Lecture 5
Concurrent Programming

Dr. Wilson Rivera

ICOM 4036: Programming Languages

Electrical and Computer Engineering Department
University of Puerto Rico

Some slides adapted from Sebesta's textbook

Lecture Outline

- Definitions related to concurrency
- Introduction to parallel programming
  - Flynn's Taxonomy (parallel architectures)
  - Memory access models
  - Parallel programming models
- Implementation (Notations) Tour
  - Threading: OpenMP
  - Data Parallel: CUDA
  - Task Parallel: TBB
  - Message Passing: MPI, Erlang

Concurrency

- Instruction level
  - e.g. Pipelining, superscalar execution, out-of-order execution
- Statement level
  - Data Oriented
- Unit level (subprograms)
  - Tasks Oriented
    - Heavyweight task
      - Executes in its own address space (e.g. Ada Threads)
    - Lightweight task
      - All tasks run in the same address space (e.g. Java Threads, C# Threads)

Concurrency: A condition of a system in which multiple
tasks are logically active at one time.

Parallelism: A condition of a system in which multiple
tasks are actually active at one time.

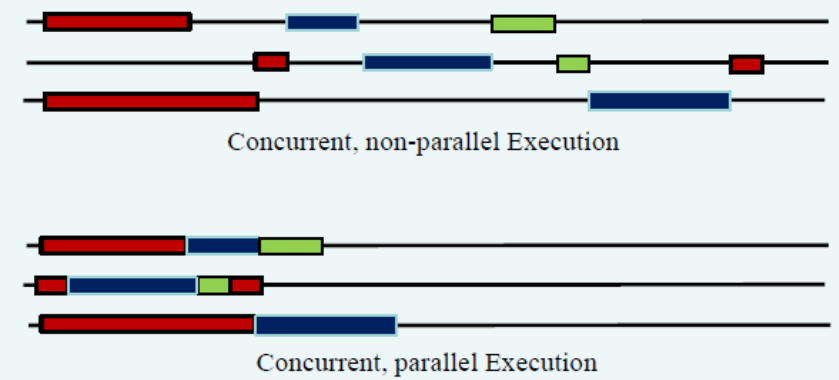
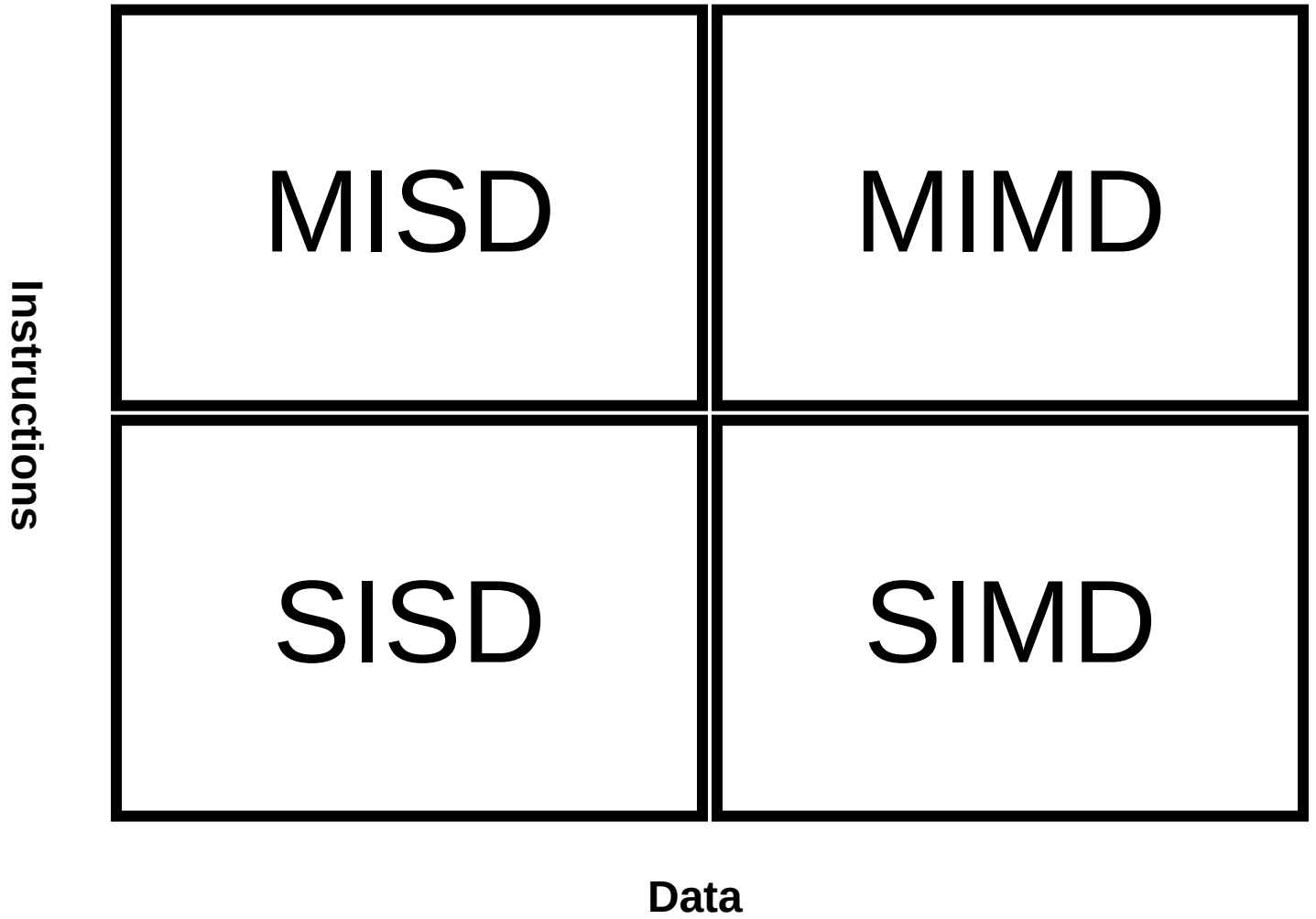


Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

Intro to Parallel Programming

- **Parallel computing** refers to the simultaneous use of multiple compute resources to solve a computational problem.
  - Flynn's Taxonomy (parallel architectures)
  - Memory access models
  - Parallel programming models

Flynn's Taxonomy



Flynn's Taxonomy

- **Single Instruction, Single Data (SISD)**
  - A serial (non-parallel) computer where only one instruction stream is being executed by the CPU during a clock cycle
  - Only one data stream is being used as input during a clock cycle

Flynn's Taxonomy

- **Multiple Instruction, Single Data (MISD)**
  - A single data stream is fed into multiple processing units
  - Each processing unit operates on the data via independent instruction streams.
  - Non commercial examples of this type of parallel computer architecture have ever existed.

Flynn's Taxonomy

- **Single Instruction, Multiple Data (SIMD)**
  - All processing units execute the same instruction at any given clock cycle
  - Each processing unit can operate on a different data element
  - This type of computer architecture is best suited for specialized problems characterized by a high degree of regularity
    - Image processing
  - Processor Arrays
    - Connection Machine CM-2, MasPar, ILLIAC IV
  - Vector Pipelines
    - IBM 9000, Cray X-MP, Cray Y-MP, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10
  - Most modern computers, particularly those with graphics processing units (GPUs) employ SIMD execution units.

