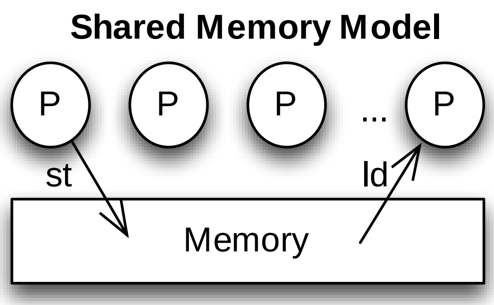
Chapter 2: Perspectives on Parallel Programming

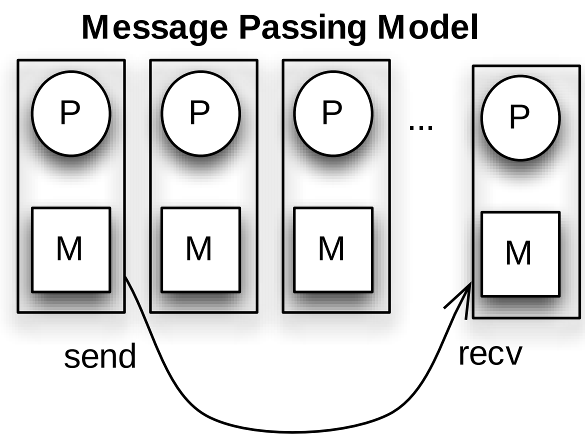
Module 2.1
Parallel Programming Models

Programming Models

- What is programming model?
  - An abstraction provided by the hardware to programmers
  - Determines how easy/difficult for programmers to express their algorithms into computation tasks that the hardware understands
- Uniprocessor programming model
  - Based on program + data
  - Bundled in Instruction Set Architecture (ISA)
  - Highly successful in hiding hardware from programmers
- Multiprocessor programming model
  - Much debate, still searching for the right one…
  - Most popular: shared memory and message passing

Shared Mem vs. Msg Passing





- Shared Memory / Shared Address Space
  - Each memory location visible to all processors
- Message Passing
  - Each memory location visible to 1 processor

Thread/process – Uniproc analogy

Process: share nothing

```
if (fork() == 0)
    printf("I am the child process, my id is %d", getpid());
else
    printf("I am the parent process, my id is %d", getpid());
```

-heavyweight => high thread creation overhead
-The processes share nothing => explicit communication using socket, file, or messages

Thread: share everything

```
void sayhello() {
  printf("I am child thread, my id is %d", getpid());
}

printf("I am the parent thread, my id is %d", getpid());
clone(&sayhello,<stackarg>,<flags>,())
```

+ lightweight => small thread creation overhead
+ The processes share addr space => implicit communication

Thread communication analogy

```
int a, b, signal;
…
void dosum(<args>) {
  while (signal == 0) {}; // wait until instructed to work
  printf("child thread> sum is %d", a + b);
  signal = 0;  // my work is done
}

void main() {
  a = 5, b = 3;
  signal = 0;
  clone(&dosum,…)         // spawn child thread
  signal = 1;             // tell child to work
  while (signal == 1) {}  // wait until child done
  printf("all done, exiting\n");
}
```

- Shared memory in multiproc provides similar memory sharing abstraction

Message Passing Example

```
Int a, b;
…
void dosum() {
  recvMsg(mainID, &a, &b);
  printf("child process> sum is %d", a + b);
}

Void main() {
  if (fork() == 0)  // I am the child process
    dosum();
  else {            // I am the parent process
    a = 5, b = 3;
    sendMsg(childID, a, b);
    wait(childID);
    printf("all done, exiting\n");
  }
}
```

Differences with shared memory:
- Explicit communication
- Message send and receive provide automatic synchronization

Quantitative Comparison

Table 2.1: Comparing shared memory and message passing programming models.

