7. Performance of concurrent systems

1. Concurrent computing
2. Parallel algorithms
3. Distributed algorithms
4. Sequential interaction
5. Multi-stream interaction

Inquiry

- How many cores does your device have?
- How do scalability and communication time figure in measuring the efficiency of parallel and distributed solutions?
Topic objective

Describe design and performance issues in parallel, distributed, and interactive computation.

1. Concurrent computing

- What is the main environment of computing today?
- What is the Von Neumann architecture?
- In what computing situations do multiple computational processes occur concurrently?
Subtopic objective

7.1 Describe two forms of concurrency*

Forms of concurrency

- Definition: Interaction of computing entities
- A processor may run two or more programs at the same time (multitasking)
- To do so, it saves its current state in one program’s fetch/execute cycle, and loads the state of another program’s cycle, to run a time slice of that program, numerous times per second
- A process is a programming abstraction that simulates ownership of all computer resources
- Other instances of concurrency: parallel and distributed computing
7. Performance of concurrent systems

Serial computing

• The 50-year-old Von Neumann architecture defines most computing today:
  - one processor
  - data is in memory
  - program is in memory
  - enables general-purpose computing
• The microprocessor carries out a *fetch/execute cycle*, retrieving and executing machine instructions one at a time
• *Multi-core* machines break the one-processor assumption

A Java *thread* simulates ownership of a CPU

• *Thread*: “A unit of execution, … expected to share memory and other resources with other threads executing concurrently” (*Wiktionary*)
• One program may run multiple threads, e.g., for disk access and user I/O
• The processor or processors execute multiple threads concurrently
Parallel and distributed computing

- **Parallel computing** uses
  - processors on the same local bus
  - global clock
  - shared memory
- **Distributed computing** uses
  - processors separated by distance
  - message passing
  - separate clocks
  - shared memory

Why parallel computing?

- Networked PCs can cooperate today to perform gigantic tasks in parallel cheaply
- Multi-CPU servers are common today to meet the needs of client/server computing
- “Moore’s Law”, which predicts computer speed doubling every 1.5 years, may cease to apply
- In a 1-gigahertz CPU, electricity travels only 1 inch per clock cycle, close to size of chip
- *Example:* The brain’s billions of neurons work concurrently
Some parallelizable problems

Challenges:
Given as many processors as you need, and shared memory:

- Search an unsorted array for value $x$ in constant time
- Find best of $n$ tennis players in $O(\log n)$ matches
- Add the elements of an array in logarithmic time

Modeling concurrency

- Circuit model of parallel computing:
  Processing is hard-wired into a combinational circuit that executes an algorithm
- PRAM (Parallel Random-Access Machine):
  an extension of the Von Neumann architecture to multiple CPUs sharing memory
- Process algebras, e.g., $\pi$ Calculus:
  Assume communication by message passing
7. Performance of concurrent systems

PRAM model

- Assume all processors have access to shared memory in $O(1)$ time
- Each processor also has private memory
- Shared memory is globally addressable
- PRAMs are \{concurrent, exclusive\} read, \{concurrent, exclusive\} write – CRCW, CREW, ERCW, EREW
- PRAM is the standard assumption for theoretical work in parallel computation
- *Alternative model*: mesh network, with processors in indirect communication

Single/multiple instructions/data

Approaches employed to achieve parallelism: 

**MIMD**
- Most common
- Every processor may execute a different instruction stream on a different data stream
- May be synchronous or asynchronous, deterministic or nondeterministic

**SIMD**
- All CPUs synchronously execute the same instruction, but on different data items
- *Varieties*: processor arrays, vector pipelines
- Deterministic
Shared-memory architectures

- All CPUs access all memory as global address space
- Memory access time may be uniform (UMA) or non-uniform (NUMA)
- Where processors use cache, cache coherency may be an issue

Synchronization in concurrent computing

- Message exchange may require taking turns receiving and sending
- Access to shared memory may require avoiding contention
- Asynchronous communication is enabled via message queues
- Types of synchronization:
  - **Barrier**: all tasks until last task halt at barrier
  - **Lock/semaphore**: serializes access to global data or to critical section of code
  - Synchronization of (message) **communication**
Cache coherence

- To make slow RAM access appear fast, a *cache* is used on CPUs today, parallel and sequential.
- Using a cache is somewhat like keeping a phone list on your desk to avoid paging through a large phone book.
- CPUs that share memory need to keep a common view of RAM.