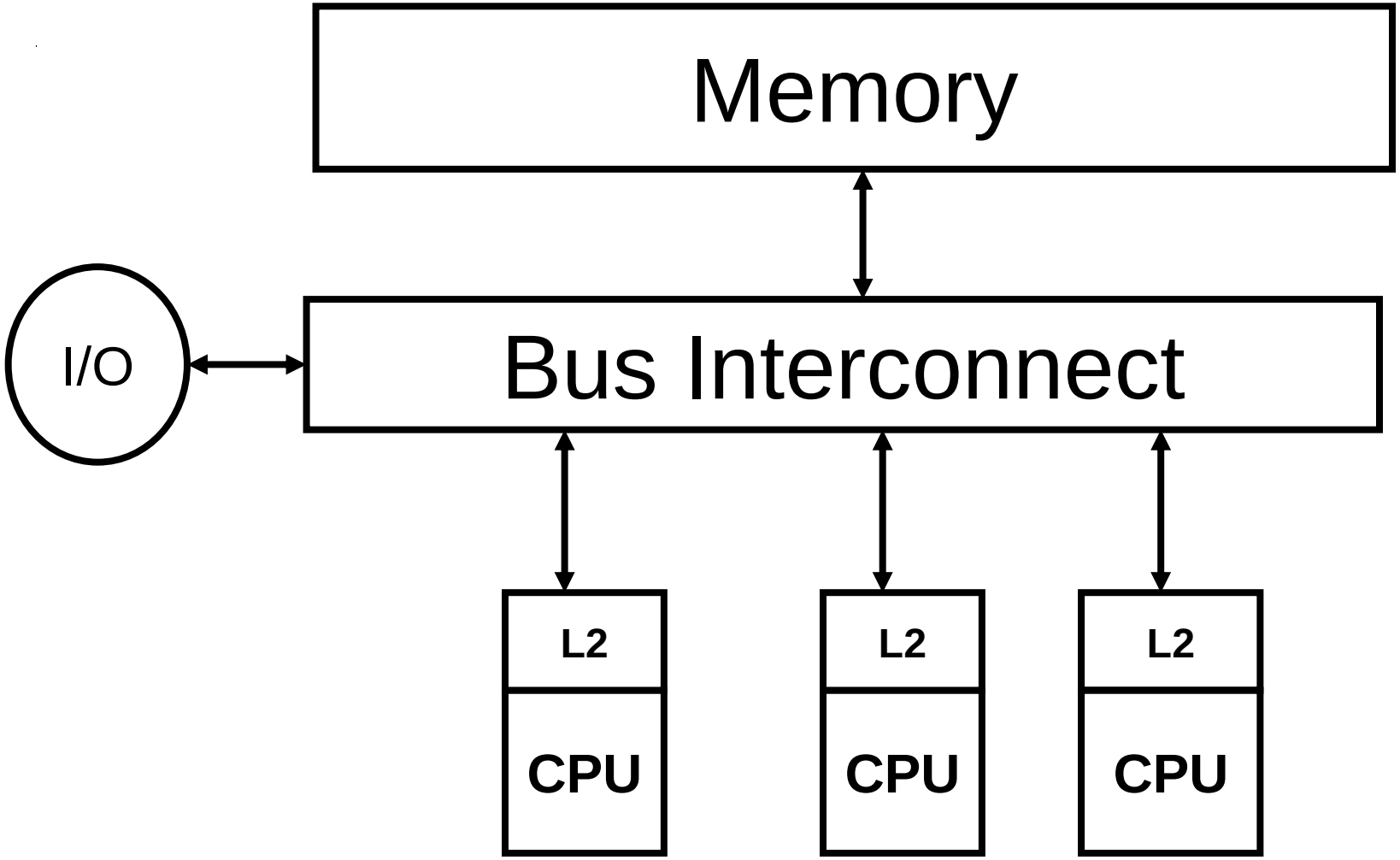
Flynn's Taxonomy

- **Multiple Instruction, Multiple Data (MIMD)**
  - Every processor may be executing a different instruction stream, and working with a different data stream.
  - Examples of this parallel computer architecture include most current clusters, and multicore computers.
    - In practice MIMD architectures may also include SIMD execution sub-components.

Memory Access Models

- Shared memory
- Distributed memory
- Hybrid Distributed-Shared Memory

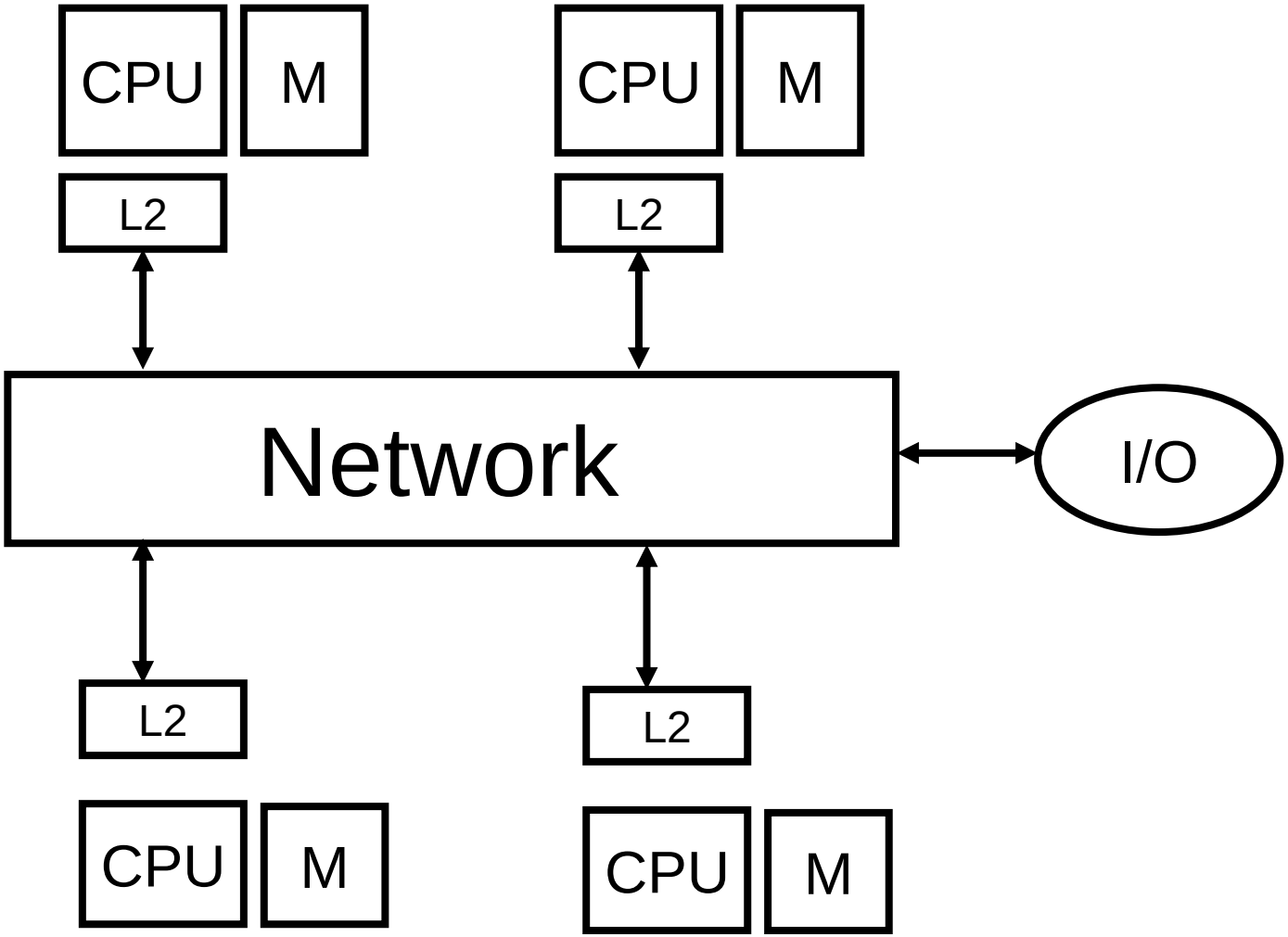
Shared Memory



Shared Memory

- multiple processors can operate independently but share the same memory resources
  - so that changes in a memory location effected by one processor are visible to all other processors.
- Two main classes based upon memory access times
  - **Uniform Memory Access** (UMA), commonly represented by Symmetric Multiprocessor (SMP) machines, where identical processors have equal access and access times to memory, and
  - **Non Uniform Memory Access** (NUMA), commonly built by physically linking two or more SMPs. In this case one SMP can directly access memory of another SMP.
- Main disadvantage is the lack of scalability between memory and CPUs.
  - Adding more CPUs geometrically increases traffic on the shared memory CPU path, and for cache coherent systems, it increases traffic associated with cache and memory management

Distributed Memory



Distributed Memory

- Processors have their own local memory. When a processor needs access to data in another processor
  - it is usually the task of the programmer to explicitly define how and when data is communicated

Hybrid Distributed-Shared Memory

- This type of parallel computers is often built by networking multiple SMPs.
- Each SMP knows only about its own memory, and network communications are required to move data from one SMP to another.

Parallel Programming Language Models

- Threads Model
- Data Parallel Model
- Task Parallel Model
- Message Passing Model

Threads Model

- A single process can have multiple, concurrent execution paths
  - The thread based model requires the programmer to manually manage synchronization, load balancing, and locality, which in turn requires detailed understanding of the underlying hardware.
  - With the number of threads growing as the number of cores does, dealing with bugs caused by deadlocks and race conditions reduces developer productivity significantly.
- POSIX Threads
  - specified by the IEEE POSIX 1003.1c standard (1995) and commonly referred to as Pthreads,
  - Library-based explicit parallelism model.
- OpenMP
  - Compiler directive-based model that supports parallel programming in C/C++ and FORTRAN.

Data Parallel Model

- Each task works on a different partition of the same data structure and each task performs the same operation on their partition of data.
  - On shared memory architectures, all tasks may have access to the data structure through global memory.
  - On distributed memory architectures the data structure is split up and resides as chunks in the local memory of each task.
  - Data parallel model implementations in general perform computation on arrays of data using array operators, and communications using array shift or rearrangement operators.
  - This model is suitable for problems with static load balancing (e.g. very regular fluid element analysis, image processing).
  - It is easy to debugging, as there is only one copy of code executing in a highly synchronized fashion.

- Implementations
  - FORTRAN 90 (F90), FORTRAN 95 (F95)
  - High Performance Fortran (HPF)
  - RapidMind, Ct, MapReduce, and CUDA

Task Parallel Model

- Each processor executes a different thread or process on the same or different data, and each thread may execute the same or different code.
- Implementations
  - Threading Building Blocks (TBB)
  - Task Parallel Library (TPL)
  - Intel Concurrent Collections (CnC)

Message Passing Model

- A set of tasks use their own local memory during computation
  - Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines
  - Tasks exchange data through communications by sending and receiving messages
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.
  - From a programming perspective, message passing implementations commonly comprise a library of subroutines that are embedded into source code.
- Implementations
  - MPI
  - Erlang

OPENMP

OpenMP

- http://www.openmp.org
- http://www.cOMPunity.org
- Open Multi Processing

- An Application Program Interface (API) that may be used to explicitly direct *multi-threaded, shared memory parallelism*
- Comprised of three primary API components:
  - Compiler Directives
  - Library Routines
  - Environment Variables
- Portable:
  - The API is specified for C/C++ and Fortran
  - Multiple platforms have been implemented
  - OpenMP 3.1 (2011)
  - OpenMP 4.0 (coming November 2012)

OpenMP

- OpenMP Directives
  - Parallel construct
  - Work sharing constructs
  - Synchronization constructs
- Run-time Library Routines
- Environment Variables

Parallel Construct

```
#include <omp.h>

main ()
{  int nthreads, tid, n;

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel if (n>10,000) \
                        private (tid)
    {

             /* Obtain and print thread id */
            tid = omp_get_thread_num();
            printf("Hello World from thread = %d\n", tid);

             /* Only master thread does this */
             if (tid == 0)
             {
                 nthreads = omp_get_num_threads();
                 printf("Number of threads = %d\n", nthreads);
             }

  }/* All threads join master thread and terminate */

}
```

