CS420 – Lecture 26

Marc Snir

Fall 2018

Parallel Programming Models and Systems

Parallel programming model: What are the parallel constructs and operations

Parallel programming systems: How are those expressed

A model (e.g., one-sided communication) can be expressed by libraries or languages
A system may support different models: E.g., OpenMP supports loop parallelism and task parallelism

Implicit Parallelism

Write sequential program, lean on compiler and hardware to execute in parallel.

Implicit Parallelism

Write sequential program, lean on compiler and hardware to execute in parallel.

Instruction level parallelism (ILP): Execute multiple instructions per cycle, while preserving dependencies in sequential code

- Compiler helps by ordering instructions so that many can execute simultaneously (e.g. by executing loads as soon as possible, and avoiding, if possible, dependencies between successive instructions
- Hardware helps by using register renaming to avoid "false dependencies" and by checking loads against the store queue

Implicit Parallelism

Write sequential program, lean on compiler and hardware to execute in parallel.

Instruction level parallelism (ILP): Execute multiple instructions per cycle, while preserving dependencies in sequential code

- Compiler helps by ordering instructions so that many can execute simultaneously (e.g. by executing loads as soon as possible, and avoiding, if possible, dependencies between successive instructions
- Hardware helps by using register renaming to avoid "false dependencies" and by checking loads against the store queue

Vectorization: Use instructions that perform vector operations

- Compiler helps by compiling simple loops into vector operations
- Hardware helps by providing vector registers and vector ALUs

Programmer's role

Programmer's role

- Write code that compiler can "understand" (facilitate alias analysis – e.g., "restrict")

Programmer's role

- Write code that compiler can "understand" (facilitate alias analysis – e.g., "restrict")
- Write code that is easy to vectorize

Programmer's role

- Write code that compiler can "understand" (facilitate alias analysis – e.g., "restrict")
- Write code that is easy to vectorize
- Write code that access data with good (temporal/spatial) locality

Programmer's role

- Write code that compiler can "understand" (facilitate alias analysis – e.g., "restrict")
- Write code that is easy to vectorize
- Write code that access data with good (temporal/spatial) locality
- Use appropriate compiler options

Programmer's role

- Write code that compiler can "understand" (facilitate alias analysis – e.g., "restrict")
- Write code that is easy to vectorize
- Write code that access data with good (temporal/spatial) locality
- Use appropriate compiler options
- Don't write your own code if a library is available

Shared Memory Explicit Parallelism

- Language vs. pragmas vs. overloading
- Extensions to existing language (C++, Java) vs. something totally different (Chapel)
- More declarative (what to compute) vs. more imperative (how to compute)
- Data-driven partitioning vs. control-driven partitioning

Vector Parallelism

Fortran code:

```
real A(10,20), B(10,20)
logical L(10,20)
 A = A + 1.0
A = SQRT(A)
L = A .EQ. B
A(1:7,3) = B(2:8,1)*B(4:10,1)

real A(20,20,2)
A(:,:,1) = 0.25*(CSHIFT(A(:,:,0),1,1)+CSHIFT(A(:,:,0),-1,1)
                        +CSHIFT(A(:,:,0),1,2)+CSHIFT(A(:,:,0),-1,2))

real A(20)
integer IDX(20)
A = A(IDX)
```

- Vector operations in language can be compiled into hardware vector instructions or SIMT code or multithreaded code: "Easiest" form of parallelism.

- Vector operations in language can be compiled into hardware vector instructions or SIMT code or multithreaded code: "Easiest" form of parallelism.
- Irregular code does not vectorize well

Explicit parallelism – Parallel Iterators

- Execute code for each element of a collection (STL container)
- Simplest: execute code for each element of a range
- Works best if it easy to split collection and iterates are independent, or are combined using a simple parallel algorithms, such as reduction

Java

```
double average = roster
 .parallelStream()
 .filter(p -> p.getGender() == Person.Sex.MALE)
 .mapToInt(Person::getAge)
 .average()
 .getAsDouble();
```

C++: `for_each`, `reduce`, `transform_reduce`, `copy_if` ...