Aspects	**Shared Memory**	**Message Passing**
Communication	implicit (via loads/stores)	explicit messages
Synchronization	explicit	implicit (via messages)
Hardware support	typically required	none
Development effort	lower	higher
Tuning effort	higher	lower

Development vs. Tuning Effort

- Easier to develop shared memory programs
  - Transparent data layout
  - Transparent communication between processors
  - Code structure little changed
  - Parallelizing compiler, directive-driven compiler help
- Harder to tune shared memory programs for scalability
  - Data layout must be tuned
  - Communication pattern must be tuned
  - Machine topology matters for performance

More Shared Memory Example

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
sum = 0;
for (i=0; i<8; i++)
  if (a[i] > 0)
    sum = sum + a[i];
Print sum;
```

+ Communication directly through memory.
+ Requires less code modification
- Requires privatization prior to parallel execution

```
begin parallel // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter=4;
shared double sum=0.0, a[], b[], c[];
shared lock_type mylock;

start_iter = getid() * local_iter;
end_iter = start_iter + local_iter;
for (i=start_iter; i<end_iter; i++)
  a[i] = b[i] + c[i];
barrier;

for (i=start_iter; i<end_iter; i++)
  if (a[i] > 0) {
    lock(mylock);
      sum = sum + a[i];
    unlock(mylock);
  }
barrier;    // necessary

end parallel // kill the child thread
Print sum;
```

More Message Passing Example

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
sum = 0;
for (i=0; i<8; i++)
  if (a[i] > 0)
    sum = sum + a[i];
Print sum;
```

+ Communication only through messages
- Message sending and receiving overhead
- Requires algo and program modifications

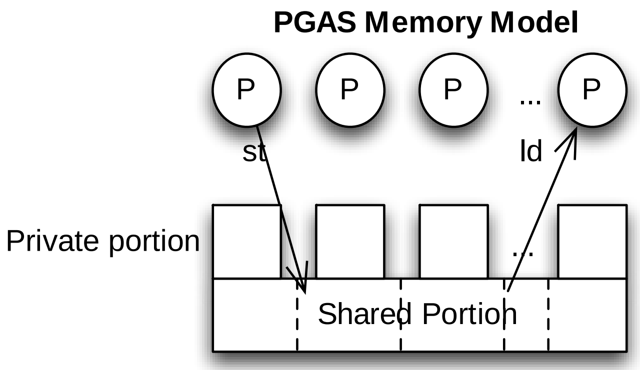
```
// parent and child already spawned
id = getpid();
local_iter = 4;
start_iter = id * local_iter;
end_iter = start_iter + local_iter;

if (id == 0)
  send_msg (P1, b[4..7], c[4..7]);
else
  recv_msg (P0, &b[4..7], &c[4..7]);

for (i=start_iter; i<end_iter; i++)
  a[i] = b[i] + c[i];

local_sum = 0;
for (i=start_iter; i<end_iter; i++)
  if (a[i] > 0)
    local_sum = local_sum + a[i];
if (id == 0) {
  recv_msg (P1, &local_sum1);
  sum = local_sum + local_sum1;
  Print sum;
}
else
  send_msg (P0, local_sum);
```

Other Programming Models

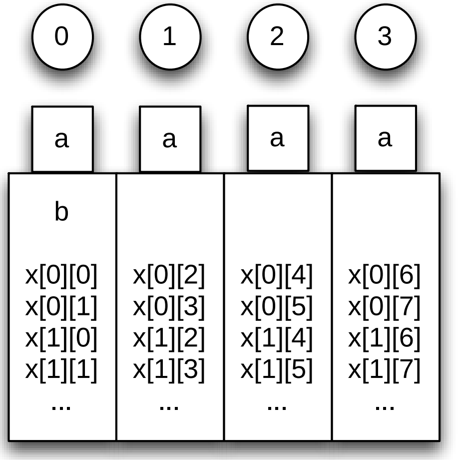
- PGAS
  - Partitioned Global Address Space
- Data Parallel
- MapReduce
- Transactional Memory

PGAS

- Shared memory model too simple for NUMA
  - Data layout is hidden from programmers
  - But thread-data proximity is important for performance
- PGAS provides shared & private address space




```
int a;
shared int b;
shared [2] double x[8][8];
```