Parallel Construct

#pragma omp parallel *[clause ...] newline*

if *(scalar_expression)*
private *(list)*
shared *(list)*
default (shared | none)
firstprivate *(list)*
reduction *(operator: list)*
copyin *(list)*

structured_block

Only Fortran API supports default(private)

Parallel Construct

- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team.
- There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  - omp_set_num_threads() library function
  - OMP_NUM_THREADS environment variable
  - Implementation default

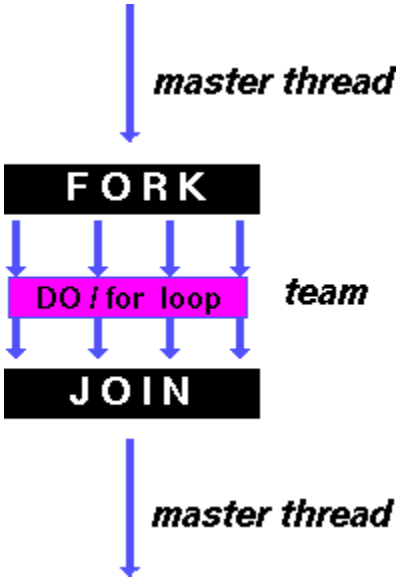
- Threads are numbered from 0 (master thread) to N-1

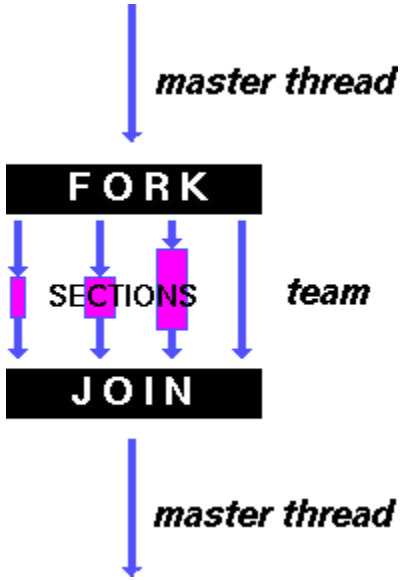
Parallel Construct

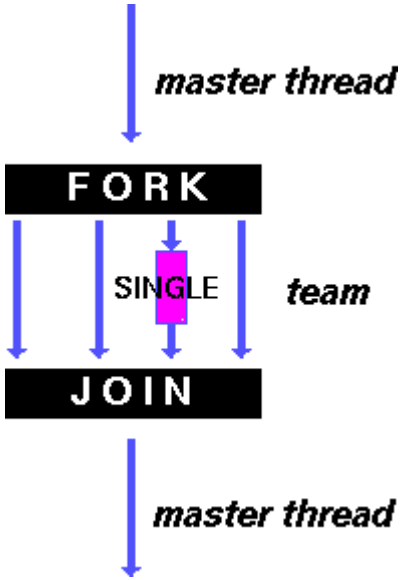
- Dynamic Threads
  - By default, a program with multiple parallel regions will use the same number of threads to execute each region. This behavior can be changed to allow the run-time system to dynamically adjust the number of threads that are created for a given parallel section.
    - Use of the omp_set_dynamic() library function
    - Setting of the OMP_DYNAMIC environment variable
      - » setenv OMP_DYNAMIC TRUE
- Nested Parallel Regions
  - A parallel region nested within another parallel region results in the creation of a new team, consisting of one thread, by default.
- Restrictions
  - A parallel region must be a structured block that does not span multiple routines or code files
  - It is illegal to branch into or out of a parallel region

Work Sharing Constructs

- FOR
  - Shares iterations of a loop across the team. Represents a type of "data parallelism".
- SECTIONS
  - Breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".
- SINGLE
  - Serializes a section of code







Work Sharing: FOR Directive

```
double doc_product ( int n, double x[], double y[] )
{
       int i;
       double xdoty;

       xdoty = 0.0;

     # pragma omp parallel Shared ( n, x, y )private ( i ) \
       reduction ( + : xdoty )
     {
     # pragma omp for

       for ( i = 0; i < n; i++ )
       {
         xdoty = xdoty + x[i] * y[i];
       }

       return xdoty;
     }
}
```

Work Sharing: FOR Directive

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000

main ()
{ int i, chunk;
  float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
 chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
       {
       #pragma omp for schedule (dynamic,chunk) nowait
       for (i=0; i < N; i++)
       c[i] = a[i] + b[i];
       } /* end of parallel section */
}
```

Work Sharing: FOR Directive

#pragma omp for *[clause ...]*

schedule *(type [,chunk])*
ordered
private *(list)*
firstprivate *(list)*
lastprivate *(list)*
shared *(list)*
reduction *(operator: list)*
nowait

for_loop

Schedule

- **Static**
  - Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion
  - In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads
  - Done at compilation time
- **Dynamic**
  - Fixed portions of work; size is controlled by the value of chunk
  - When a thread finishes, it starts on the next portion of work
  - Done at run time
- **Guided**
  - Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially
- **Runtime**
  - The compiler (or runtime system) decides what is best to use

Work Sharing: SECTIONS Directive

```
#include <omp.h>
#define N 1000

main ()
{ int i; float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
a[i] = b[i] = i * 1.0;

#pragma omp parallel shared(a,b,c) private(i)
     {
          #pragma omp sections nowait
          {
          #pragma omp section
          for (i=0; i < N/2; i++)
          c[i] = a[i] + b[i];

          #pragma omp section
          for (i=N/2; i < N; i++)
          c[i] = a[i] * b[i];
          } /* end of sections */ }

     /* end of parallel section */
}
```

Work Sharing: SECTIONS Directive

```
#pragma omp sections [clause ...]
                    private (list)
                    firstprivate (list)
                    lastprivate (list)
                    reduction (operator: list) nowait

    {

    #pragma omp section

        structured_block

    #pragma omp section

        structured_block

    }
```

Work Sharing: SINGLE Directive

#pragma omp single *[clause ...]*

private *(list)*

firstprivate *(list)*

nowait

structured_block

Parallel For

```
#pragma omp parallel for default(none) \
        shared(m,n,a,b,c) private(i,j)
   for (i=0; i<m; i++)
   {
      a[i] = 0.0;
      for (j=0; j<n; j++)
         a[i] += b[i*n+j]*c[j];
   } /*-- End of omp parallel for --*/
}
```

OpenMP: Quicksort

```
void quick_sort(double *data, int initialIndex, int finalIndex)
{
   if(initialIndex < finalIndex)
   {
        int mid = partition_data(data, initialIndex, finalIndex);

        #pragma omp parallel sections
        {
                #pragma omp section
                {       quick_sort(data,initialIndex, mid-1);
                }
                #pragma omp section
             {
                        quick_sort(data, mid+1, finalIndex);
             }
        }
    }
}
```

Synchronization Constructs

- MASTER Directive
- CRITICAL Directive
- BARRIER Directive
- ATOMIC Directive
- FLUSH Directive
- ORDERED Directive

MASTER Directive

- The MASTER directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code
  - There is no implied barrier associated with this directive

```
#pragma omp master
    structured_block
```

BARRIER Directive

- When a BARRIER directive is used, a thread will wait at that point until all other threads have reached that barrier.

- Should be used If data is updated asynchronously and data integrity is at risk
  - Between parts in the code that read and write the same section of memory
  - After one iteration in a solver

- Unfortunately, barriers tend to be expensive and also may not scale to a large number of processors. Therefore, use them with care

```
#pragma omp barrier
```

Master and Barrier

```
#pragma omp parallel
{
    do_many_things();
#pragma omp master
        { exchange_boundaries(); }
#pragma omp barrier
    do_many_other_things();
}
```

CRITICAL Directive

- The CRITICAL directive specifies a region of code that must be executed by only one thread at a time.
  - If a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.
  - The optional name enables multiple different CRITICAL regions to exist:
    - Names act as global identifiers. Different CRITICAL regions with the same name are treated as the same region.
    - All CRITICAL sections which are unnamed, are treated as the same section.

#pragma omp critical *[name]*
structured_block

OpenMP example: Critical

```
#pragma omp parallel
  {
    #pragma omp single
      printf("Number of threads is %d\n",omp_get_num_threads());
  }

  sum = SUM_INIT;
  printf("Value of sum prior to parallel region: %d\n",sum);

  #pragma omp parallel default(none) shared(n,a,sum) \
        private(TID,sumLocal)
  {
    TID = omp_get_thread_num();
    sumLocal = 0;

    #pragma omp for
     for (i=0; i<n; i++)
       sumLocal += a[i];

    #pragma omp critical (update_sum)
    {
     sum += sumLocal;
     printf("TID=%d: sumLocal = %d sum = %d\n",TID,sumLocal,sum);
    }
  } /*-- End of parallel region --*/
```

Tasking in OpenMP

```
my_pointer = listhead;
#pragma omp parallel
{ #pragma omp single nowait
        { while(my_pointer) {
                #pragma omp task firstprivate(my_pointer)
                {
                    do_independent_work (my_pointer);
                }
                my_pointer = my_pointer->next ;}
        } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier
```

Tasking in OpenMP

