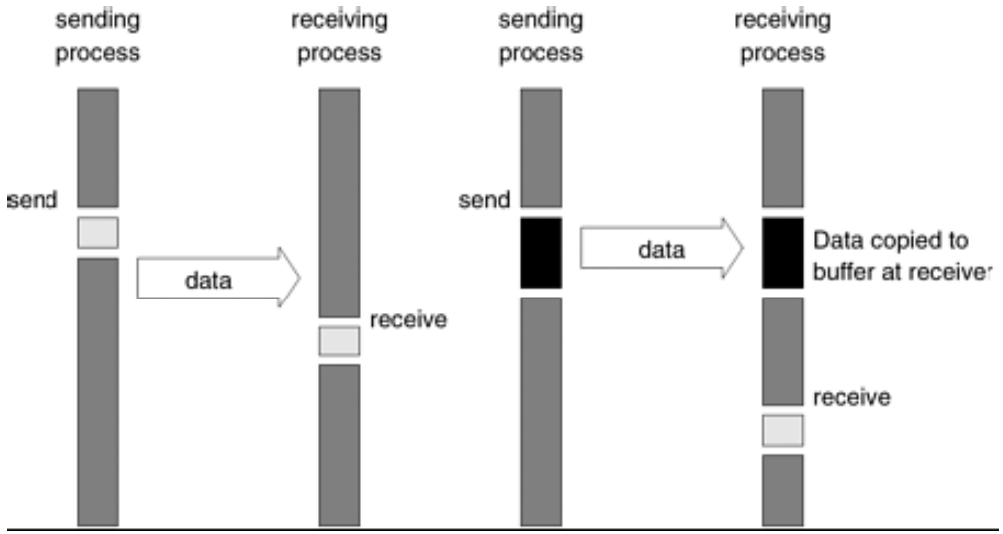
Blocking Buffered Send/Receive

- The sender
  - has a buffer preallocated for communicating messages.
  - copies the data into the designated buffer
  - returns after the copy operation has been completed.
  - continue with the program knowing that any changes to the data will not impact program semantics.
- The actual communication can be accomplished in many ways depending on the available hardware resources.
  - If the hardware supports asynchronous comm. (independent of the CPU), then a network transfer can be initiated after the mess. has been copied into the buffer.
- Receiving end:
  - the data is copied into a buffer at the receiver as well.
  - When the receiving process encounters a receive operation, it checks to see if the message is available in its receive buffer.
  - If so, the data is copied into the target location.

Blocking Buffered Send/Receive

(a) in the presence of communication hardware with buffers at send&receive ends;
(b) in the absence of communication hardware:

- sender interrupts receiver and deposits data in buffer at receiver end.
- both processes participate in a communication operation and the message is deposited in a buffer at the receiver end.
- When the receiver eventually encounters a receive operation, the message is copied from the buffer into the target location.



Impact of finite buffers in message passing

- Consider the following code fragment:

```
P0                                      P1
for (i = 0; i < 1000; i++) {            for (i = 0; i < 1000; i++) {
        produce_data(&a);                       receive(&a, 1, 0);
        send(&a, 1, 1); }               consume_data(&a); }
```

- If proc. P1 was slow getting to this loop, proc. P0 might have sent all of its data
- If there is enough buffer space, then both processes can proceed;
- However, if the buffer is not sufficient (i.e., buffer overflow), the sender would h to be blocked until some of the corresponding receive operations had been posted, thus freeing up buffer space.
- This can often lead to unforeseen overheads and performance degradation.
- It is a good idea to write programs that have bounded buffer requirements.
- A simple code fragment such as the following deadlocks since both processes wait to receive data but nobody sends it.

```
P0                              P1
receive(&a, 1, 1);                      receive(&a, 1, 0);
send(&b, 1, 1);                 send(&b, 1, 0);
```

