ACM/CS 114
Parallel algorithms for scientific applications

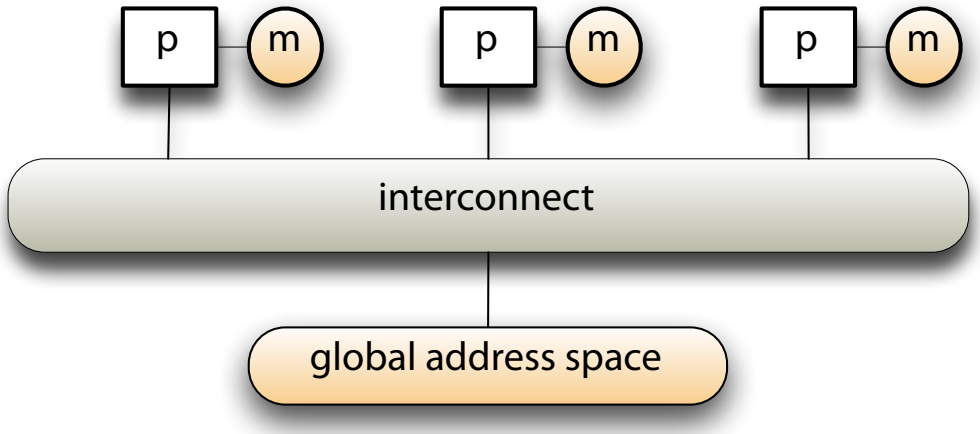
Michael A. G. Aïvázis

California Institute of Technology

Winter 2012

Impact of architecture on algorithm design

- recall the five steps of parallel algorithm design
  - identification of the parallelizable part, partitioning into fine grain tasks, examination of the task communication patterns, task coarsening, and mapping coarse tasks onto processors
- and the layout of the generic parallel architecture:



- let's move memory around and examine how this affects the programming model
- for a trivial but instructive problem

Parallel programming models

- control
  - how is parallelism *created*
  - what is the *sequencing* of instruction streams in each task
  - how do tasks *synchronize*
- data address spaces
  - what data is private to each task; what data must be shared
  - how is logically shared data created, accessed or communicated, and synchronized
- instruction sets
  - what are the fundamental operations for process creation, communication, and synchronization
  - which operations are *atomic*
- cost
  - how fast does it run
  - are resources used efficiently
  - how hard is it to code correctly

Embarrassingly parallel: p processor reduction

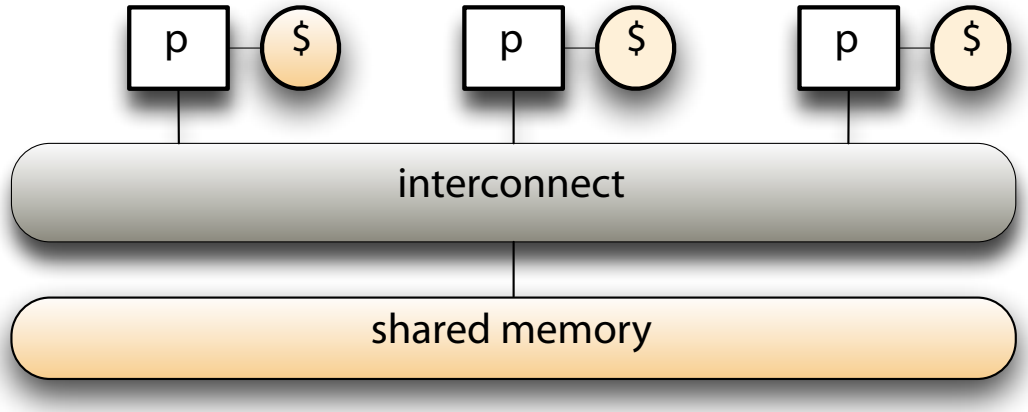
- given a function f and a sequence of numbers S of length N, evaluate the sum

$$s = \sum_{i=0}^{N-1} f(S_i)$$

- parallel tasks: the function evaluations, the computation of partial sums
- strategy: assign n/p numbers to each processor
  - each processor performs n/p evaluations of f
  - each processor computes its own partial sum
  - one(?) of them collects the p partial sums, and computes the global sum s
- two classes of data
  - logically shared:
    - the global sum
    - the input sequence S
  - logically private:
    - the evaluations of f on the local subsequence
    - the local partial sums (?)