```
int main(int argc, char *argv[]) {
#pragma omp parallel
{
        #pragma omp single
        {
                #pragma omp task
                {printf("Hello ");}
                #pragma omp task
                {printf("World ");}
                #pragma omp taskwait
                printf("\nThank You ");
        }
} // End of parallel region
}
```

Run-Time Library Routines

- OMP_SET_NUM_THREADS
- OMP_GET_NUM_THREADS
- OMP_GET_MAX_THREADS
- OMP_GET_THREAD_NUM
- OMP_GET_NUM_PROCS
- OMP_IN_PARALLEL
- OMP_SET_DYNAMIC
- OMP_GET_DYNAMIC
- OMP_SET_NESTED
- OMP_GET_NESTED
- OMP_INIT_LOCK
- OMP_DESTROY_LOCK
- OMP_SET_LOCK
- OMP_UNSET_LOCK
- OMP_TEST_LOCK
- OMP_GET_WTIME
- OMP_GET_WTICK

OMP_SET_DYNAMIC

void omp_set_dynamic(int dynamic_threads)

- Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions.
- The default setting is implementation dependent.
- Must be called from a serial section of the program.

OMP_SET_NUM_THREADS

void omp_set_num_threads(int num_threads)

- The dynamic threads mechanism modifies the effect of this routine.
  - Enabled: specifies the maximum number of threads that can be used for any parallel region by the dynamic threads mechanism.
  - Disabled: specifies exact number of threads to use until next call to this routine.
- This routine can only be called from the serial portions of the code

OMP_GET_WTIME

double omp_get_wtime(void)

- Provides a portable wall clock timing routine
- Returns a double-precision floating point value equal to the number of elapsed seconds since some point in the past.
  - Usually used in "pairs" with the value of the first call subtracted from the value of the second call to obtain the elapsed time for a block of code.
- Designed to be "per thread" times, and therefore may not be globally consistent across all threads in a team
  - depends upon what a thread is doing compared to other threads.

Environment Variables

- OMP_SCHEDULE
  - Applies only to for and parallel for (C/C++) directives which have their schedule clause set to RUNTIME.
  - The value of this variable determines how iterations of the loop are scheduled on processors.
    - setenv OMP_SCHEDULE "guided, 4"
    - setenv OMP_SCHEDULE "dynamic"
- OMP_NUM_THREADS
  - Sets the maximum number of threads to use during execution.