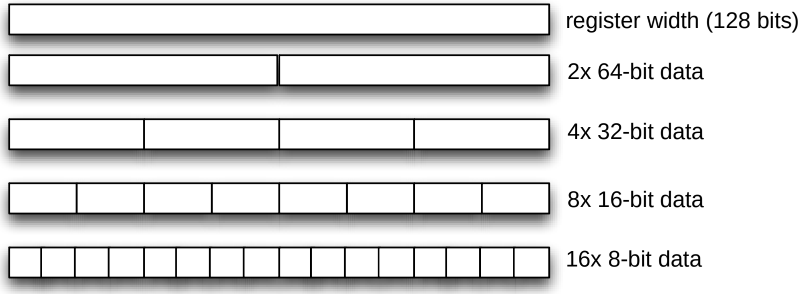
Example

```
shared [N*P /THREADS] double A[N][P]; // distribute rows of matrix A
shared [N*P /THREADS] double C[N][M]; // distribute rows of matrix C
shared [M /THREADS] double B[P][M] ; // distribute columns of matrix B

void main(void) {
  ... // matrices are initialized

  upc_forall(i=0;i<N;i++;&A[i][0]
    for (j=0; j<M; j++) {
      C[i][j] = 0;
      for(l=0; l< P; l++)
        C[i][j] +=A[i][l]*B[l][j];
   }
  upc_barrier;

  ...
}
```

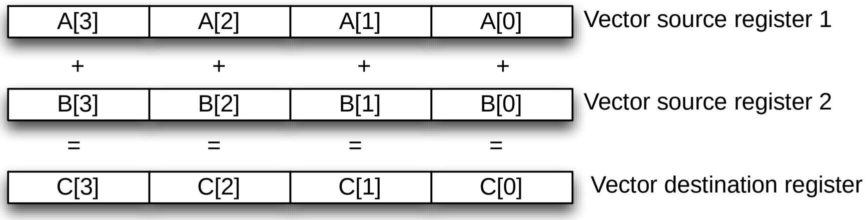
- Every node has N*P/THREADS rows of A and C
- &A[i][0] in upc_forall assigns iteration i to thread where A[i][0] is located

Data Parallel Model

- Data parallel = programming model for SIMD
- Either vector or scalar with lanes
  - Requires packing data into a wide register



  - Once packed, can compute multiple data items at once

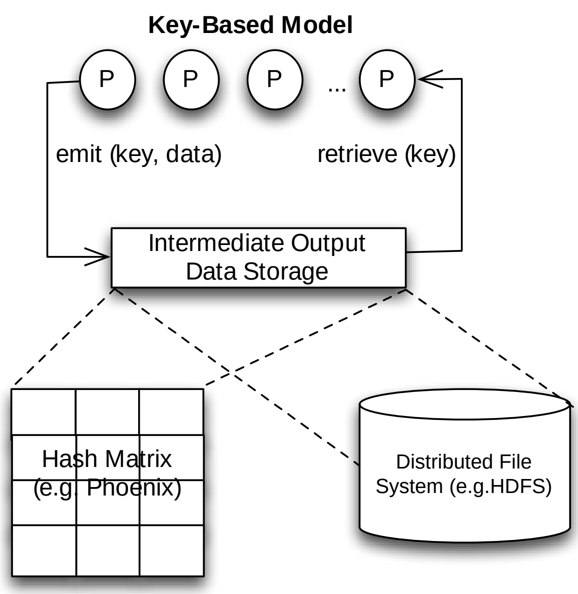


Example

```
// A 128-bit vector struct with four 32-bit floats
struct Vector4
{
  float x, y, z, w;
};

// Add two constant vectors and return the resulting vector
Vector4 SSE_Add ( const Vector4 &Operand1, const Vector4 &Operand2
{
  Vector4 Result;

  __asm
  {
    MOV EAX Operand1       // Load pointers into CPU regs
    MOV EBX, Operand2

    MOVUPS XMM0, [EAX]     // Move unaligned vectors to SSE regs
    MOVUPS XMM1, [EBX]

    ADDPS XMM0, XMM1       // Vector addition
    MOVUPS [Result], XMM0 // Save the return vector
  }
  return Result;
}
```