Distributed-memory and hybrid architectures

- Memory addresses in *distributed-memory systems* do not map between CPUs; no global address space exists.
- Cache coherency does not apply.
- Scalability is in thousands of CPUs.
- *Hybrid* distributed-shared memory systems network multiple symmetric multiprocessors.
Parallel programming models

- *Programming model*: An abstraction, above hardware and memory architecture, that is not architecture specific
- *Shared memory*
  - May be implemented using distributed physical memory as virtual memory
  - Programming interface, uses global addresses
- *Threads*
  - Like subroutines, called in parallel, share memory
  - UNIX (POSIX) and Java support threads

Other parallel programming models

- *Message-passing interface (MPI)*
  - Tasks exchange data via messages
  - MPI (1994) has been de facto industry standard
  - MPI may be used in shared-memory architectures
- *Data parallel*: parallel tasks work on different parts of a common data structure
- *Hybrid*: combines the four above programming models, e.g., MPI w/ threads or shared memory
2. Parallel algorithms

Q: What is the top parallel-computing speedup?
   a. 2x
   b. n, for n processors
   c. \(n^2\) for n processors
   d. limited by fraction of code that may be executed in parallel

Subtopic objectives

7.2a Design a parallel algorithm
7.2b Characterize the performance of a parallel algorithm*
Parallel speedup

- **Example:** We can search an array in $O(1)$ parallel time by assigning one processor to each array element so that it compares the search key to all elements in one parallel step.

Great speed gains are possible with parallelism

- The folklore speedup theorem: with $p$ processors, the maximum speedup, versus one processor, is $p$ -- false!
- **Counter-examples:**
  - if RAM access is slower than network communication
  - if doubling data doubles cache size
  - if intermediate calculations occupy great time resources
Designing parallel programs

- Problem is parallelizable if it may be partitioned into independent sub-problems, e.g., uniformly changing pixel values
- Examples: search, addition, AND, OR, maximum
- Communication between tasks is necessary if tasks share data; e.g., heat diffusion problem requires knowledge of neighboring data

Parallel and serial fraction

- The parallel fraction of a program is the code that can be executed in parallel
- Example:
  
  \[
  \begin{align*}
  A[1] &\leftarrow 1 \\
  \text{for } i &\leftarrow 1 \text{ to } 3 \text{ do } \quad \text{parallel} \\
  &\quad \text{if } A[i] = 2 \quad \text{fraction} \\
  &\quad \quad y &\leftarrow \text{true} \\
  \end{align*}
  \]
- The serial fraction is the part that’s not parallelizable
Concerns in parallel programming

- *Data dependencies*: Tasks cannot be parallel if one task uses data that is dependent on another.
- *Load balancing*: Distributing work among tasks to minimize idle time.
- *Granularity*: Measure of ratio of computation to communication.
- *Amdahl’s Law*: Maximum speedup is limited by the fraction of code that is parallelizable.
- *Example*

Amdahl’s Law

- “The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.” (Wikipedia)
Parallel search

**Search** (*A[ ], key*)

```plaintext
found ← false
forall processors PID ← 1 to |A|
    if *A[PID]* = key
        found ← true
return found
```

- Parallel time: \( \Theta(1) \)
- PRAM model: CRCW or ERCW
- No contention for access to *found*

Concurrent write

- Example of *incorrect* code:
  **Add** (*A*)
  ```plaintext
  y ← 0
  forall processors PID ← 1 to |A| in parallel
      y ← y + *A[PID]*
  return y
  ```
- This O(1) algorithm fails because *y* is written to concurrently, leading to contention
- Steps are not independent
- To compute actual sum, *y* would have to be incremented more than once
Parallel algorithm: Addition

- Add \( \frac{n}{2} \) adjacent pairs, then add sums pairwise, as in a tournament, until done

\[
\begin{array}{cccccccc}
5 & 2 & 3 & 8 & 1 & 6 & 4 & 5 \\
7 & 11 & 7 & 9 \\
18 & 16 \\
34 \\
\end{array}
\]