    - setenv OMP_NUM_THREADS 8
- OMP_DYNAMIC
  - Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.
  - Valid values are TRUE or FALSE.
    - setenv OMP_DYNAMIC TRUE

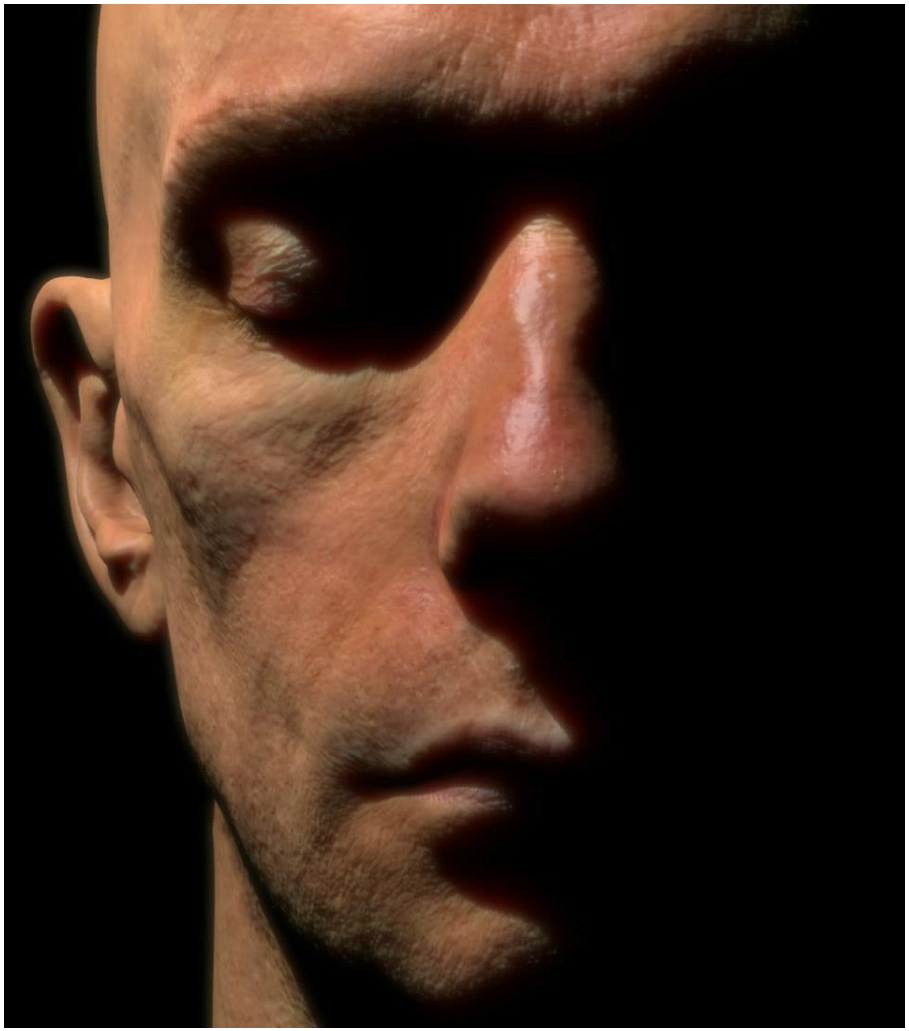
Performance considerations

- **Be aware of the Amdahl's law**
  - Minimize serial code
  - Remove dependencies among iterations
- **Balance the load**
  - Experiment with using `SCHEDULE` clause
- **Be aware of directives cost**
  - Parallelize outer loops
  - Minimize the number of directives
  - Minimize synchronization – minimize the use of `BARRIER`, `CRITICAL`, `ORDERED`
  - Consider using `NOWAIT` clause of `OMP DO` when enclosing several loops inside one `PARALLEL` region.
  - Merge loops to reduce synchronization cost
- **Reduce false sharing**
  - Use private variables
- **Try task level parallelism**

CUDA

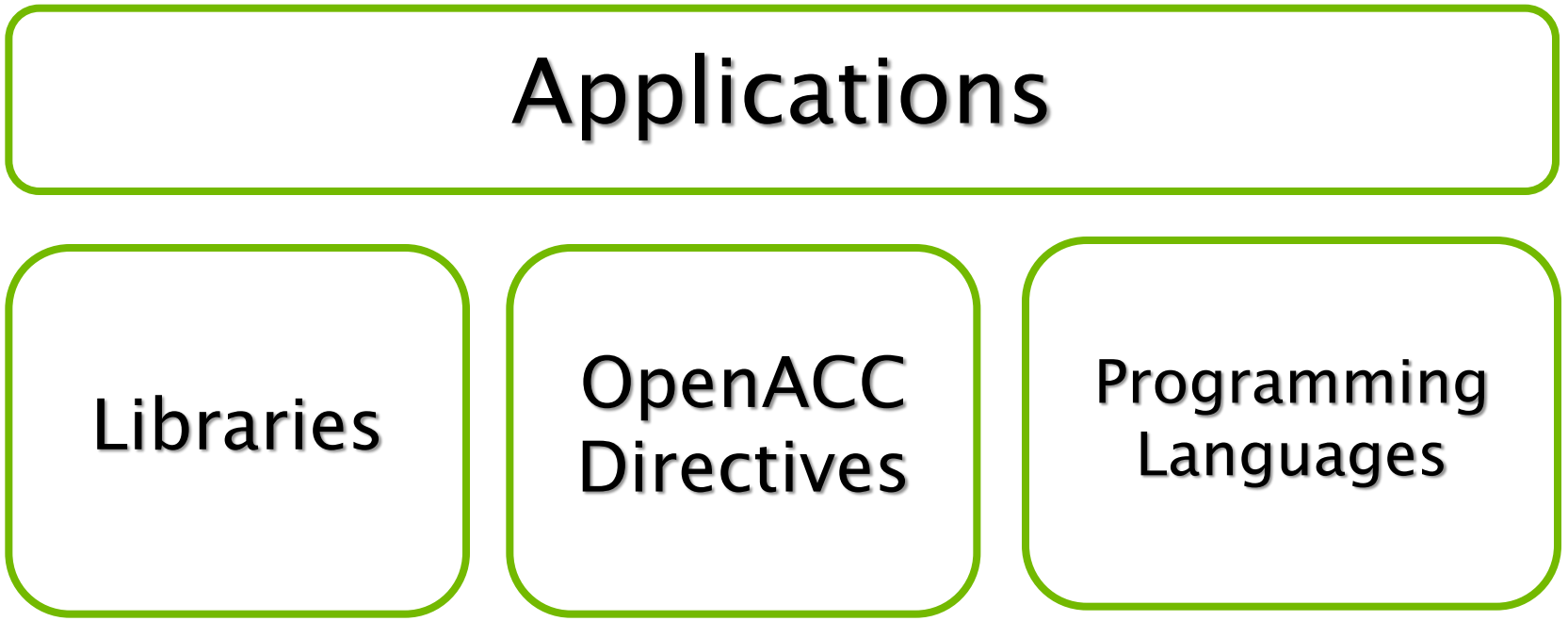
CUDA

- Compute Unified Device Architecture
  - Designed and developed by NVIDIA
  - Data parallel programming interface to GPUs
    - Requires an NVIDIA GPU (GeForce, Tesla, Quadro)



Eugene d'Eon, David Luebke, Eric Enderton
In *Proc. EGSR 2007* and *GPU Gems 3*

3 Ways to Accelerate Applications



GPU Programming Languages

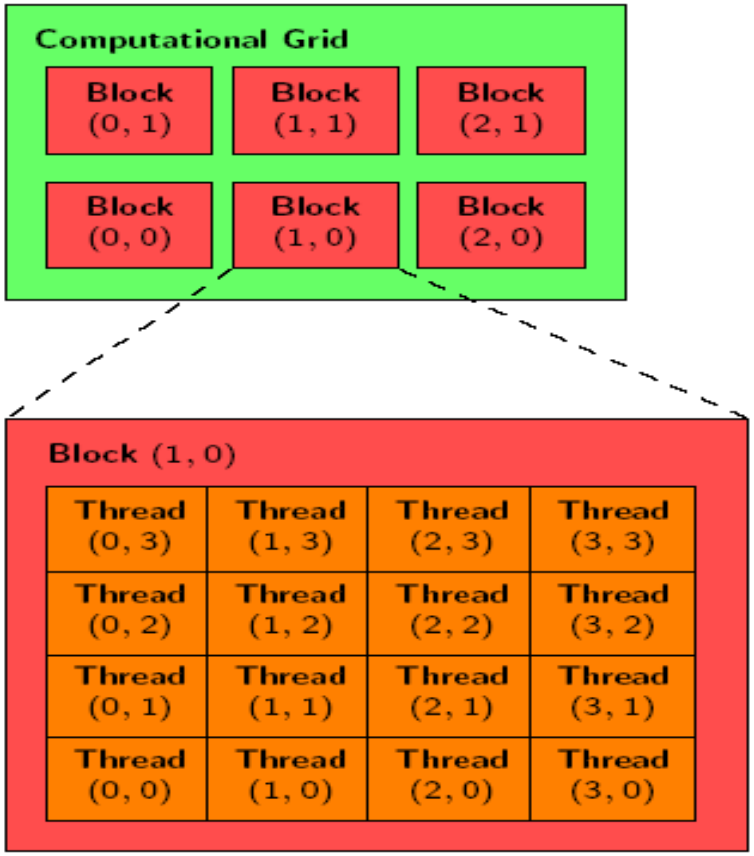
Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	OpenACC, CUDA Fortran
C	OpenACC, CUDA C
C++	Thrust, CUDA C++
Python	PyCUDA, Copperhead
F#	Alea.cuBase

GPU Threads

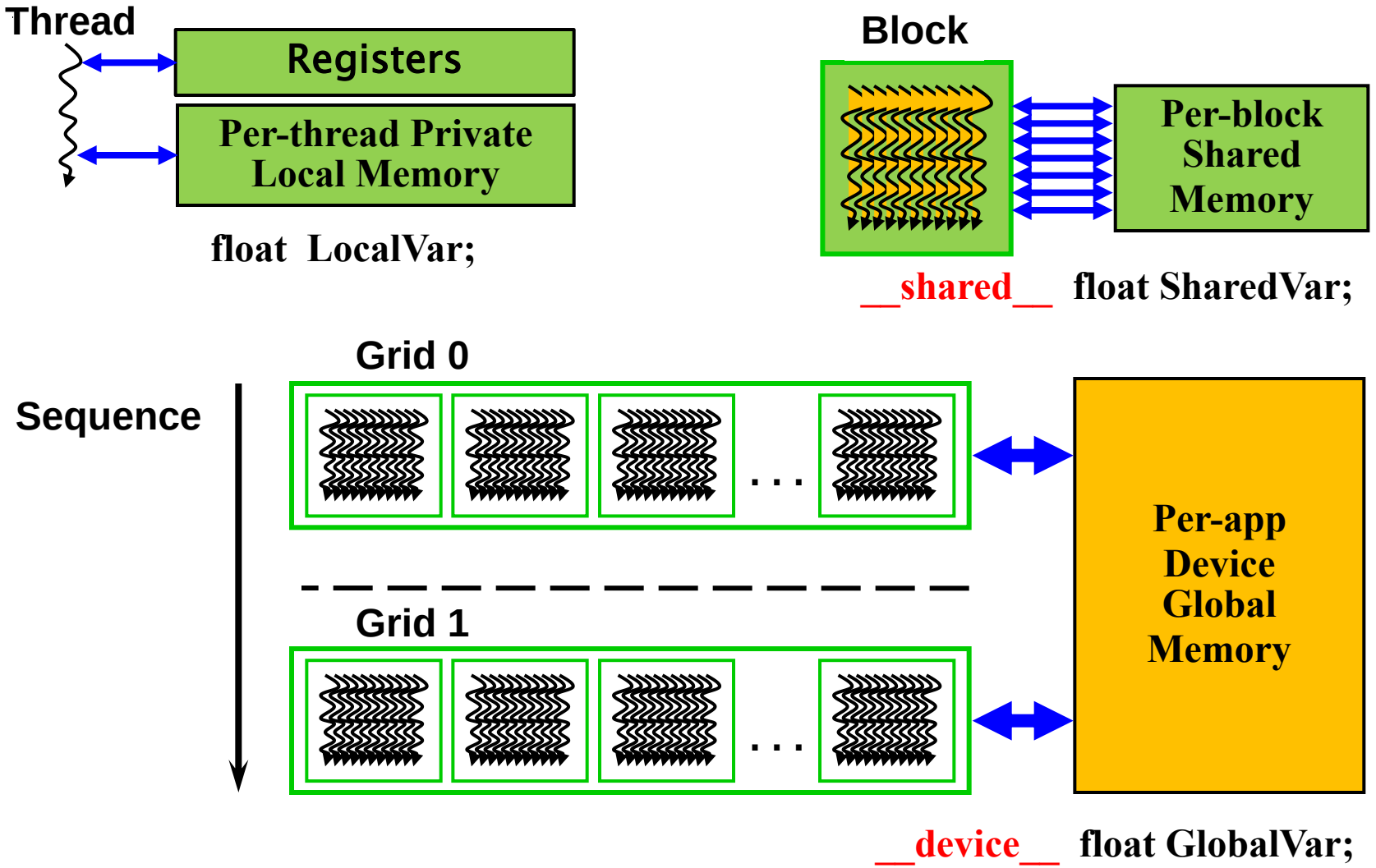
- GPU threads are extremely lightweight
  - Very little creation overhead
- GPU needs 1000s of threads for full efficiency
  - Multi-core CPU needs only a few

Computational Grid

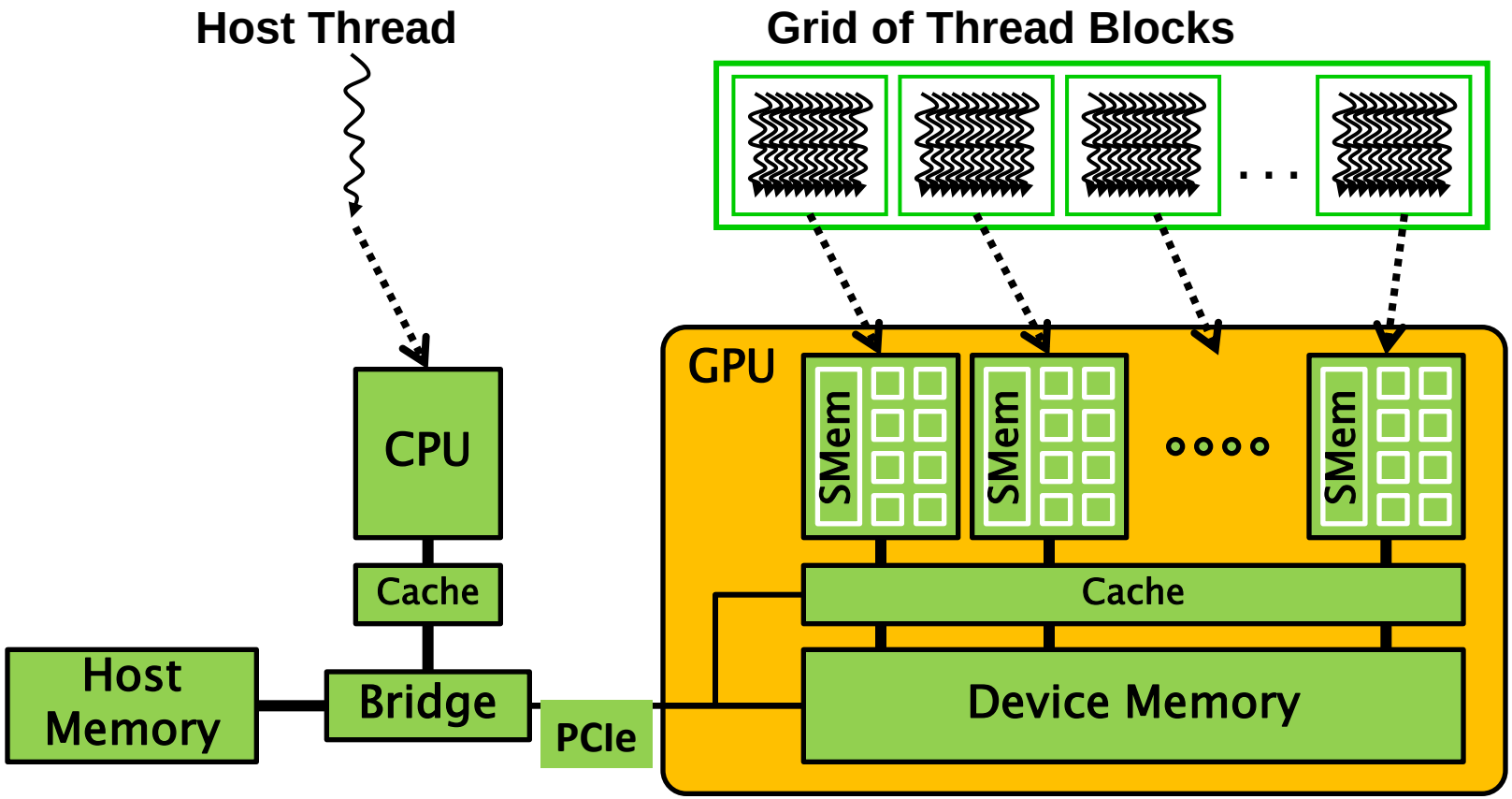
- The computational grid consist of a grid of thread blocks
- All blocks execute the same program (kernel). Only one kernel at a time.
- A thread block is a group of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
  - Two threads from two different blocks cannot cooperate



CUDA Parallel Threads and Memory



CUDA kernel maps to Grid of Blocks



Kernel Function call

- kernel<<<grid, block, stream, shared_mem>>>();
  - Grid: Grid dimension (up to 2D)
  - Block: Block dimension (up to 3D)
  - Stream: stream ID (optional)
  - Shared_mem: shared memory size (optional)

