Parallel programming

- Distribute the work
  - Load/work balancing

- Distribute the data
  - Small (and “balanced”) communication

This seems easy...

But there are some details to consider...
Speed-up and scalability

- Scalability in the parallel architecture
  - Processor numbers
  - Memory architecture
  - Interconnection network

- Scalability in the computational problem
  - Problem size
  - Computational algorithms
    - computation to memory access ratio
    - computation to communication ratio

- Parallel programming models and tools

- Application scalability ("actual scalability")
There is a lot of complexity...

- Four layers
  - application
    - algorithm, data structures
  - parallel programming interface / middleware
    - Choice of programming paradigm and language, compiler, parallel libraries, communication, synchronization
  - operating system
    - process and memory management, IO
  - hardware
    - CPU, memory, network

- Interaction between different layers!
Parallel programming - performance engineering and productivity

- Scalable, optimized applications deliver HPC promise
- Optimization through performance engineering process
  - Understand performance complexity and inefficiencies
  - Tune application to run optimally on high-end machines
- How to make the process more effective and productive?
- What performance technology should be used?
  - Performance technology part of larger environment
  - Programmability, reusability, portability, robustness
  - Application development and optimization productivity
- Process, performance technology, and its use will change as parallel systems evolve
- Goal is to deliver effective performance with high productivity value now and in the future
Sequential performance

- Sequential performance is all about:
  - How time is distributed
  - What resources are used where and when

- "Sequential" factors
  - Computation
    - choosing the right algorithm is important
    - compilers can help
  - Memory systems and cache and memory
    - more difficult to assess and determine effects
    - modeling can help
  - Input / output
Parallel performance

- Parallel performance is about sequential performance AND parallel interactions
  - Sequential performance is the performance within each thread of execution
  - “Parallel” factors lead to overheads
    - concurrency (threading, processes)
    - interprocess communication (message passing)
    - synchronization (both explicit and implicit)
  - Parallel interactions also lead to parallelism inefficiency
    - load imbalances
Load/work balancing

- Refers to the practice of distributing work among tasks so that all tasks are kept busy all of the time.
- Or... Minimization of task idle time
- Important for performance reasons – Amdal's law
Work balancing
Work Balancing

How to achieve

- Staically partition the work so that each task gets the same amount of work
- Use dynamic assignment of tasks
**Stational work balancing**

- For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.

- More general: For loop iterations where the work done in each iteration is known to be equal, evenly distribute the iterations across the tasks.
Dynamic work balancing

- When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a **scheduler - task pool** approach. As each task finishes its work, it queues to get a new piece of work.

- An algorithm/tool which detects and handles load imbalances as they occur dynamically within the code is needed.

  - Sparse arrays: Some task with zeros
  - Adaptive grid methods: Some task need to refine their mesh
Granularity

- Granularity is a qualitative measure of the size of the parallel tasks
- Often related to communication/synchronisation: tasks are typically dependent on communication/synchronization events
- Can build a “task tree” to describe the parallelisation and tasks
Granularity (What is Best?)

- The most efficient granularity depend on the algorithm and the hardware environment in which it runs.
- In most cases the overhead associated with communication and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
- Fine-grain parallelism can help reduce overheads due to load imbalance. Facilitates work balancing.
Partition the data

**Global addressable space**
- Allows the programmer to declare and “directly” access data potentially distributed in the computer

**Partitioned address space**
- Memory is logically partitioned between *local* and *remote* (a two-level hierarchy)
- Forces the programmer to pay attention to data locality, by exposing the inherent NUMA-ness of current architectures

![Diagram showing partitioned address space](image_url)
Computation is performed in multiple places.
A place contains data that can be operated on remotely.
Data lives in the place it was created, for its lifetime.

- A datum in one place may point to a datum in another place.
- Data-structures (e.g. arrays) may be distributed across many places.

A place expresses locality.
High Performance Fortran (HPF) – Now dead...