- Parallel time = \( \Theta(\log_2(n)) \)
- Pseudocode is similar to that of Broadcast
- Similar algorithms exist for OR, AND, MAX

Parallel AND, OR

AND:
\[
\begin{array}{cccccccc}
1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 \\
0 & 0 \\
0 & 0 \\
\end{array}
\]

OR:
\[
\begin{array}{cccccccc}
1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 \\
1 & 1 \\
1 & 1 \\
\end{array}
\]
Applying associative operators

**Apply (A[], op)**

\[
\begin{align*}
n & \leftarrow |A| \\
distance & \leftarrow 1 \\
\text{while } & n > 1 \\
& n \leftarrow \lceil n / 2 \rceil \\
\text{forall processors } j & \leftarrow 1 \text{ to } n \\
\text{distance} & \leftarrow \text{distance} \times 2 \\
\text{return } & A[1]
\end{align*}
\]

- This algorithm applies \( op \) to \( A \) in parallel time \( \lg n \)

Parallelizable form of associative operators

\[
f(A) = \\
\begin{cases}
\uparrow & \text{if } |A| = 0 \\
A[1] & \text{if } |A| = 1 \\
f(A[1 .. |A|/2]) \oplus f(A[|A|/2 + 1 .. |A|]) & \text{otherwise}
\end{cases}
\]

- Defines function \( f \) that applies associative binary operator \( \oplus \) to values stored in \( A \), suggesting a parallel algorithm that runs in parallel time \( \Theta(\lg n) \)
- Associative operators include \( \text{max} \), AND, \(+\), etc.
Broadcast (*Write-all*)

- Sets all elements of $A$ to $x$

\[
\text{Broadcast} (A[ ], x) \\
A[1] \leftarrow x \\
\text{for } i \leftarrow 1 \text{ to } \lceil \log_2 |A| \rceil \\
\quad \text{forall } PID \leftarrow 1 \text{ to } 2^{i-1} \text{ in parallel} \\
\quad \quad \text{if } |A| > (2^{i-1} + PID) \\
\quad \quad \quad A[ 2^{i-1} + PID ] \leftarrow A[PID ]
\]

- Time: $O(\lg n)$
- At each step, *Broadcast* writes twice as many elements as on previous step

Parallel prefix computation

- Applies associative operators like addition, AND, OR, MAX using *pointer jumping*
- Each CPU has own memory
- At each step each CPU
  a. passes its data, and its left partner’s address, to its right partner
  b. applies operator to two operands
  c. passes the address of its right partner to its left partner, so that its left and right partners are now partners
- After $\log_2 n$ steps, processor $n$ stores the result of the associative operator
Parallel prefix steps

a. Passing data and left partner address to right

b. Passing right partner address to left

c. Storing results after $\lg_2 n$ steps

Parallel prefix computation

- CPUs (boxes here) exchange data and CPU addresses across wider and wider gaps
- In four steps, a 16-CPU array can apply an associative operator (such as $+$, here) to 16 data items and accumulate the result (blue) in rightmost CPU.
Parallel matrix multiplication

- Recall serial algorithm with $\theta(n^3)$ running time
- A parallel solution computes all terms with $n^3$ processors, then computes matrix entries with $n^2$ processors in time $O(n)$
- Better solution uses $\log n + 1$ parallel steps and $n^3$ processors
- This solution achieves logarithmic time, a general goal in parallel computing (at least, polylog time, e.g., $\log^3 n = (\log n)^3$)

Parallel sorting algorithms

- *Odd-even transposition sort* for linear array:
  Adjacent CPUs compare and swap array elements, $n$ times
- *Shear sort* for mesh network architecture:
  Repeat $(\lg n + 1)$ times:
  - Sort each row, using odd-even transposition sort, alternating increasing and decreasing orderings
  - Sort columns
- *Parallel time:* $\Theta(\sqrt{2n \lg n})$
Parallel performance

- Performance depends heavily on inter-processor communication time
- Ways of arranging processors in parallel and distributed architectures:
  - Diameter
    - Mesh $\sqrt{n}$
    - $d$-dimensional hypercube $d$
    - Linear bus (e.g., Ethernet) $n$
- Check against PRAM assumption
- Diameter determines communication time