Shared memory machines

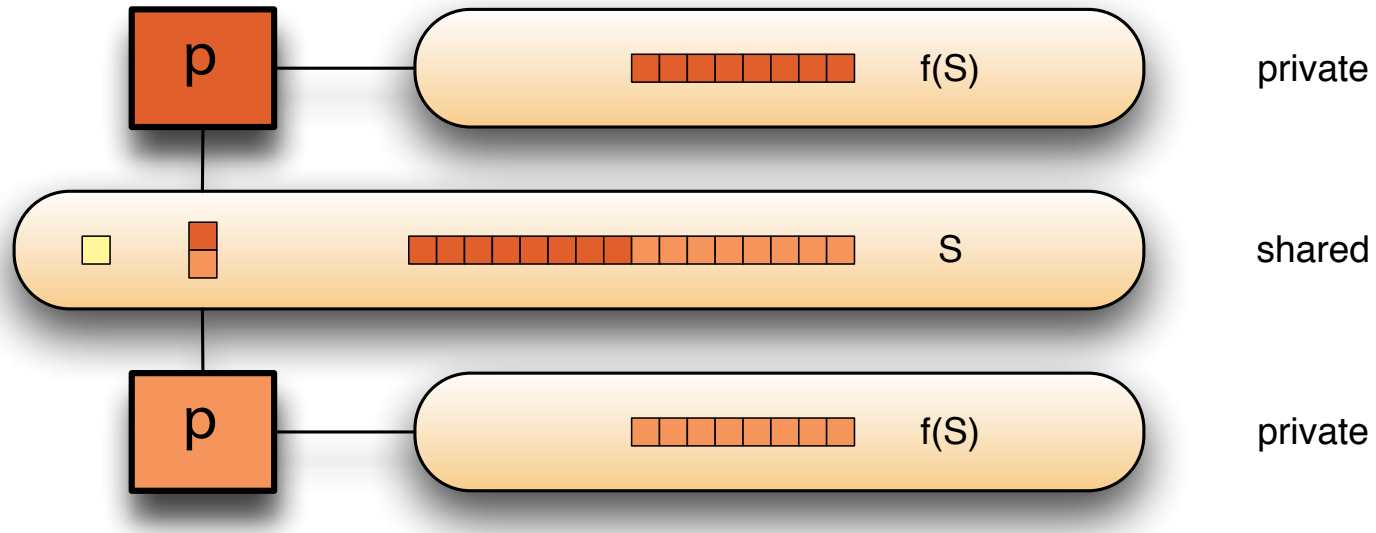
- processors are all connected to a large pool of shared memory with a global address space
- typically, each processor has some local cache, but no private memory
- *cost*: accessing the cache is *much* faster than main memory
  - tune: the memory footprint of n/p numbers should match cache size



- for shared *address space* machine:
  - replace caches with local/private memory
  - cost: repeatedly accessed data should be copied to local storage
  - not done much any more, but recently relevant thanks to hybrid CPU/GPGPU systems and the implementation details of nVidia chips

Programming in a shared address space

- the program creates and manages p instruction streams (threads)
- each with a set of private variables
  - registers, stack, cache
- collectively with a set of shared variables
  - statics, heap
- communication is *implicit*: threads just access the shared memory locations
- synchronization is *explicit*: read/write flags, locks, semaphores



Implementation in a shared address space

- let's implement with two threads

thread 1

```
1 s ← 0
2 s_1 ← 0
3 for i ← 1 to n/2 − 1 do
4 |   s_1 ← s_1 + f(S[i])
5 s ← s + s_1
```