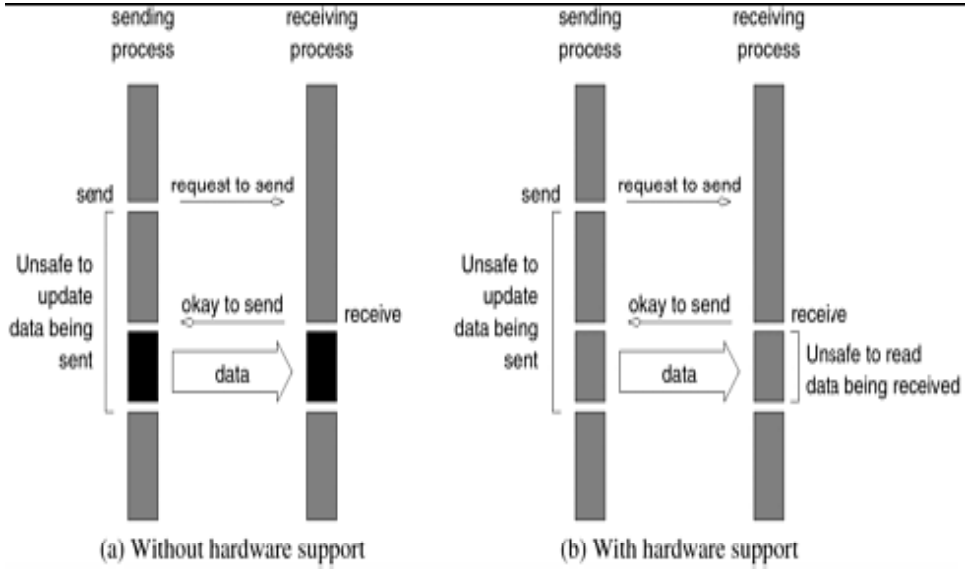
- Deadlocks are caused only by waits on receiveoperations in this case.

Non-Blocking Message Passing Operations

- Often possible to require the programmer to ensure semantic correctness & provide a fast send/receive op. that incurs little overhead.
- Returns from the send/receive op. before it is semantically safe to do so.
- The user must be careful not to alter data that may be potentially participating in a communication operation.
- Non-blocking ops are accompanied by a check-status op. indicating whether the semantics of initiated transfer may be violated or not.
- Upon return from a non-blocking send/recv op., the process is free to perform any comp. that does not depend upon the completion of the op..
- Later in the program, the process can check whether or not the non-blocking op. has completed, and, if necessary, wait for its completion
- Non-blocking operations can be buffered or non-buffered.
  - In the nonbuffered case:
    - a process wishing to send data to another simply posts a pending message and returns to the user program. The program can then do other useful work.
    - When the corresponding receive is posted, the communication operation is initiated.
    - When this operation is completed, the check-status op. nindicates that it is safe for the programmer to touch this data