Work, optimality, efficiency

- Work of a parallel algorithm on input of size $n$: $w(n) = t(n) p(n)$, where $t$ is parallel time, $p$ is number of processors needed for $n$ data items
- For optimal parallel algorithms, $w(n) = T(n)$, where $T(n)$ is serial time
- Efficient parallel algorithm: one for which $w(n) = T(n) \text{ polylog}(n)$ and $t(n) = \text{polylog}(n)$
- Polylog ($n$) is the set of functions that are powers of $\lg n$, e.g. $(\lg n)^2$, $(\lg n)^3$, etc.
- Efficiency is invariant w.r.t. versions of the PRAM model [explain]
Performance metrics

- **Speedup** of a parallel algorithm: Rate of reduction in parallel time as the number of processors rises.
  
  \[ S(k) = \text{Ideal: } S(k) = k \]

- **Efficiency** of algorithm: Work rate per processor,
  
  \[ E(k) = \frac{S(k)}{k} \text{ Ideal: } 1.0 \]

- **Scalability** of algorithm: Ratio of efficiencies at two scales \( \phi(k_1, k_2) = \frac{E(k_2)}{E(k_1)} \).
  
  \[ \text{Ideal: } 1.0 \]

Parallelism and intractability

- Parallel solutions do not make intractable problems tractable
  
  - **Reason**: Exponential work in polynomial parallel time would require an exponential number of processors

- \( Q \): Are there some NP-hard problems that can be solved in polynomial parallel time?
Scalability in parallel systems

- A scalable pair, *(parallel system, parallel algorithm)*, is one in which speedup is roughly linear in number of processors (Gustafson’s Law)

- This requires small *serial fraction* (fraction of unparallelizable steps in algorithm); many algorithms have significant serial fraction

- *Speedup*: serial time / parallel time

- *Efficiency*: speedup / # processors

- *Isoefficiency*: a metric of scalability, the ratio of problem size to minimum # processors $p$ needed to obtain an increase in speedup proportional to $p$

NC: Tractable problems

- NC (“Nick’s Class”, after N. Pippenger) are the problems with parallel solutions – computed in $O((\lg n)^k)$ (polylog) time – by $O(n^c)$ processors – for some constants $c, k$

- NC assumes PRAM model

- PTIME on $O(n^c)$ processors would require an exponential amount of work

*Examples; relate to speedup*
3. Distributed algorithms

- Is the effectiveness of a work team proportional to the sum of the IQs of the team members?

Subtopic objective

7.3 Describe a distributed algorithm
Distributed algorithms

- Distributed algorithms assume distributed, not shared, memory
- Processors are not on the same bus as in parallel systems; *no global clock*
- A central issue is *communication time*
- Examples:
  - Ring broadcast
  - Generalized broadcast
  - Leader election

Categories of distributed algorithms

- Shared-variable (mutual exclusion, contention, cooperation – e.g., producer-consumer)
- Distributed consensus (clock synchronization, process control, database results) to achieve consistent global state
- Distributed network
- Concurrency control
### Ring broadcast

- Operates on systems in which each processor has one successor, one predecessor
- Initiator sends token, then waits for token from predecessor (which indicates success); non-initiator receives token and then sends token
- Token may be accompanied by a message
- *Message complexity* (number of tokens or messages sent): $\Theta(n)$
- Other distributed topologies are possible; *Ring-broadcast’s* confirmation of success does not work for them

### Generalized broadcast

- Operates on ring or non-ring topologies
- Initiator sends token to each neighbor, non-initiators receive token from one neighbor and send to all neighbors
- To confirm success, must add *echo* algorithm, processors must form spanning tree, and initiator must receive echo from all other processors
- Echo spanning-tree’s algorithm has time $\Theta(d)$ and message complexity $m$, where $d$ is diameter of network, $m$ is number of edges
Static distributed algorithms for networks

- Static network algorithms are executed for particular tasks in an environment that does not change while the algorithm runs
- *Examples:*
  - Routing
  - Leader election
  - Function computation

Distributed packet routing

*Given:*
- *Network* $G = (V, E)$
- Many packets are each routed on a path from some source to some destination;
- Only one packet may travel on one edge $e \in E$ in a unit of time;
- Packets join queue at $e$ in case of conflict
- *Problem:* Find a path for each packet
Packet routing policy decisions

• How many steps are needed for all packets to arrive at destinations?
• What is queue management policy?
• When should each packet be released at its source?
• What is schedule of minimum duration?