Explicit Control parallelism – Multithreading

- Shared memory – all threads can access all variables (subject to scoping rules)
- *static multithreading:* A fixed number of threads coordinating through the use of shared variables – e.g. an OpenMP parallel section
- *dynamic multithreading:* A (possibly) fixed number of workers (called *threads* in OpenMP) are executing dynamically created tasks (or chunks of parallel iterates in OpenMP) – scheduler decides which worker executes which task,

Dynamic multithreading takes care of load balance

- Full language support (Java) vs. pragmas (OpenMP) vs. library (C++)
- More declarative (C++) or more imperative (OpenMP)

Main constructs

- Parallel iterator – Possibly nested
- Fork-join – Possibly nested
- Futures
- Divide&Conquer

Parallel iterator can be implemented as fork-join, but direct implementation can be much more efficient

Nested parallel loops and fork-join can generate irregular parallelism that stretches schedulers because of the need for global load balancing.

Full language support: e.g., Java

```
parallelFor  (lb, ub) .exec (new Loop()
   {
   // Thread-local variable declarations (optional)
    public void start()
   {
   // One-time thread-local initialization (optional method)
   }
   public void run (int i)
   {
   // Loop body code for iteration i (required method)
   }
   public void finish()
   {
   // One-time thread-local finalization (optional method)
   }
 });
```

Pragmas/Directives – E.g. OpenMP

- Less elegant than full language support
- Pragmatically, may encourage "sequential thinking"
- No fundamental differences

Library

E.g. Threads Building Blocks (TBB)
Sequential code:

```
void SerialApplyFoo( float a[], size_t n ) {
    for( size_t i=0; i!=n; ++i )
        Foo(a[i]);
}
```

Parallel loop 1

Programmer creates code that can execute a chunk of iterates

```
class ApplyFoo {
    float *const my_a;
public:
    void operator()( const blocked_range<size_t>& r ) const {
        float *a = my_a;
        for( size_t i=r.begin(); i!=r.end(); ++i )
           Foo(a[i]);
    }
    ApplyFoo( float a[] ) :
        my_a(a)
    {}
};
```

The function `parallel_for` is passed the entire iteration range and the newly defined class. `parallel_for` will chunk the iteration range and allocate the chunks to threads

```
void ParallelApplyFoo( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n), ApplyFoo(a));
}
```

- No special compiler needed – only need to link to the TBB runtime library
- Code might be less efficient since compiler does not "understand" parallelism
- Syntax is more burdensome

Futures & Remote Procedure Call (RPC)

RPC: a function call that can be executed on a separate thread

- Is RPC executed synchronously or asynchronously?
- Does it return a result?
- What is the execution context?

C++ future is an asynchronous RPC that (usually) returns a value and has its own context (passed arguments and captured references)

Divide&Conquer

- If problem is small then solve problem sequentially
- Else
  - Divide problem into (two) independent subproblems
  - Solve two subproblems in parallel
  - Combine two subproblem solutions

Quicksort: Divide&conquer sorting algorithm where all work is done in the divide part

Mergesort: Divide&conquer sorting algorithm where all work is done in the combine part

Divide&Conquer logic can be implemented using recursive fork-join

Divide&Conquer in TBB

```
void ParallelQuicksort( T* begin, T* end ) {
 if( end-begin>=SMALL ) {
  using namespace std;
// Divide Code
   T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
   swap( *begin, mid[-1] );
// Recursion
  tbb::parallel_invoke( [=]{ParallelQuicksort( begin, mid-1 );},
      [=]{ParallelQuicksort( mid, end );} );
  } else {
// leaf code
   SerialQuicksort( begin, end );
   }
// empty combine code
}
```

TBB implements loop parallelism using the divide&conquer pattern.