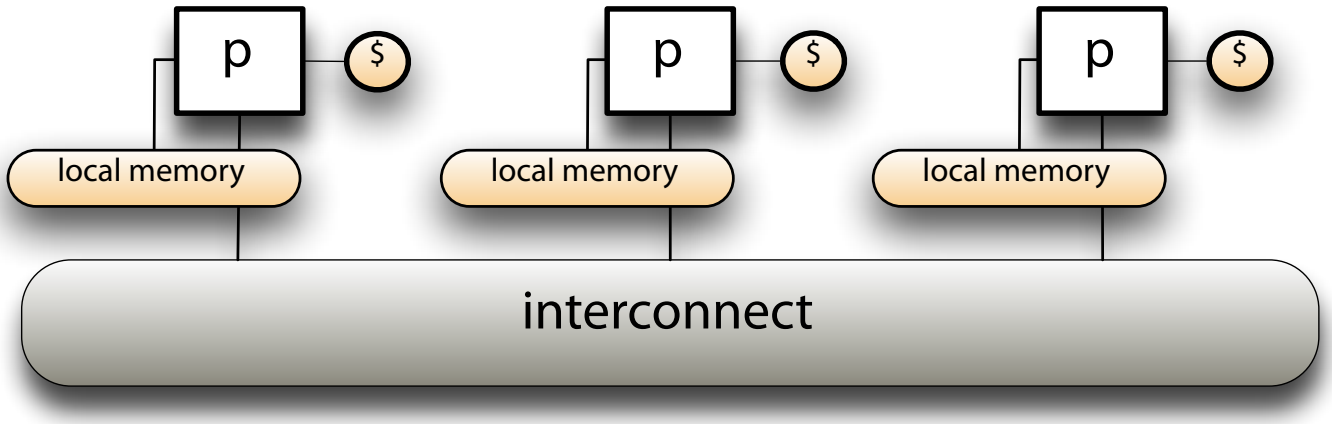
thread 2

```
1 s ← 0
2 s_2 ← 0
3 for i ← n/2 to n do
4 |   s_2 ← s_2 + f(S[i])
5 s ← s + s_2
```

- what is wrong with this code?
  - *race condition*
  - instructions from different threads can be executed in any order
  - can you deduce all the possible values of s after both threads finished executing?
  - one possible solution is to place line 5 in a lock/load/modify/store/unlock block

Distributed memory machines

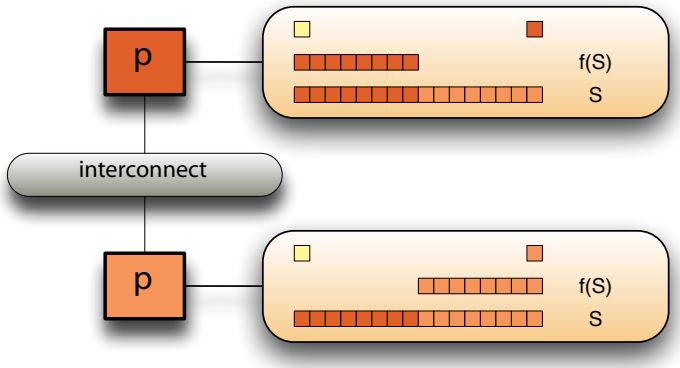
- processors are all connected to their own private memory
- processors have no access to each other's memory, except through explicit exchanges
- each node is connected to a communication substrate: ethernet, myrinet, infiniband



- all communication and synchronization is carried over the interconnect

Programming in a distributed address space

- message passing
  - programs consists of a collection of *n named* processes
    - typically numbered 0 through $n - 1$
    - thread of control, local address space
    - local variables, statics, heap
  - processes communicate via *explicit* data exchanges
    - matching pair of send/receive by source and destination processors respectively
    - primitives for efficient implementation of many-to-many exchanges
  - coördination is implicit in every communication
  - logically shared data must be *partitioned* among the local processes
- standard libraries: MPI, the survivor



Implementation in a distributed address space

- naïve implementation

processor 1

```
1 s_1 ← 0
2 for i ← 1 to n/2 − 1 do
3 |   s_1 ← s_1 + f(S[i])
4 send s_1 to p_2
5 s_2 ← recv from p_2
6 s ← s_1 + s_2
```

