Component Transparency

- Opaque objects hide implementation details
  - raises level of abstraction
  - makes expansion difficult
- Transparent components
  - allow internal component reuse
  - example of working in programming model

```
int main() {
  pthread_t thread;
  pthread_attr_t attr;
  // …
}
```

```
template<class T>
class p_array : public p_container_indexed<T> {
  typedef p_container_indexed<T> base_type;
  size_t m_size;
  //…
};
```

Component Composition

Build a new component using building blocks.

```
template<typename View>
bool p_next_permutation(View& vw) {
  …
  reverse_view<View> rvw(vw);
  iter1 = p_adjacent_find(rvw);
  …
  iter2 = p_find_if(rvw, std::bind1st(pred,*iter1));
  …
  p_reverse(rvw);
  return true;
}
```

Programming Language

- Programming model language options:
  - provide a new language
  - extend an existing language
  - provide directives for an existing language
  - use an existing language

Fortress

```
component HelloWorld
  export Executable

  run()=do
    print "Hello, world!\n"
  end
end
```

Cilk

```
cilk void hello() {
  printf("Hello, world!\n");
}

int main() {
  spawn hello();
  sync;
}
```

Providing a new language

- Advantage
  - Complete control of level of abstraction
  - Parallel constructs embedded in language
- Disadvantage
  - Compiler required for every target platform
  - Developers must learn language

Fortress

```
component HelloWorld
  export Executable

  run()=do
    print "Hello, world!\n"
  end
end
```

Extending a language

- Advantage
  - Developers have less to learn
  - Complete control of level of abstraction
  - Parallel constructs embedded in syntax
- Disadvantage
  - Compiler required for every target system
  - Limited by constraints of base language

```
cilk void hello() {
  printf("Hello, world!\n");
}
int main() {
  spawn hello();
  sync;
}
```

Directives for a language

- Advantage
  - Developers have less to learn
  - Parallel constructs easily expressed in directives
  - Use available compilers if needed (no parallelization)
  - Specialized not necessarily needed on system
- Disadvantage
  - Compiler required for every target system
  - Higher levels of abstraction can't be achieved
  - Limited by constraints of base language
  - No composition

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

Library for a language

- Advantage
  - Developers learn only new API
  - Compilers available on more systems
- Disadvantage
  - Limited by constraints of base language

```
void* hello(void*) {
  printf("Hello, world!\n");
  pthread_exit(NULL);
}
```

```
int main() {
  pthread_t thread;
  pthread_attr_t attr;
  pthread_attr_init(&attr);

  pthread_create(&thread, &attr,
   hello, NULL);
}
```

Debuggable

Programming environments provide many options for debugging parallel applications.

Built-in	provides proprietary tools that utilize extra runtime information	Charm++
Tracing	provides hooks for tools to log state during execution	MPI, Charm++
Interoperability with standard tools	Leverage standard tools available on platform (e.g., gdb, totalview)	STAPL, TBB, Pthreads, MPI, OpenMP

Defect Management

- Reduce Defect Potential
  - Programming style reduces likelihood of errors
  - Use of container methods reduces out-of-bounds accesses

```
class tbb_work_function {
  void operator()(const blocked_range<size_t>& r) {
    for(size_t i = r.begin(); i != r.end(); ++i)
      C[i] = A[i]*B[i];
  }
};
```

- Provide Defect Detection
  - Components support options to detect errors at runtime
  - E.g., PTHREAD_MUTEX_ERRORCHECK enables detection of double-locking and unnecessary unlocking

Tunability

Programming environments support application optimization on a platform using:

- Performance Monitoring
  - Support measuring application metrics
- Implementation Refinement
  - Support for adaptive/automatic modification of application
  - Manual mechanisms provided to allow developer to implement refinement

Performance Monitoring

- Built-in support
  - Environment's components instrumented
  - Output of monitors enabled/disabled by developer
  - Components written by developer can use same instrumentation interfaces
- Interoperable with performance monitoring tools
  - Performance tools on a platform instrument binaries

Implementation Refinement

- Adjust implementation to improve performance
  - distribution of data in a container
  - scheduling of iterations to processors
- Adaptive/Automatic
  - Monitors performance and improves performance without developer intervention
  - Example: Algorithm selection in STAPL
- Manual mechanisms
  - Model provides methods to allow developer adjustment to improve performance
  - Example: Grain size specification to TBB algorithms

Machine Model

- Programming models differ in the amount and type of machine information available to user
  - TBB, Cilk, OpenMP: user unaware of number of threads
  - MPI: user required to write code as a function of the machine in order to manage data mapping
- Programming as a function of the machine
  - Lowers level of abstraction
  - Increases programming complexity

Interoperability with other models

- Projects would like to use multiple models
  - Use best fit for each application module
  - Modules need data from one another
- Models need flexible data placement requirements
  - Avoid copying data between modules
  - Copying is correct, but expensive
- Models need generic interfaces
  - Components can interact if interfaces meet requirements
  - Avoids inheriting complex hierarchy when designing new components

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)**

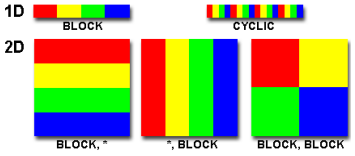
pRange can recursively divide work (based on user defined dependence graph) to increase degree of parallelism. ARMI splits and schedules work into multiple concurrent threads.

Performance - Latency Reduction

Data placement (HPF, STAPL, Chapel)

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

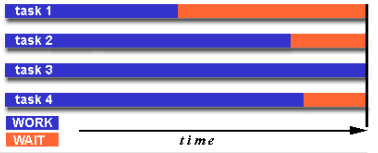
```
INTEGER, DIMENSION(1:16):: A,B
!HPF$ DISTRIBUTE(BLOCK) :: A
!HPF$ ALIGN WITH A :: B
```



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

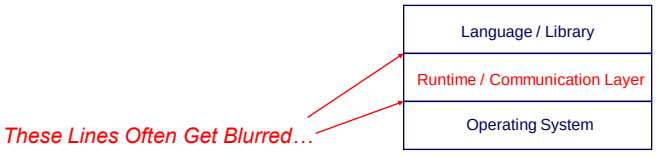
- Language versus Library
- Runtime System
  - Interchangeable
  - Virtualization
  - Load balancing
  - Reliance on specific machine features
- Effects of exposed machine model on portability
- I/O Support

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.

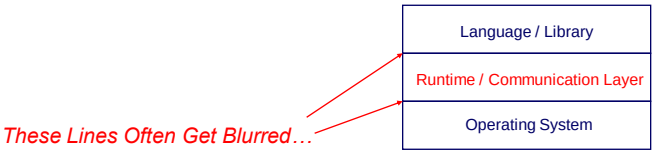
Runtime System

- **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 – EOL4