```
__global__ void filter(int *in, int *out);

dim3 grid(16, 16);
dim3 block (16, 16) ;
filter <<< grid, block, 0, 0 >>> (in, out);
\\ filter <<< grid, block >>> (in, out);
```

CUDA Example: Add_matrix

```
// Set grid size
const int N = 1024;
const int blocksize = 16;

// Compute kernel
__global__
void add_matrix( float* a, float *b, float *c, int N )
{
 // threadIdx.x is a built-in variable provided by CUDA at runtime
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if ( i < N && j < N )
      c[index] = a[index] + b[index];
}
```

CUDA Example: Add_matrix

```
int main() {
    \\ CPU memory allocation
    float *a = new float[N*N];
    float *b = new float[N*N];
    float *c = new float[N*N];

    for ( int i = 0; i < N*N; ++i ) {
      a[i] = 1.0f; b[i] = 3.5f; }

    \\GPU memory allocation
    float *ad, *bd, *cd;
    const int size = N*N*sizeof(float);
    cudaMalloc( (void**)&ad, size );
    cudaMalloc( (void**)&bd, size );
    cudaMalloc( (void**)&cd, size );
```

CUDA Example: `Add_matrix`

```
    \\ copy data to GPU
    cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

    \\ execute kernel
    dim3 dimBlock( blocksize, blocksize );
    dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
    add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

    \\ copy result back to CPU
    cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

    \\ clean up and return
    cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
    delete[] a; delete[] b; delete[] c;
    return EXIT_SUCCESS;
}
```

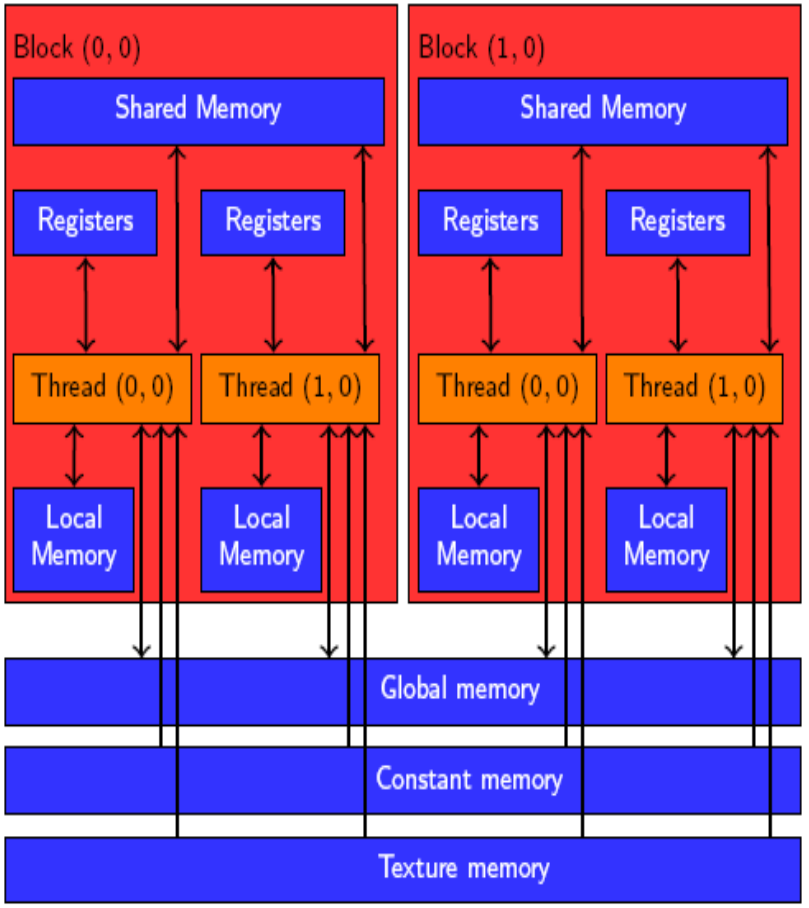
NVCC Compiler

- CUDA kernels are typically stored in files ending with .cu
- NVCC uses the host compiler (CL/G++) to compile CPU code
- NVCC automatically handles #include's and linking

```
>> nvcc  -o executable  program.cu
>> ./executable
```

Memory Model

- Registers
  - Per thread, Read-Write
- Local memory
  - Per thread, Read-Write
- Shared memory
  - Per block Read-Write For sharing data within a block
- Global memory
  - Per grid Read-Write Not cached
- Constant memory
  - Per grid Read-only Cached
- Texture memory
  - Per grid Read-only Spatially cached



CUDA Type Qualifiers

Function type qualifiers

`__device__`

- Executed on the device
- Callable from the device only

`__global__`

- Executed on the device
- Callable from the host only

`__host__`

- Executed on the host
- Callable from the host only
- Default type if unspecified

Variable type qualifiers

`__device__`

- global memory space
- Is accessible from all the threads within the grid

`__constant__`

- constant memory space
- Is accessible from all the threads within the grid

`__shared__`

- space of a thread block
- Is only accessible from all the threads within the block

CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`int var;`	register	thread	thread
`int array_var[10];`	local	thread	thread
`__shared__ int shared_var;`	shared	block	block
`__device__ int global_var;`	global	grid	application
`__constant__ int constant_var;`	constant	grid	application

- "automatic" scalar variables without qualifier reside in a register
  - compiler will spill to thread local memory
- "automatic" array variables without qualifier reside in thread-local memory

CUDA Variable Type Performance

Variable declaration	Memory	Penalty
`int var;`	register	1x
`int array_var[10];`	local	100x
`__shared__ int shared_var;`	shared	1x
`__device__ int global_var;`	global	100x
`__constant__ int constant_var;`	constant	1x

- scalar variables reside in fast, on-chip registers
- shared variables reside in fast, on-chip memories
- thread-local arrays & global variables reside in uncached off-chip memory
- constant variables reside in cached off-chip memory

Example – shared variables

```
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
  // compute this thread's global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i > 0)
  {
    // each thread loads two elements from global memory
    int x_i = input[i];
    int x_i_minus_one = input[i-1];
    result[i] = x_i - x_i_minus_one;
  }
}
```

Example – shared variables

```
// optimized version of adjacent difference
__global__ void adj_diff(int *result, int *input)
{
  // shorthand for threadIdx.x
  int tx = threadIdx.x;
  // allocate a __shared__ array, one element per thread
  __shared__ int s_data[BLOCK_SIZE];
  // each thread reads one element to s_data
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;
  s_data[tx] = input[i];

   if(tx > 0)
     result[i] = s_data[tx] - s_data[tx-1];
   else if(i > 0)
  {
    // handle thread block boundary
    result[i] = s_data[tx] - input[i-1];
  }
__syncthreads();
}
```

CUDA Timer

```
int main()
{     float myTime;
      cudaEvent_T myTimerStart, myTimerStop;
      cudaEventCreate(&myTimerStart);
      cudaEventCreate(&myTimerStop);

      cudaEventRecord(myTimerStart, 0);
      // task to be timed
      cudaEventRecord(myTimerStop, 0);