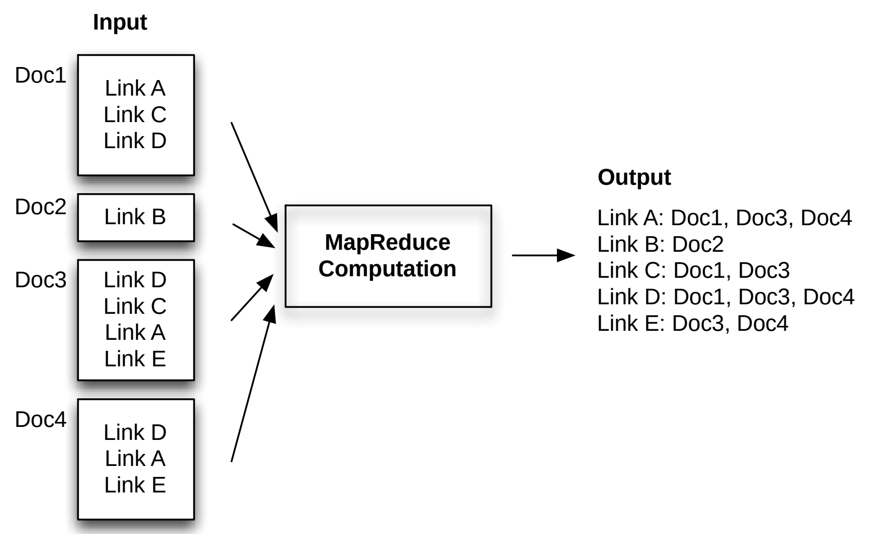
MapReduce

- Data accessed through key rather than location
  - Can be implemented on shared memory or message passing models
- Two steps:
  - Map (produce <key,val> pairs) and
  - Reduce (aggregate values for each key)



Example: Inverted Index

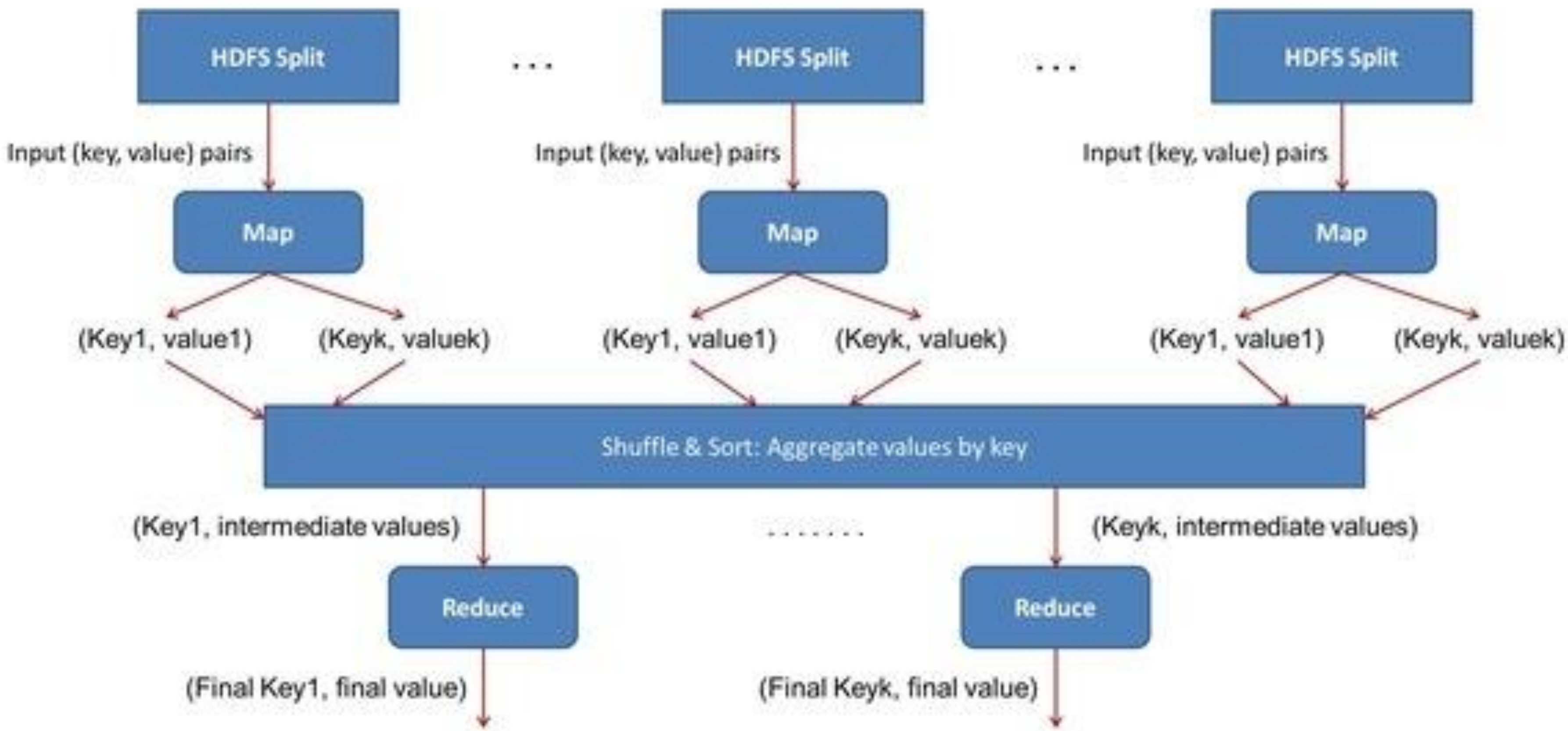
- Files distributed over map workers
- Each map worker produces <LinkX,DocY> when it encounters LinkX on DocY
- Reduce worker aggregates all Docs having the same Link