Leader election

• Among a set of initiators, one must sometimes be chosen to coordinate a task; suppose minimum-ID is criterion
• Leader election for a ring entails broadcasting each processor’s ID and each initiator determining whether its ID is minimum of the network
Time analysis of leader election algorithm

- Time: \( n \) steps; message complexity: \( O(n^2) \)
- Better algorithm has complexity \( O(n \lg n) \)
- On arbitrary topology, leader election problem is \( O(n) \), message complexity \(|V| \lg |V| + |E|\)

Dynamic network “algorithms”

- These are not algorithms, but designs for interactive, non-algorithmic processes
- Examples:
  - Termination
  - Deadlock detection
  - Global snapshot
  - Synchronization
Contention resolution in distributed systems

- **Given:** processes $p_{1..n}$ that access a database
- If any two processes attempt to access the same data at the same time, both must wait
- For all processes just to wait one time step would yield deadlock
- **Problem:** Design a protocol that each process executes, without centralized control, that avoids conflict; i.e., an independent set

Contestion resolution protocol

- At each step, each process attempts to access DB, with probability $p$
- **Analysis:**
  - Probability of success at step $t$: $p(1 - p)^{n-1}$ because process must attempt, and all other processes must not attempt to access
  - Hence success is maximized if $p = 1/n$
  - With probability at least $(1 - 1/n)$, all processes will succeed within $\Theta(n \lg n)$ steps
Randomized distributed contention resolution

- For processes contending for the same resources
- For each process $p_i$
  - Choose random value $x \in \{0,1\}$
  - Enter conflict-free set iff $x = 1$ and each process with which it is in conflict chooses 0

High-speed packet switching

- *Switches* have *input links* $I_1, \ldots, I_m$ and *output links* $O_1, \ldots, O_n$
- Packets arrive on specific input links; must be sent out on specific output links
- *Concern*: To handle packets on separate input links for output on the same output link
- *Solution*: Input and output buffers on each link enable queuing of packets
- *Problem*: How can input queues be managed to simulate pure output queueing?
Packet routing strategies

- **Note:** the only factors in finding low-duration schedules are *dilation* ($d$, maximum path length) and *congestion* ($c$, maximum number of overlapping paths)
- **Randomized strategy:** to disrupt harmful synchronization in schedule to reduce conflicts
- **Algorithm:** each packet
  - Chooses random delay $s$
  - Waits $s$ time steps
  - Moves ahead one edge per step

Scalability in distributed systems

- **Communication time** is a significant factor in determining scalability of distributed systems
- **Quality of service** (QoS) must be maintained for many data streams in a scalable system
- **Productivity:** throughput times average value of response, divided by cost per second
- **Scalable systems:** ones whose productivity is maintained as scale varies
- **Relative scalability:** the proportion of power-cost ratios of two systems at different scales
4. Sequential interaction

- Does a Google car execute an algorithm?
- Does a robot have an algorithm to walk from one room to another?

Subtopic objective

7.4 Distinguish algorithms from interactive processes*
Interactive and multi-agent computing vs. algorithms

- New paradigm: computing is not just *algorithms*, but also *interaction*
- Problems are not *functions*, but *services* and *missions*, with Quality-of-Service constraints
- *Examples of interaction*: agent computation in dynamic environments; reinforcement learning; networked computing
- How well can *multi-agent systems* improve on sequential interaction?

Two kinds of computing

<table>
<thead>
<tr>
<th>Algorithmic</th>
<th>Interactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transforming finite input to finite output (<em>function</em>)</td>
<td>Providing a <em>service</em> over time (by agents)</td>
</tr>
<tr>
<td>Structured design</td>
<td>Object-oriented design</td>
</tr>
<tr>
<td>Logic and search in AI</td>
<td>Agent-oriented AI</td>
</tr>
<tr>
<td>Rule-based reasoning</td>
<td>Adaptation, control</td>
</tr>
<tr>
<td>Compositional behavior</td>
<td>Emergent behavior</td>
</tr>
<tr>
<td>Closed systems</td>
<td>Open systems</td>
</tr>
</tbody>
</table>
Algorithms

Algorithmic computation (Knuth): The effective transformation of a finite, pre-specified input, to a finite output, in a finite number of steps.