      cudaEventSynchronize(myTimerStop);
      cudaEventElapsedTime(&myTime, myTimerStart,
                                          myTimerStop);   }
```

TBB

TBB: Threading Building Blocks

- http://threadingbuildingblocks.org
- Task-based parallelism
- C++ Template library
  - abstracts platform details and threading mechanisms for scalability and performance
  - TBB's task manager
  - Scalable memory allocator
  - Concurrent container
  - Generic Parallel Algorithms

TBB

- Tasks are light-weight entities at user-level
- TBB parallel algorithms map tasks onto threads automatically
  - Task scheduler manages the thread pool
    - Scheduler is *unfair* to favor tasks that have been most recent in the cache
  - Task scheduler is designed to address common performance issues of parallel programming with native threads
    - Fair scheduling → Non-preemptive unfair scheduling
    - High overhead → Programmer specifies tasks, not threads
    - Load imbalance → Work-stealing balances load

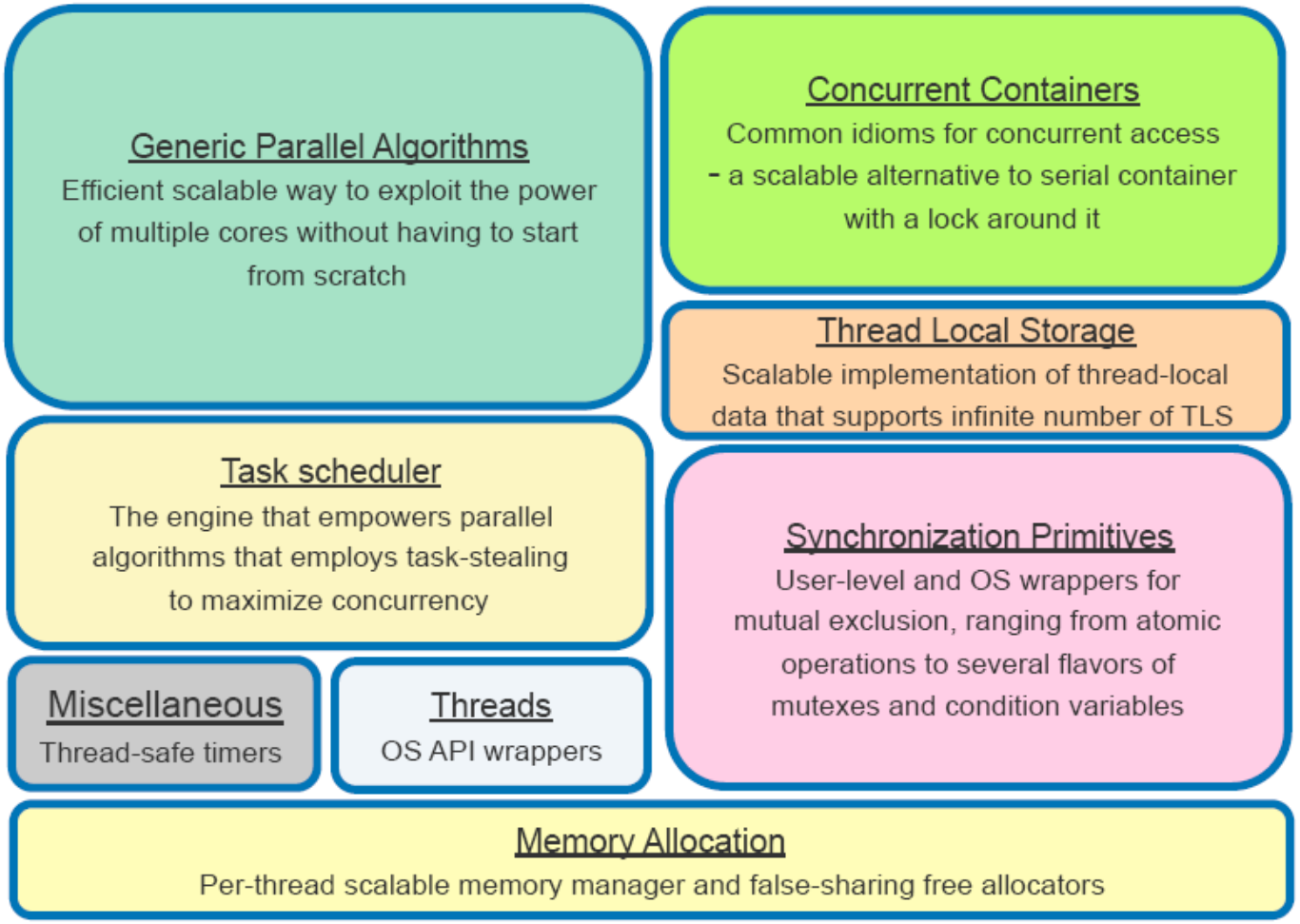
TBB

- TBB is not intended for
  - I/O bound processing
  - Real-time processing
- Direct use only from C++
- Distributed memory not supported

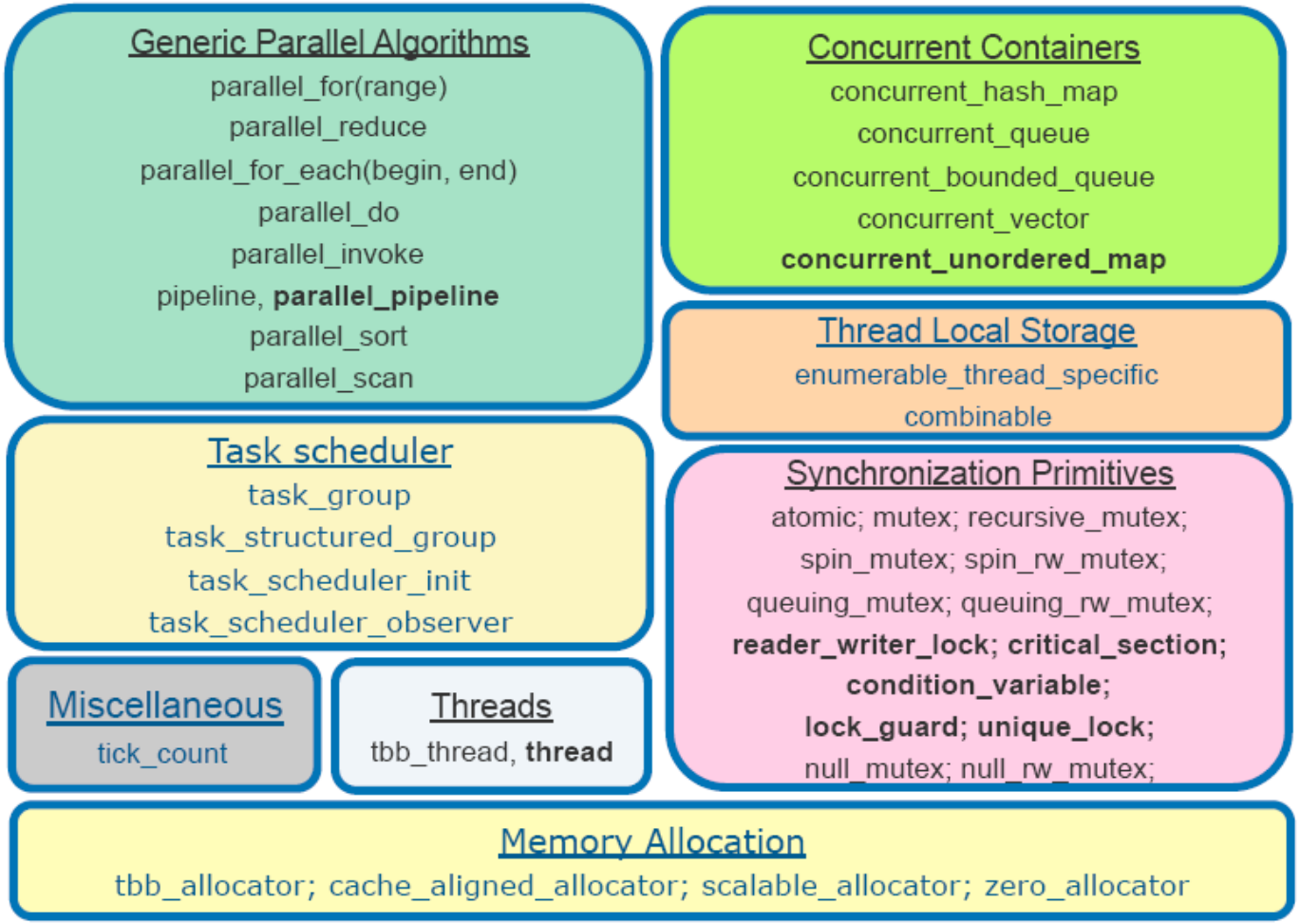
TBB versus OpenMP

- TBB provides acceptable scalability for multi-core platforms
  - TBB has been designed to be more conducive to application parallelization on client platforms such as laptops and desktops
  - TBB goes beyond data parallelism to be suitable for programs with nested parallelism, irregular parallelism and task parallelism.
- OpenMP and MPI continue to be good choices in High Performance Computing applications

TBB Modules


Generic Parallel Algorithms
Efficient scalable way to exploit the power of multiple cores without having to start from scratch
Concurrent Containers
Common idioms for concurrent access - a scalable alternative to serial container with a lock around it
Thread Local Storage
Scalable implementation of thread-local data that supports infinite number of TLS
Task scheduler
The engine that empowers parallel algorithms that employs task-stealing to maximize concurrency
Synchronization Primitives
User-level and OS wrappers for mutual exclusion, ranging from atomic operations to several flavors of mutexes and condition variables
Miscellaneous
Thread-safe timers
Threads
OS API wrappers
Memory Allocation
Per-thread scalable memory manager and false-sharing free allocators

TBB Modules


Generic Parallel Algorithms
parallel_for(range)
parallel_reduce
parallel_for_each(begin, end)
parallel_do
parallel_invoke
pipeline, parallel_pipeline
parallel_sort
parallel_scan
Concurrent Containers
concurrent_hash_map
concurrent_queue
concurrent_bounded_queue
concurrent_vector
concurrent_unordered_map
Thread Local Storage
enumerable_thread_specific
combinable
Task scheduler
task_group
task_structured_group
task_scheduler_init
task_scheduler_observer
Synchronization Primitives
atomic; mutex; recursive_mutex;
spin_mutex; spin_rw_mutex;
queuing_mutex; queuing_rw_mutex;
reader_writer_lock; critical_section;
condition_variable;
lock_guard; unique_lock;
null_mutex; null_rw_mutex;
Miscellaneous
tick_count
Threads
tbb_thread, thread
Memory Allocation
tbb_allocator; cache_aligned_allocator; scalable_allocator; zero_allocator

TBB example

```
#include “tbb/task_scheduler_int.h”
#include “tbb/blocked_range.h”
#include “tbb/parallel_for.h”

using namespace tbb;

int main(int argc, char *argv[])
{
   task_scheduler_init init;
   parallel_change_array();

}
```

TBB example: Parallel_for

```
void parallel_change_array()

{
   parallel for ( blocked_range <int> (0, list_count),
   [=](const blocked_range <int> & r)
      {
          for (int i=r.begin(); i< r.end(); i++)
         {
               a[i] *=2;
         }
      }
    );
}
```

TBB Example: Time

```
#include <iostream>
#include "tbb/task_scheduler_init.h"
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"
#include "tbb/tick_count.h"
using namespace tbb;
using namespace std;

class vector_mult{
    double *const v1, *const v2;   double *v3;

public:
    vector_mult(double *vec1, double *vec2, double *vec3)
        : v1(vec1), v2(vec2), v3(vec3) { }
    void operator() (const blocked_range<size_t> &r) const {
        for(size_t i=r.begin(); i!=r.end(); i++)
            v3[i] = v1[i] * v2[i];
    }
};
```

TBB Example: Time