Space of possible protocols

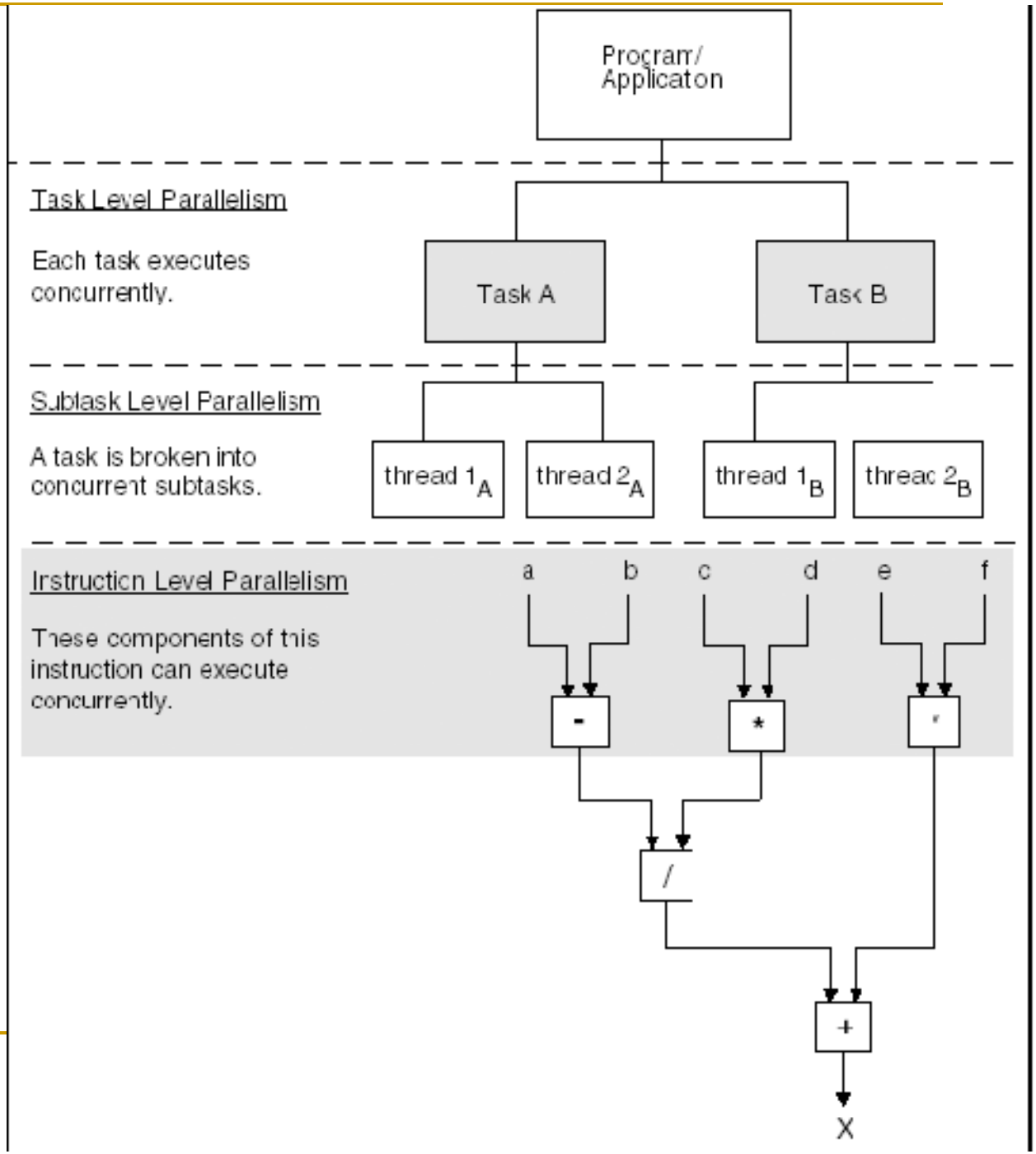
	Blocking Operations	Non-Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non-Buffered	Sending process blocks until matching receive operation has been encountered	
	Send and Receive semantics assured by corresponding operation	Programmer must explicitly ensure semantics by polling to verify completion