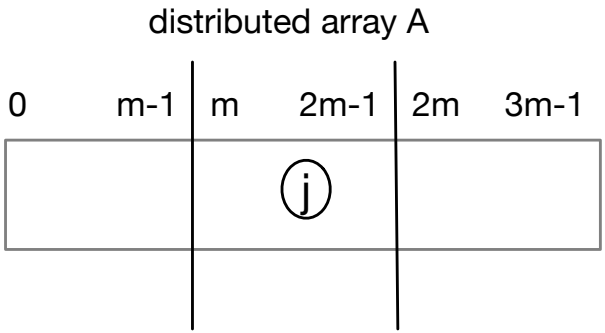
Distributed memory parallelism – E.g., MPI, Shmem

- *Static multiprocessing + distributed memory:* A fixed number of processes, each with its own address space.
- Each process can only access its address space
- Explicit calls are used to communicate across processes

  1-sided: Put/get
  2-sided: Send/receive
  Collective: broadcast/reduce
- MPI supports well send-receive and collectives; 1-sided is a later addition
- Shmem (Openshmem) supports well 1-sided

PGAS

People like the convenience of a global address space: Being able to access any entry in a large (distributed) array using global "names"



`x=A[j]` vs. `x = Get(proc=sizeof(A)/numproc, disp= sizeof(A)%numproc)`

PGAS model:

- Shared and private segments
- Global and local references

Solution

Example: Co-array Fortran (CAF) (now part of Fortran 2008)

```
real A(100) ! regular, local array
real B(100)[*] ! coarray: each process has an array with 100 entries
real C
C = B(15) ! access to the local copy of B -- regular load
C = B(15)[7] ! access to a possibly remote copy-- may need a get()
A(:) = B(:)[3] ! copy the entire array from a possibly remote process
```

Code is executed by each process

Example: Universal Parallel C (UPC)

```
shared float a[10][5]; // a is a global array distributed round-robin
float b; // local variable
b = a[3][2]; // access to a possibly remote entry
```

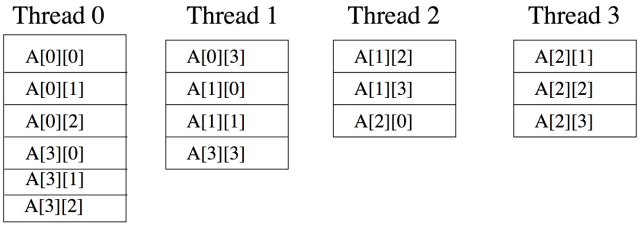
Code is executed at each process (called “thread” in UPC)

User can control the chunk size in the distribution of a global array

Assume THREADS = 4

```
shared [3] int A[4][THREADS];
```

will result in the following data layout:

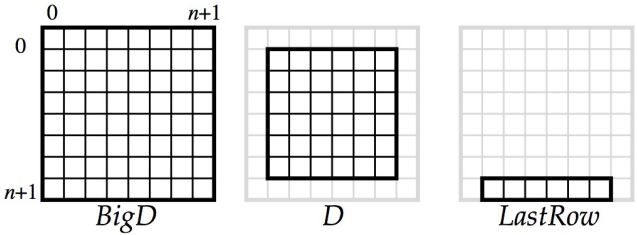


Partitioned Global Adress Space (PGAS)

- UPC pointers can be local (address) or global (rank, address)
- Dereferencing a global pointer is expensive (get/put)
- Global pointer arithmetic in UPC is expensive (less so in CAF or in UPC++)
- Hard to remember what is local and what is global in UPC (less so in CAF or UPC++)

Example Chapel – sequential

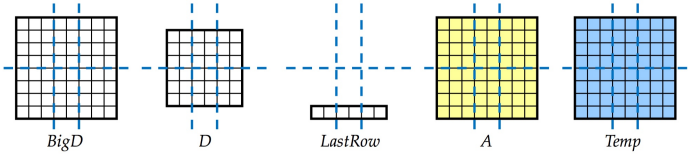
```
const BigD = {0..n+1, 0..n+1},
      D = BigD[1..n, 1..n],
      LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;
do {
  forall (i,j) in D do
     Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;
     const delta = max reduce abs(A[D] - Temp[D]);
     A[D] = Temp[D];
} while (delta > epsilon);
```



Chapel also supports sparse, irregular domains

Chapel parallel

```
const BigD = {0..n+1, 0..n+1} dmapped Block({1..n, 1..n}),
          D = BigD[1..n, 1..n],
          LastRow = D.exterior(1,0);
....
```