- Directives extend Fortran for distributed memory parallel programming
  - First definition early 1993, revision 1997
  - Japanese created additional features in JA-HPF

- Main features are **directives** for data mapping and parallel loops
  - Work performed where the data is stored
  - Some library routines

- Broad participation in standards effort
**HPF Example**

```hpf
!HPF$ 
DISTRIBUTE W ( BLOCK )
!HPF$
INDEPENDENT, NEW ( X ), REDUCTION ( SUM )

DO I = 1, N
  X = W(I) * (I - 0.5)
  SUM = SUM + F ( X )
END DO
```

* Team of processes execute entire program
* Loop iterations are distributed among processes based on data distribution
* Communication at end of loop to obtain global value SUM

* Each process has local segment of W
* Each process has its own copy of variable X
* Each process computes local value of SUM
* SUM updated at end of loop, result replicated

---

**Parallel Do**

**Forall**

**Independent**
What Happened to HPF?

- Compilers slow to arrive, and supported different styles of HPF programming
  - Based upon Fortran 90, also slow to mature
- Considered suitable for structured (regular) grids only
- MPI flexible and established by the time HPF compilers matured
  - Codified experience with early comms libraries
- Japanese vendors continued to add features and provide compilers after others gave up
Distribution of a shared array in UPC

- Elements are distributed in block-cyclic fashion
- Each thread "owns" blocks of adjacent elements

```
shared [2] int X[10];
```

```
shared [*] int X[10];
```

```
shared [1] int X[10];
shared int X[10];
```
Distribution of a shared array in UPC

shared [2] int X[10];

shared [*] int X[10];

shared [ ] int X[10];

shared [1] int X[10];
shared int X[10];

Logical Distribution

Physical Distribution
An example: particle-particle interaction

- A particle system has
  - a finite number of particles
  - moving in space according to e.g. Newton’s Laws (i.e. $F = ma$)
  - time is continuous

- Examples
  - stars in space with laws of gravity
  - atoms in a molecule with electrostatic forces
  - neutrons in a fission reactor
  - cars on a freeway with Newton’s laws plus model of driver and engine
Force on each particle can often be subdivided:

\[
\text{force} = \text{external\_force} + \text{nearby\_force} + \text{far\_field\_force}
\]

The far-field force is normally the tricky one for parallelization...
Parallelism in External Forces

- The force on each particle is independent
- “embarrassingly parallel”
- The work can often be statically and evenly distributed
- Evenly distribute particles on processors
  - Any distribution works
  - Locality is not an issue, no communication
Parallelism in Nearby Forces

- Nearby forces require interaction => communication
- Force may depend on other nearby particles
  - Ex: collisions
  - simplest algorithm is $O(n^2)$: look at all pairs to see if they collide
- Usual parallel model is domain decomposition of physical domain
- Challenge 1: interactions of particles near processor boundary
  - need to communicate particles near boundary to neighboring processors
  - surface to volume effect means low communication
  - Which communicates less: squares (as below) or slabs?
- Challenge 2: load imbalance, if particles cluster
  - galaxies, electrons hitting a device wall

Need to check for collisions between regions
Load balance via Tree Decomposition

- To reduce load imbalance, divide space unevenly
- Each region contains roughly equal number of particles
- Quad tree in 2D, Oct-tree in 3D

Example: each square contains at most 3 particles
Parallelism in Far-Field Forces

- Far-field forces involve all-to-all interaction => communication

- Force depends on all other particles
  - Ex: gravity
  - Simplest algorithm is $O(n^2)$
  - Just decomposing space does not help since every particle apparently needs to “visit” every other particle

- Use more clever algorithms to beat $O(n^2)$
Far-field forces: Particle-Mesh Methods

Superimpose a regular mesh
“Move” particles to nearest grid point
Exploit fact that far-field satisfies a PDE that is easy to solve on a regular mesh
- FFT, Multigrid
Accuracy depends on how fine the grid is and uniformity of particles
**Far-field forces: Tree Decomposition**

- Based on approximation
- $O(n \log n)$ or $O(n)$ instead of $O(n^2)$
- Forces from group of far-away particles “simplifies”
  - They resemble a single larger particle
- Use tree; each node contains an approximation of descendents
- Several Algorithms
  - Barnes-Hut
  - Fast Multipole Method (FMM) of Greengard/Rohklin
Current challenges in parallel programming
Current challenges in parallel programming
A Layered Programming Approach

- Domain Specific Languages, other means for application scientists to provide information
- Adapted versions of today’s portable parallel programming APIs (MPI, OpenMP, PGAS, Charm++)
- Maybe some non-portable low-level APIs (threads, CUDA, Verilog)
- Machine code, device-level interoperability stds, powerful runtime

Applications
New kinds of info
Familiar
Custom
Very low-level
Heterogeneous Hardware
More Dynamic Execution?

- What will the runtime (RT) environment look like? How dynamic will it be?
- Role of runtime system? Relationship between RT and OS, programming models? How is information exchanged?

Performance less predictable in dynamic execution environment