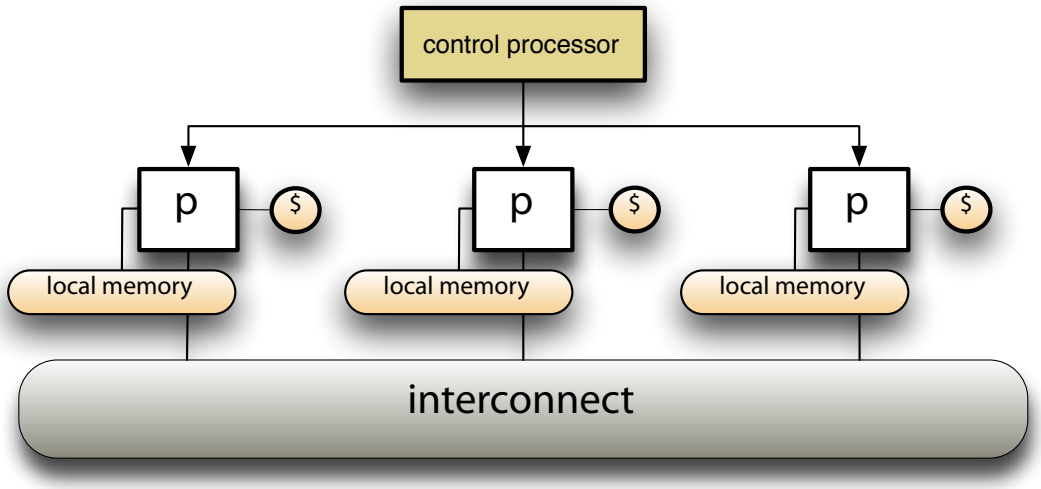
processor 2

```
1 s_2 ← 0
2 for i ← n/2 to n do
3 |   s_2 ← s_2 + f(S[i])
4 send s_2 to p_1
5 s_1 ← recv from p_1
6 s ← s_2 + s_1
```

- what is wrong with this code?
  - *race condition*; more subtle than before
  - send may block until a matching receive is executed
- options:
  - pair up sends and receives to create logically atomic exchanges
  - use non-blocking or asynchronous primitives
  - use a many-to-many communication primitive, if available

SIMD machines

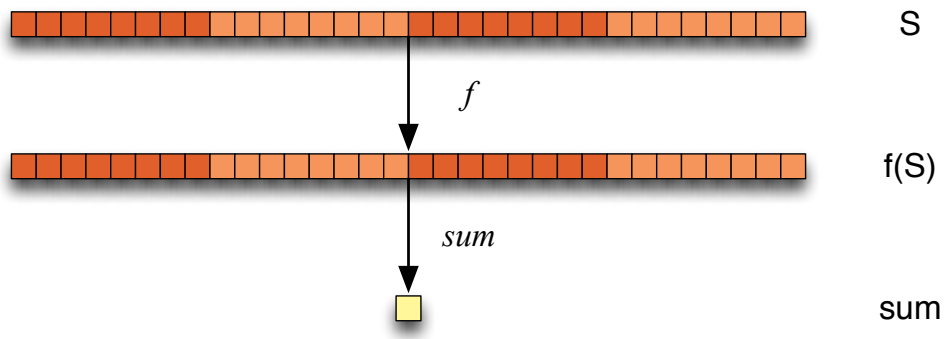
- a large number of small, special purpose processors
- a single "controller" manages the instruction stream
  - each processor executes the same instruction on its local data
  - may be able to specify which processors are active/idle



- hardware of this type fell out favor years ago
  - but the associated programming model is still popular
  - implemented by mapping n-fold parallelism to p processors
  - mostly done by compilers, e.g. *High Performance FORTRAN* (HPF)
- relevant again thanks to the CPU+GPGPU hybrids, which have similar layout

Data parallel programming model

- single instruction stream of *parallel* operations
- parallel operation applied to entire data structure
  - you may be able to restrict the *range* of operations to some defined subset of the data
- communication and synchronization are implicit in the definition of the parallel operators



- rather elegant, easy to understand, easy to reason about
- unfortunately, not all problems fit the paradigm nicely
- implemented by parallel functional languages, MATLAB

Clusters

- commodity hardware configurations:
  - CPUs with multiple cores: $2 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow ?$
  - motherboards with multiple CPUs
  - sizeable memory on a single board
- this hybrid is the current mainstream deployment: most of the Top500
- shared memory within a *blade*, message passing across blades
- there had to be an acronym: CLUMPs; fortunately no one uses it...
- programming models:
  - treat machine as flat and always use message passing
    - simple but ignores the performance characteristics of the memory hierarchy
    - message passing library may be smart enough to switch between network and shared memory dynamically, e.g. openMPI
  - expose both layers explicitly
    - higher performance, but unpleasant to program
    - hard to make portable