Hadoop MapReduce (More Detail)

- Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.
- A MapReduce *job* usually splits the input data-set into independent chunks which are processed by the *map tasks* in a completely parallel manner.
  - The framework sorts the outputs of the maps, which are then input to the *reduce tasks*.
  - Typically both the input and the output of the job are stored in a file-system.
  - The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.
- Typically the compute nodes and the storage nodes are the same, that is, the MapReduce framework and the Hadoop Distributed File System (see HDFS Architecture Guide) are running on the same set of nodes.
  - This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster.
- The MapReduce framework consists of a single master JobTracker and one slave TaskTracker per cluster-node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master.

Hadoop MapReduce (More Detail)

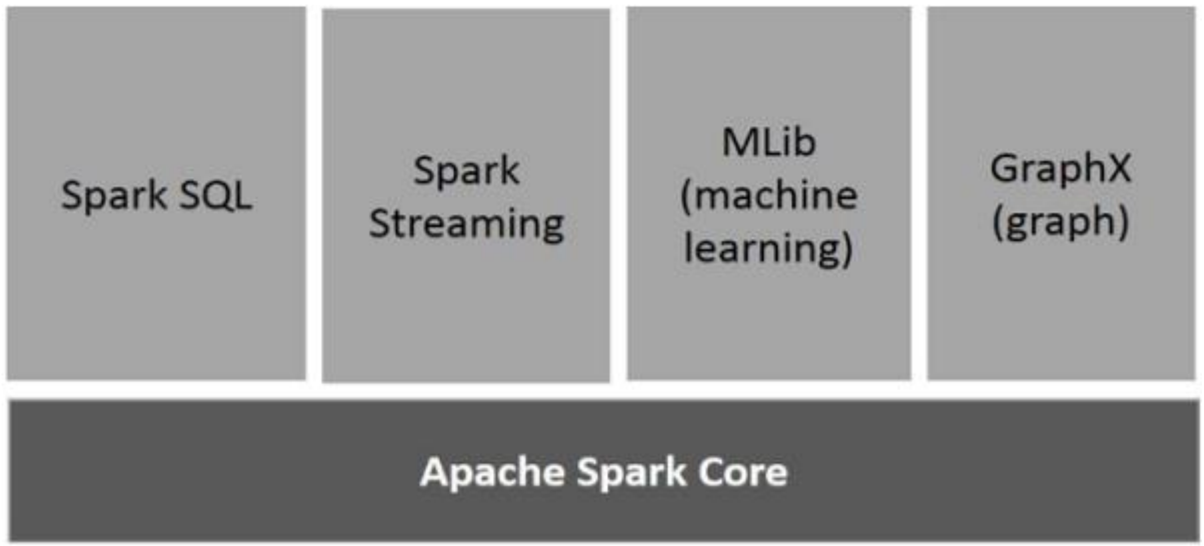


Tutorial: https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

Spark

- It's also a top-level (newer) Apache project focused on processing data in parallel across a cluster, but the biggest difference is that it works in-memory.
- Spark handles work in a similar way to Hadoop, except that computations are carried out in memory and stored there, until the user actively persists them.
  - Initially, Spark reads from a file on HDFS, S3, or another filestore, into an established mechanism called the SparkContext.
  - Out of that context, Spark creates a structure called an RDD, or Resilient Distributed Dataset, which represents an immutable collection of elements that can be operated on in parallel.
  - As the RDD and related actions are being created, Spark also creates a DAG, or Directed Acyclic Graph, to visualize the order of operations and the relationship between the operations in the DAG.
- Spark has been found to run 100 times faster in-memory, and 10 times faster on disk. It's also been used to sort 100 TB of data 3 times faster than Hadoop MapReduce on one-tenth of the machines.
  - Spark performance, as measured by processing speed, has been found to be optimal over Hadoop
    - Spark is not bound by input-output concerns every time it runs a selected part of a MapReduce task. It's proven to be much faster for applications
    - Spark's DAGs enable optimizations between steps. Hadoop doesn't have any cyclical connection between MapReduce steps, meaning no performance tuning can occur at that level.

Components of Spark



Transactional Memory

- Transaction = code region with ACID property (Atomicity, Consistency, Isolation, Durability)