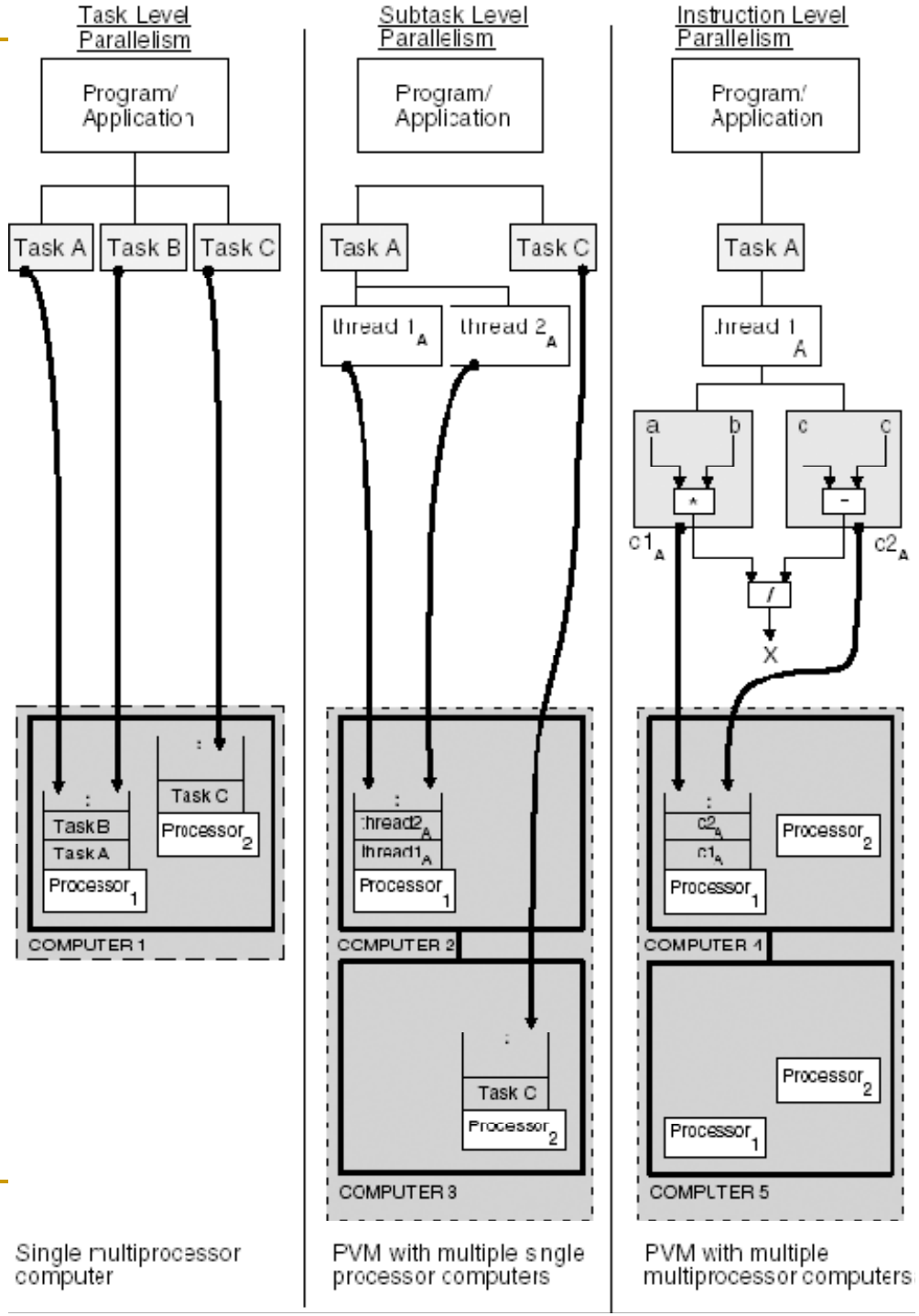
Non-blocking non-buffered send & recv ops
(a) in absence of communication hardware;
(b) in presence of communication hardware.

Levels of parallelism

Levels of parallelism that are possible within a single computer program



Levels of parallelism combined with the basic parallel processor configurations



Parallelism levels

1. *Microparallelization*
   - takes place inside a single processor
   - does not require the intervention of the programmer to implement.
2. *Medium-grain* parallelization
   - associated with language supported or loop level parallelization.
   - While some headway has been made in automating this level of parallelization with optimizing compilers, the results of these attempts are only moderately satisfactory.
3. *Coarse-grain* parallelization
   - associated with distributed memory parallel computers
   - is almost exclusively introduced by the programmer.
4. *Grid-level parallelization*
   - currently the focus of intensive research
   - very promising model for solving large problems,
   - its applicability is limited to certain classes of computational probls, belonging to the “large-scale embarrassingly parallel” category.