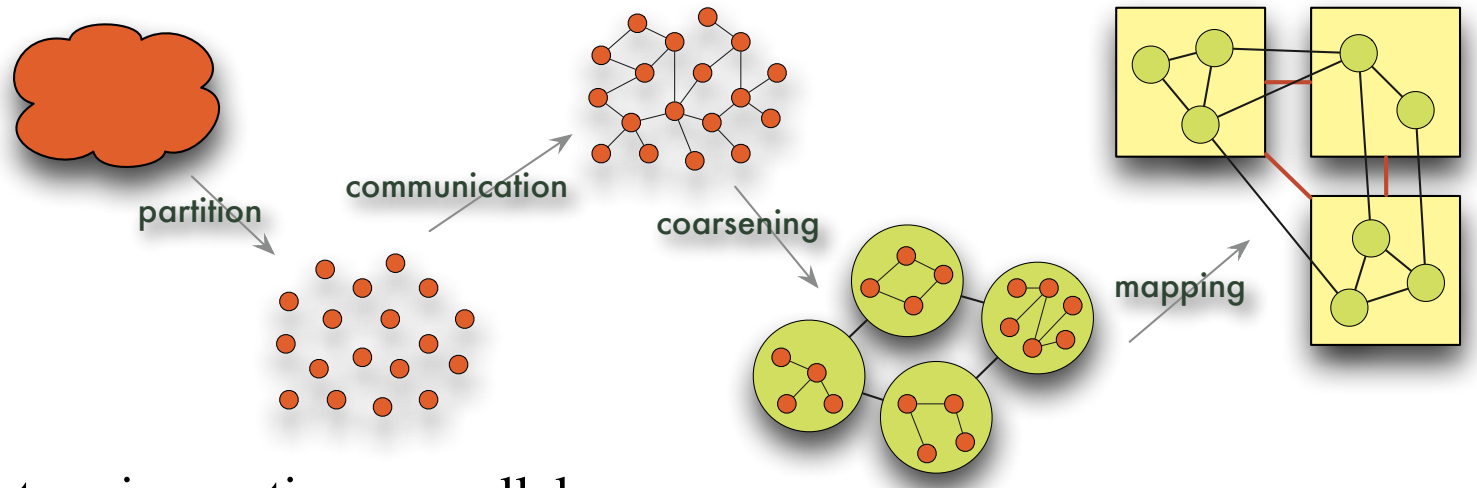
Bulk synchronous programming model

- strategy that applies to both shared memory and message passing programming models
- program consists of interleaved *phases* synchronized by global barriers
  - *compute* phases: all processors
    - operate on local data – distributed memory
    - operate with read access to global data – shared memory
  - *communication* phases: all processors participate in data exchanges, rearrangements or reductions of global data
- single program multiple data
  - everybody is doing the same thing
  - maps well to the structure of solutions of PDEs
    - exchange data on processor partition boundaries
    - apply boundary conditions
    - agree on a stable Δt size (a global reduction)
    - advance the solution by Δt
    - repeat until you run out of time...

Recap

- in the early days, each machine was unique
  - hardware, support for one programming model, perhaps a dedicated language with its compiler
  - when a new machine came out you had to throw all your code away and start again
    - ok for getting a thesis out, bad for building a career
- now we distinguish the programming model from the underlying hardware, so we can write portable, *correct* code that runs on large classes of machines
- unfortunately, writing *fast* code still requires tuning for some architectural details
  - the design challenge is to simplify this process
  - e.g., by exposing these details as user-configurable options that are determined at run-time
  - best handled by *application frameworks*

Parallelization steps



- steps in creating a parallel program
  - identify the work that can be done in parallel
  - partition it in terms of work units, the fine grain tasks
  - analyze the communication patterns among work units
  - coarsen into processes, the abstract entities that carry out tasks
  - map to processors, the physical entities that execute the processes
- goal: maximize the speedup due to parallelism

Parallelizing our reduction example

- for our example $s = \sum f(S)$
- partition into tasks
  - computing each of the $F(S_i)$
  - n-fold parallelism, where ideally $n \gg p$
  - computing the sum s
- communication
  - distribution of the initial input sequence S
  - collection of the partial sums
- coarsening: compute the partial sum of a cluster of evaluations
  - thread k sums up $s_k = \sum_{i=kn/p}^{(k+1)n/p1} f(S_i)$
  - thread 1 sums up the partial results and communicates the result to the other threads
- mapping: processor i runs thread i
- at *runtime*
  - start up the p threads
  - communicate/synchronize with thread 1