Table of Contents

Performance

- Latency Management
- Load Balancing
- Creating a High Degree of Parallelism

Performance - Memory Wall

Complex memory hierarchies greatly affect parallel execution.

Processing elements may share some components (e.g., L1/L2 caches, RAM), but usually not all.

Parallelism exacerbates the effects of memory latency.

- **Contention** from centralized components.
- **Non uniform latency** caused by distributed components.

Desktop Core2Duo
Private L1 Cache
Shared L2 Cache
Shared Centralized UMA

SGI Origin
Private L1 Cache
Private L2 Cache
Shared, Distributed NUMA

Linux Cluster
Private L1 Cache
Private L2 Cache
Private, Distributed NUMA

Performance - Memory Contention

The extent to which processes access the same location at the same time.

- Types of contention and mitigation approaches.
  - False sharing of cache lines.
    - Memory padding to cache block size.
  - 'Hot' memory banks.
    - Better interleaving of data structures on banks.
  - True Sharing.
    - Replication of data structure.
    - Locked refinement (i.e., distribution) for aggregate types.
- Most models do not directly address contention.

Performance - Managing Latency

There are two approaches to managing latency.

- Hiding - tolerate latency by overlapping a memory accesses with other computation.
  - User Level
  - Runtime System
- Reducing - minimize latency by having data near the computation that uses it.

Hiding Latency - User Level

Model has programming constructs that allow user to make asynchronous remote requests.

- Split-Phase Execution **(Charm++)**
  Remote requests contain address of return handler.

```
class A {
  foo() {
    B b;
    b.xyz(&A::bar());
  }
  bar(int x) { … }
};
```

```
class B {
  xyz(Return ret) {
    …
    ret(3);
  }
};
```

- Futures
  Remote requests create a handle that is later queried.

```
future<double> v(foo());   //thread spawned to execute foo()
…                          //do other unrelated work
double result = v.wait();  //get result of foo()
```

Hiding Latency - Runtime System

Runtime system uses extra parallelism made available to transparently hide latency.

e.g., Multithreading **(STAPL / ARMI)**

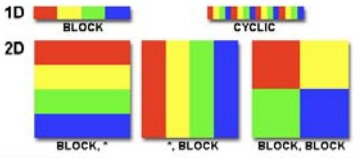
pRange can recursively divide work (based on user defined dependence graph) to increase degree of parallelism. ARMI splits and schedules work into multiple concurrent threads.

Performance - Latency Reduction

Data placement (HPF, STAPL, Chapel)

Use knowledge of algorithm access pattern to place all data for a computation near executing processor.

```
INTEGER, DIMENSION(1:16):: A,B
!HPF$ DISTRIBUTE(BLOCK) :: A
!HPF$ ALIGN WITH A :: B
```

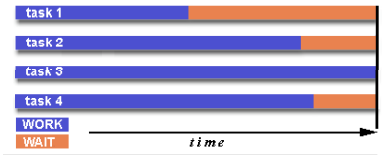


Work placement (STAPL, Charm++)

Migrate computation to processor near data and return final result. Natural in RMI based communication models.

Load Balancing

Keep all CPUs doing equal work.
Relies on good *work scheduling*.



- Static **(MPI)**
  Decide before execution how to distribute work.
- Dynamic **(Cilk, TBB)**
  Adjust work distribution during execution.
  - Requires finer work granularity (> 1 task per CPU)
    Some models change granularity as needed (minimize overhead).
  - Work Stealing
    Allow idle processors to 'steal' queued work from busier processors.

Enabling a High Degree of Parallelism

Parallel models must strive for a high degree of parallelism for maximum performance.

Makes transparent latency hiding easy.

Enables finer granularity needed for load balancing.

Table of Contents

Portability

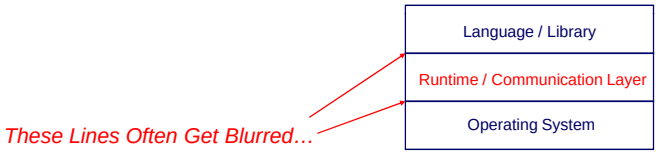
- Language versus Library
- Runtime System
  - Interchangeable
  - Virtualization
  - Load balancing
  - Reliance on specific machine features
- Effects of exposed machine model on portability
- I/O Support

Language versus Library

- Models with specialized language require a compiler to be ported and sometimes additional runtime support.
  - Cray's **Chapel**, **Titanium**, Sun's **Fortress**.
- Library approaches leverage standard toolchains, and often rely on widely available standardized components.
  - **STAPL** requires C++, Boost, and a communication subsystem (MPI, OpenMP Pthreads).
  - **MPI** requires communication layer interfaceand command wrappers (mpirun) to use portable versions (MPICH or LamMPI). Incremental customization can improve performance.

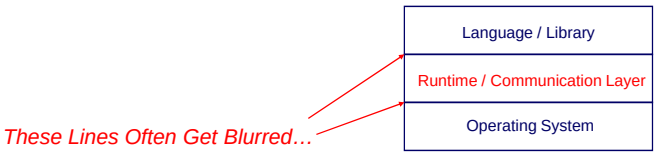
Runtime System

- **Interchangeable**
  Runtime system (e.g., threading and communication management) specific to model or is it modular?
- **Processor Virtualization**
  How are logical processes mapped to processors?
  Is it a 1:1 mapping or multiple processes per processor?