- The execution of the code is partitioned using the owner compute rule (data parallelism)
- Chapel's compiler compiles the codes to be executed on each process and the required communication (ghost cell updates)

A classification of distributed memory programming models

Data
- local – each process has its own address space
- global – shared arrays can be accessed by all processes

Control
- local – code is executed by each process
- global – there is logically one thread of execution; compiler distributes code execution to the processes

		Control	
		local	global
Data	local	MPI	?
	global	UPC	Chapel

Yet More Choice: Remorte Procedure Call

Some distributed memory libraries/languages (UPC++, X10, Charm++) support injecting calls into remote processes; the calls execute in the context of the remote process.

- Makes sense for a model with static, persistent threads, each with their own private data environment.

Charm++

- Basic Object is *Chare*: Contains data and external methods that can be called by other chares; the methods do not return values.
- Chares are distributed across compute nodes; there are typically multiple chares per node, and they can be migrated.

Charm++ Hello World

Header file (hello.h)

```
class Hello : public CBase_Hello {
  public:
  Hello(); // C++ constructor

  void sayHi(int from); // Remotely invocable "entry method"
};
```

Charm++ Interface file (hello.ci)

```
module hello {
  array [1D] Hello {
    entry Hello();
    entry void sayHi(int);
  };
};
```

Source file (hello.cpp)

```
#include "hello.decl.h"
#include "hello.h"

extern CProxy_Main mainProxy;
extern int numElements;

Hello::Hello() {
  // No member variables to initialize in this simple example
}
```

```
void Hello::sayHi(int from) {

  // Have this chare object say hello to the user.
  CkPrintf("Hello␣from␣chare␣#␣%d␣on␣processor␣%d␣(told␣by␣%d)\n",
  thisIndex, CkMyPe(), from);

  // Tell the next chare object in this array of chare objects
  // to also say hello. If this is the last chare object in
  // the array of chare objects, then tell the main chare
  // object to exit the program.
  if (thisIndex < (numElements - 1)) {
    thisProxy[thisIndex + 1].sayHi(thisIndex);
  } else {
    mainProxy.done();
  }
}

#include "hello.def.h"
```

Frameworks

- Have a fixed (simple) template of how parallelism is effected
- User plugs into the framework its own methods

Example: map-reduce
Example: Pregel (and many others) for graph analytics

Pregel

- Input is directed graph with values at vertices and edges
- Computation consists of a sequence of "supersteps"
- At each superstep each vertex can read values at incoming edges, update its own values, update values at outgoing vertices and send messages to other vertices (typically, its neighbors)
- Messages sent by a vertex at superstep s is accessed by other vertices at superstep $s + 1$ (bulk synchronous parallel programming model). Same for edge updates.

Pregel API

```
template <typename VertexValue,
typename EdgeValue,
typename MessageValue>
class Vertex {
  public:
  virtual void Compute(MessageIterator* msgs) = 0;
  const string& vertex_id() const;
  int64 superstep() const;
  const VertexValue& GetValue();
  VertexValue* MutableValue();
  OutEdgeIterator GetOutEdgeIterator();
  void SendMessageTo(const string& dest_vertex,
  const MessageValue& message);
  void VoteToHalt();
};
```

User only needs to overwrite the `Compute()` method, and magic happens.

How many programming systems?

The Holy Grail: One system used for mutithreading. distributed memory and accelerators (GPUs)

- Possible to achive only with high-level frameworks that are descriptive

MPI+X: One system for distributed memory and one for multithreading and GPUs

- E.g., MPI + OpenMP

MPI+X+Y: One system for distributed memory, one for mutithreading and one for GPU

- E.g., MPI + C++ + Cuda

- Using programming models adjusted to each hardware system can lead to better performance
- It also leads to code that is hardware to develop and maintain

- There is much more
  - Evolving features of MPI, OpenMP
  - New evolving general languages
  - Domain specific languages (DSL)
  - Frameworks & libraries
- Problem
  - Dominant systems (MPI, OpenMP) are inelegant but well-supported
  - Emerging systems are more elegant, but less well supported and may disappear