- Algorithms compute functions
- A system that executes an algorithm is closed
- Algorithms are equivalent to batch computation

Interactive computation

- Feature of computing today: Computation as an ongoing service, not assumed to terminate
- Must solve problems whose inputs cannot be completely specified a priori
- Dynamic input and output during computation
- Persistence of state between interaction steps
- Environment is an active partner in computation
Driving is nonalgorithmic

- The problem of driving a car is interactive, not reducible to an algorithm
- Given input of a map, algorithmic approach would compute a series of outputs to steering wheel, gas, brake, etc.
- No one could drive anywhere this way
- External conditions can affect car’s motion during driving

Communication

- One-way communication is the sending of strings, over a finite alphabet of symbols, from one entity to another
- Two-way communication is the concurrent activity of two entities engaged in one-way communication with each other
- Two-way communication does not require that either entity waits for an input string before emitting output, or that either entity has an exclusive communication relationship with the other.
Interaction and synchrony

- Direct interaction is two-way communication in which some outputs of each entity may causally affect the entity's later inputs from the other.
- Computing entity interacts synchronously with environment $E$ if $A$ interacts with $E$ and both $A$ and $E$ wait for only one input token before emitting an output token.
- Asynchronous interaction occurs in the absence of synchrony as defined here.

Sequential interaction

(Synchronous) sequential interactive computation: Interaction involving two participants, at least one of which is a finite computing agent (machine, device).

- Characterized by a single interaction stream of input alternating with output.
- If one participant is an agent, the other is its environment.
- Interaction may involve changes of state.
Autonomous agents and models of sequential interaction

- Unified Modeling Language (UML) models sequential-interactive systems not supported by algorithm-based notations like flowcharts, module hierarchies, pseudocode
- *Autonomous agents* may initiate actions and may or may not synchronize with their environments
- To model autonomous agents, standard UML must be extended (Bauer, Muller, Odell, 2000)

POMDPs

- *Partially observable Markov decision processes* are problems in inaccessible stochastic environments with the Markov property
- *Markov property*: Probability that system will be in a given state at next step depends only on current state, not on past history
Reinforcement learning and policy search

- Agents that pre-compute action sequences cannot adapt their behavior to new percepts
- *Reinforcement learning agents* interact with environment
- *RL* evolves a function, called a *policy*, that maps perceived states to actions
- Agent that follows a policy incorporates sensory information about state into action determination

Evolution in dynamic environments

```
t ← 0, Initialize (P₀, E₀), y ← Evaluate (P₀, E₀)
While not terminate (y, t) do
    t ← t + 1
    Eᵣ ← Alter-environment(Eᵣ₋₁, Pᵣ₋₁)
    Pᵣ ← Select(Pᵣ₋₁, y)
    Pᵣ ← Alter-population(Pᵣ)
    y ← Evaluate (Pᵣ, Eᵣ)
```

- The evolutionary algorithm must model the environment explicitly if the environment is dynamic and persistent
- Note that the fitness function *Evaluate* is relative to the evolving state of the environment here
Scalability in algorithms and interaction

• Time efficiency of algorithms is expressed as scalability (rise in running time relative to quantity of data operated on)

• *Intractable* algorithmic problems are unscalable in that running time is considered to be exponential in data size

• A robust reactive system likewise should respond in time that does not increase “too much” in proportion to size of its input

Complexity of interactive protocols and input sequences

• *Complexity* \( C(M; \sigma; w) \) is the total number of microsteps, for all macrosteps in the sequence, plus the number of I/O symbols received multiplied by the ratio of computation speed to communication speed, where \( w \) is starting worktape contents, \( w \in \text{reach}(M) \).

• The *stepwise average time complexity* of the (protocol, input sequence) pair is its time complexity divided by the number of macrosteps, \( C_{\text{avg}}(M; \sigma; w) \).
Time complexity of a sequential-interactive processes and services

- For interactive process $M$, time complexity $C(M)$ is the rate of growth of $(C(M; \sigma; w))$ as the length of $\sigma$ rises, given $\sigma \in (\Sigma^*)^*$; $w \in \Sigma^*$.