Runtime System

- **Load Balancing**
  Support for managing processor work imbalance?
  How is it implemented?
- **Reliance on Machine Features**
  Runtime system require specific hardware support?
  Can it optionally leverage hardware features?



Effects of Parallel Model

What effect does the model's level of abstraction have in mapping/porting to a new machine?

- Does it hide the hardware's model (e.g., memory consistency) or inherit some characteristics? Portability implications?
- Is there interface of machine characteristics for programmers? Optional use (i.e., performance tuning) or fundamental to code development?

Support for I/O

Some parallel models specifically address I/O, providing mechanisms that provide an abstract view to various disk subsystems.

ROMIO - *portable I/O extension included with MPI (Message Passing Interface).*

Table of Contents

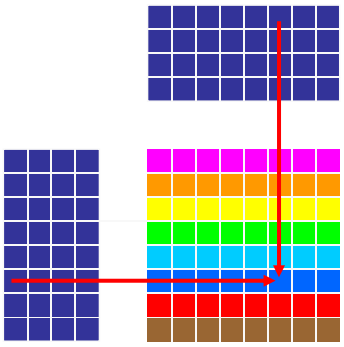
Shared Memory Programming

- Smaller scale parallelism (100's of CPUs/cores)
- Single system image
- Thread-based
- Threads have access to entire shared memory
  - Threads may also have private memory

Shared Memory Programming

- No explicit communication
  - Threads write/read shared data
  - Mutual exclusion used to ensure data consistency
- Explicit Synchronization
  - Ensure correct access order
  - E.g., don't read data until it has been written

Example - Matrix Multiply



```
for(int i=0; i<M; ++i) {
  for(int j=0; j<N; ++j) {
    for(int k=0; k<L; ++k) {
      C[i][j] +=
          A[i][k]*B[k][j];
    }
  }
}
```

One way to parallelize is to compute each row Independently.

Table of Contents

- Introduction to Parallelism
- Introduction to Programming Models
- Shared Memory Programming
  - OpenMP
  - pThreads
- Message Passing Programming
- Shared Memory Models
- PGAS Languages
- Other Programming Models

OpenMP

- Allows explicit parallelization of loops
  - Directives for Fortran and C/C++
  - Support for task parallelism

```
#pragma omp parallel for
for(int i=0; i<N; ++i) {
  C[i] = A[i] + B[i];
}
```

- Vendor standard
  - ANSI X3H5 standard in 1994 not adopted
  - OpenMP standard effort started in 1997
  - KAI first to implement new standard

Materials from http://www.llnl.gov/computing/tutorials/openMP/

The OpenMP Model

Execution Model

- Explicitly parallel
- Single-threaded view
- SPMD
- Implicit data distribution
- Nested parallelism support
- Relaxed consistency within parallel sections

The OpenMP Model

Productivity

- Provides directives for existing languages
- Low level of abstraction
- User level tunability
- Composability supported with nesting of critical sections and parallel loops

Performance

- Load balancing
  - optional selection of runtime scheduling policy
- Scalable parallelism
  - Parallelism proportional to data size

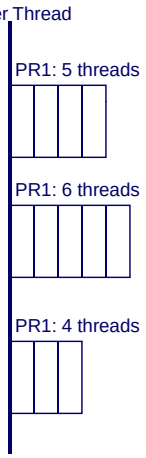
The OpenMP Model

Portability

- Directives allow use of available compilers
  - Application compiles and runs, but no parallelization
- Supports processor virtualization
  - N:1 mapping of logical processes to processors
- Load balancing
  - optional selection of runtime scheduling policy
- No reliance on system features
  - can utilize specialized hardware to implement Atomic update

OpenMP Thread Management

- Fork-Join execution model
- User or developer can specify thread count
  - Developer's specification has priority
  - Variable for each parallel region
  - Runtime system has default value
- Runtime system manages threads
  - User/developer specify thread count only
  - Threads "go away" at end of parallel region



OpenMP Thread Management

- Determining number of threads
  - omp_set_num_threads(int) function
  - OMP_NUM_THREADS environment variable
  - Runtime library default
- Threads created only for parallel sections

Creating Parallel Sections

- Parallel for

```
#pragma omp parallel for \
  shared(a,b,c,chunk) \
  private(i) \
  schedule(static,chunk)
for (i=0; i < n; i++)
  c[i] = a[i] + b[i];
```

- Options
  - Scheduling Policy
  - Data Scope Attributes

- Parallel region

```
#pragma omp parallel
{
  // Code to execute
}
```

- Options
  - Data Scope Attributes

Data Scope Attributes

Private	variables are private to each thread
First Private	variables are private and initialized with value of original object before parallel region
Last Private	variables are private and value from last loop iteration or section is copied to original object
Shared	variables shared by all threads in team
Default	specifies default scope for all variables in parallel region
Reduction	reduction performed on variable at end of parallel region
Copy in	assigns same value to variables declared as thread private