Microparallelism

- Modern day desktop processors such as those developed by Intel, AMD, IBM, etc., are already highly parallelized.
  - Such processors have multiple pipelines for integer and floating-point operations, so two different levels of parallelization can be considered:
  1. First, the depth of the pipeline:
     - if a pipeline of depth *k* is used then *k* operations can be executed at the same time.
  2. Second, number of pipelines:
     - assuming that there are *l* integer pipelines of depth *k*1 and *m* floating-point pipelines of depth *k*2 in a given processor, and that all pipelines are operating at maximum capacity at a given moment, then $k1 \times l + k2 \times m$ operations are executed by the processor concurrently in every cycle.
- Availability of this level of parallelism is a function of dependencies inside a stream of machine language operations.
  - These dependencies are analyzed and microparallelism is supported:
  1. by the logic unit inside of the processor on the hardware level and
  2. by the compiler& compiler supplied optimization on the software level.

Medium-grain Parallelism

- Suppose multiple processors are connected to a global (logically and physically) shared memory, the *typical* way of introducing parallelization is to perform similar operations on subsets of data.
- Natural algorithmic level of achieving such parallelization: divide between multiple processors work performed by a loop.
  - A given loop is divided into as many parts as there are processors
  - Assume that the number of parts equals the number of processors
  - Each part is executed independently by a separate processor.
- Supported either through a set of special directives or through high-level language extensions.
- Compilers capable of automatically generating this level of par.
  - The results have been disappointing thus far.
  - Parallelizing compilers are relatively successful in generating par.code with simple loops, addition of two vectors, matrix multiplication, etc.
  - In more complicated cases, i.e., when functions are called inside of loops, the code must still be manually divided into parallel units.

Coarse-level parallelism

- Typical approach to distributed memory parallelization is to create independent programming units that will execute separate work units communicating with each other via message passing
- Minimizing the number of messages passed between components becomes an important goal of program design.
- One must seek to divide a distributed parallel program into large computational units that are as independent from each other as possible and only rarely communicate.
  - Most often each work unit is a derivative of the main program and performs the same subset of operations as the other work units but on separate data sets.
    - This type of an approach is called SPMD (single program, multiple data) and often referred to as coarse level parallelization.
- This level of parallelization must be implemented manually by the programmer as the division of work is based on a *semantic* analysis of the algs used to solve the problem
- One of the more important problems: load balancing.

Coarse-level parallelism

- Since each computational unit is relatively independent it may require different time to complete its work.
  - This may, in turn, lead to a situation when all processors but one remain idle as they wait for the last one to complete its job.
  - Work units should be large and independent & should also complete their tasks within a similar time => SPMD approach somewhat more attractive than the division of work into completely independent units.
- Make the software “match” the underlying hardware.
  - treat a shared memory machine as a distributed memory computer and apply approaches based on message passing.
  - treat a distributed memory computer as a shared memory system (approach impractical)
  - hybrid hardware, where shared memory nodes have been combined into a distributed memory configuration, which results in a distributed shared memory computer.
    - Treating such a machine as a distributed memory computer and applying appropriate parallelization techniques is usually more successful than treating it as a shared memory environment.

Grid Parallelism

- A no. computers, irrespective of their individual architectures are loosely connected via a network.
- In the most general case, each machine and connections between them is assumed to be different.
  - Extremely heterogeneous system,
  - Requires the coarsest level of parallelization:
    - the work must be divided into independent units that can be completed on different computers at different speed and returned to the main solution coordinator at any time and in any order, without compromising the integrity of the solution.
- Examples of successfully tested tasks:
  - analysis of very large sets of independent data blocks, in which the problem lies in the total size of data to be analyzed
    - such as in the SETI@home project