- The time complexity of sequential-interactive service $L$ is the time complexity of the minimum-complexity protocol $M$ on which $L$ can be observed, where $M$ is an interactive process and $L = PSL(M)$.

5. Multi-stream interaction

- What, if anything, does the Internet compute?
- What can occur in a three-person conversation that cannot with two persons?
- How do the interactions of a robot and a mobile device differ from those of a laptop?
Subtopic objectives

7.5 Describe features of multi-stream interaction

Multi-stream vs. sequential interaction

- *Multi-stream interaction* occurs when an entity is concurrently interacting with at least two other entities
- *Examples*: Robots and mobile devices with multiple I/O streams
Special features of multi-stream interaction

- In contrast to sequential interaction, it may be characterized by:
  - *Autonomy*, where agents initiate interactions
  - *Nondeterminism* when asynchronous send operations collide
  - *Dynamic linking* and unlinking, creation/destruction of nodes
  - *Indirect* interaction via a shared environment

Agents, environments, and persistent state

- Environment, like interacting agents, has persistent state (memory)
- Environment-agent relation features symmetry
- Indirect interaction uses persistent state of environment as medium
- When environment is *passive* (relays inputs between agents, unaltered), it is medium of pure indirect interaction
Mission vs. function vs. service

- Algebra and propositional logic provide rules for *evaluation* of formulas
- Algorithms compute recursively definable *functions*
- Sequential-interactive agents offer *services*
- Multi-agent systems accomplish *missions* requiring *quality of service* for all users
- Interaction in MASs may be *asynchronous*
- Mission may require a minimum QoS regardless of number of users (*scalability*)

Indirect interaction

- Let $A$ and $E$ interact asynchronously.
- Suppose $E$ may be decomposed into $E'$ and $B$, where $E' = E - \{B\}$
- Then $A$ and $B$ *interact indirectly* via $E$ iff mutual causality holds between the behaviors of $A$ and $B$. 
Indirect interaction and multiagent systems

• In a MAS characterized by locality of interaction and mobility of agents, it is only possible for agents to influence overall system behavior remotely, i.e., indirectly

• Richness of multiagent interaction:
  – due partly to ability of each agent to interact with multiple others
  – hence indirectly with all others (otherwise system partitions)

Scalability in multi-agent systems

• Notions of autonomy and asynchrony, as implied in the notion of agents, shape concept of scalability of MASs

• Condition: Each agent must provide a level of quality of service

• Models of scalability note mesh and hierarchy topologies, but not topologies made possible by shared variables

• Research notes that scalability is limited by any extra load that is due to increase in number of agents in the system
Scalable architectures and models

- *Scalable MAS architecture* (w.r.t. a class of missions): a design architecture whose instances are all scalable w.r.t. that class of missions

- *Scalable computational model* of multi-stream interaction: one capable of serving as the formal foundation of MAS architectures and instances that are scalable w.r.t. nontrivial classes of missions

Some notions of MAS scalability

- *Scalable MAS instance*: one that can perform a class of missions (hence satisfying their constraints) regardless of the number of agents or environmental entities

- *Statically-scalable MAS* (w.r.t. a class of missions): one that is scalable if agents and environmental entities are present at startup time

- *Dynamically-scalable MAS*: one that is scalable if agents and environmental entities may appear or disappear during execution
Unscalability of message passing

- When the system’s mission requires adaptability under conditions where each agent is simple, the massive communication load requires use of the agents’ environment (indirect interaction)
- *Example:* Under message-passing assumption, all agents may communicate with all other agents, requiring $O(n)$ memory overhead per agent and $O(n^2)$ connections among agents, prohibitive where $n$ is large

Showing unscalability of message passing

- *Motivation:* As unscalable architectures in AI are *brittle* and will fail in realistic settings (R. Brooks), so will unscalable MAS architectures and models
- *Hypothesis:* As the number of agents rises asymptotically, either number of connections grows too fast, or else paths between agents become too long
- Other dimensions to show unscalability:
  - Synchronization vs. asynchrony
  - Centralized vs. decentralized storage
References (parallel, distributed)


L. Lamport and N. Lynch. 1990


References (sequential interaction)

Peter Wegner. Why interaction is more powerful than algorithms. CACM 40 (5), 1997.

References (multi-stream)