- Transactional Memory (TM) adopts Atomicity and Isolation
  - It allows us to remove some locks and worries over deadlocks

Lock Example

```
image = Image_Read(fn_input);

for (i=0; i<image->row; i++) {
  for (j=0; j<image->col; j++) {
    lock();
    histoRed[image->red[i][j]]++;
    histoGreen[image->green[i][j]]++;
    histoBlue[image->blue[i][j]]++;
    unlock();
  }
}
```

```
image = Image_Read(fn_input);
lock_t redLock[256], greenLock[256], blueLock[256];

for (i=0; i<image->row; i++) {
  for (j=0; j<image->col; j++) {
    lock(&redLock(image->red[i][j]));
    histoRed[image->red[i][j]]++;
    unlock(&redLock(image->red[i][j]));

    lock(&greenLock(image->red[i][j]));
    histoGreen[image->red[i][j]]++;
    unlock(&greenLock(image->red[i][j]));

    lock(&blueLock(image->red[i][j]));
    histoBlue[image->blue[i][j]]++;
    unlock(&blueLock(image->red[i][j]));
  }
}
```

TM Example

```
image = Image_Read(fn_input);

for (i=0; i<image->row; i++) {
  for (j=0; j<image->col; j++) {
    atomic {
      histoRed[image->red[i][j]]++;
      histoGreen[image->green[i][j]]++;
      histoBlue[image->blue[i][j]]++;
    }
  }
}
```

- Transaction enclosed by atomic {…}
- Hardware or software ensures atomicity and isolation
  - SW inflexible and slow
  - HW expensive
- No locks needed and no locking overheads are incurred

Closing Comments

- Many programming models out there
- Most are based on shared memory or message passing, and build on top of them
- Trade offs between control vs. complexity

- Overall, parallel programs still require a lot of tuning despite the programming model used

Module Review Questions

- What are two basic parallel programming models?
- What are key advantages shared memory model have over message passing model?
- What primitives are necessary for supporting shared memory parallel programming?
- In what way Transactional Memory simplify parallel programming?

Module Review Questions

- What are two basic parallel programming models?
  - Shared memory and message passing
- What are key advantages and disadvantages shared memory model have over message passing model?
  - Pluses: implicit communication, lower development effort, finer communication, Minuses: explicit synchronization, higher tuning effort, requires hardware support
- What primitives are necessary for supporting shared memory parallel programming?
  - Variable scope (shared vs. private), synchronization primitives
- In what way Transactional Memory simplify parallel programming?
  - Higher abstraction (simpler coding and reasoning), removing lock-related problems