```
const size_t vec_size = 1000000;

int main(int argc, char* argv[])
{
    // allocate storage for vectors
    double *v1, *v2, *v3;
    v1 = (double*)malloc(vec_size*sizeof(double));
    v2 = (double*)malloc(vec_size*sizeof(double));
    v3 = (double*)malloc(vec_size*sizeof(double));


  for(size_t i=0; i<vec_size; i++) { v1[i] = v2[i] = v3[i] = i; }
    task_scheduler_init init;
    tick_count parallel_start = tick_count::now();
    parallel_for( blocked_range<size_t>(0,vec_size,1000),
           vector_mult(v1,v2,v3) );
    tick_count parallel_end = tick_count::now();

    return 0;
}
```

TBB Example: Dot Product

```
#include <iostream>
#include "tbb/task_scheduler_init.h"
#include "tbb/parallel_reduce.h"
#include "tbb/blocked_range.h"
using namespace tbb;
using namespace std;

class vector_dot_prod {
    double *const v1, *const v2;
public:
    double result;
    // constructor copies the arguments into local storage
    vector_dot_prod(double *vec1, double *vec2)
        : v1(vec1), v2(vec2), result(0) { }
    // splitting constructor
    vector_dot_prod(vector_dot_prod &vdp, split)
        : v1(vdp.v1), v2(vdp.v2), result(0) { }
    // overload () so it does a dot product
    void operator() (const blocked_range<size_t> &r) {
        for(size_t i=r.begin(); i!=r.end(); i++)
            result += v1[i] * v2[i];
    }
    // join method adds partial results
    void join(vector_dot_prod &v) { result += v.result; }
};
```

TBB Example: Dot Product

```
int main(int argc, char* argv[])
{
    // allocate storage for vectors
    double *v1, *v2, result;
    v1 = (double*)malloc(vec_size*sizeof(double));
    v2 = (double*)malloc(vec_size*sizeof(double));

     // dot-prod the vectors in parallel
    task_scheduler_init init;
    vector_dot_prod v(v1,v2);
    parallel_reduce( blocked_range<size_t>(0,vec_size,1000), v );

    // print the result to make sure it is correct
    cout << "Parallel dot product of the two vectors is " << v.result <<
endl;

    return 0;
}
```

MPI

Topic Overview

- Message Passing Standard
- Point-to-point Communication
- Collective Communication

The MPI Process

- Development began in early 1992
- Open process/Broad participation
  - IBM,Intel, TMC, Meiko, Cray, Convex, Ncube
  - PVM, p4, Express, Linda, …
  - Laboratories, Universities, Government
- Final version of draft in May 1994
- Public and vendor implementations are now widely available
- MPI 2.0 release in 1997
  - Parallel I/O
  - Remote memory operation (One sided)
- MPI 3.0 current effort

Compiling and Running MPI

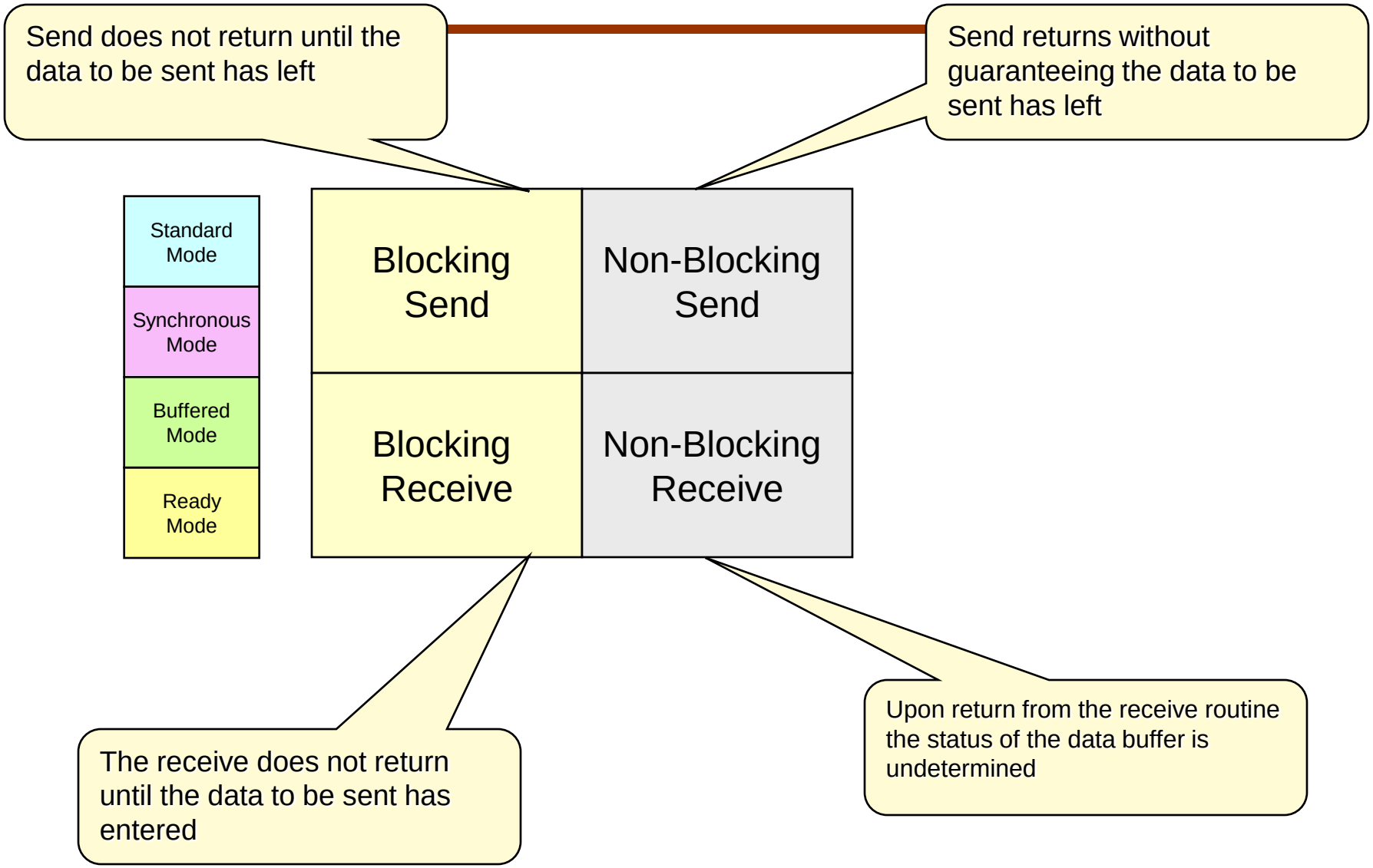
- mpicc –o executable source.c –libraryname
- mpirun –np 16 executable

- Open source: MPICH, OpenMPI

Skeleton MPI Program

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Point-to-Point Communication



Message Passing

```
#include <stdio.h>
#include "mpi.h

int main(int argc,char *argv[])
{
  int myid, numprocs, tag,source,destination,count, buffer;
  MPI_Status status;
  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  tag=1234;  source=0;  destination=1;  count=1;

  if(myid == source){
    buffer=5678;
    MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
    printf("processor %d  sent %d\n",myid,buffer);
  }

  if(myid == destination){
    MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
    printf("processor %d  got %d\n",myid,buffer);
  }
  MPI_Finalize();
}
```

Message Passing

```
MPI_Send( & buf, count, datatype, dest,
          tag, comm )
```

- `buf` is the address of the data to be sent
- `count` is the number of elements of the MPI datatype which `buf` contains
- `datatype` is the MPI datatype
- `dest` is the destination process for the message. This is specified by the rank of the destination within the group associated with the communicator `comm`
- `tag` is a marker used by the sender to distinguish between different types of messages
- `comm` is the communicator shared by the sender and the receiver

Datatypes in MPI

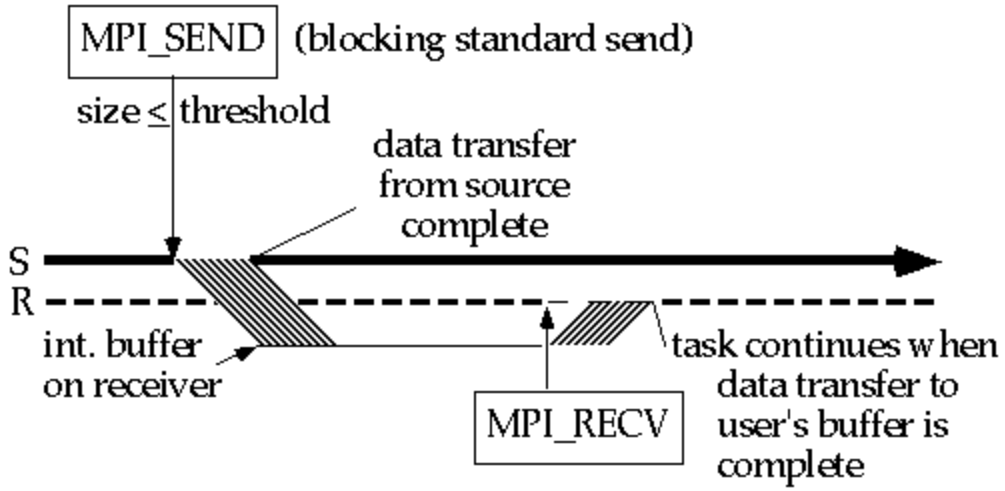
MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Blocking Communication

- Blocking send will send message and return
  - Does not mean that message has been received, just that process free to move on without adversely affecting message.
- Four blocking point-to-point modes depending on when a send operation is initiated or completed
  - Standard
    - A send may be initiated even if a matching receive has not been initiated
  - Ready
    - A send can be initiated only if a matching receive has been initiated
  - Synchronous
    - A send will not complete until message deliver is guaranteed.
  - Buffered
    - Completion is always independent on matching receive. The message is buffered to ensure this.

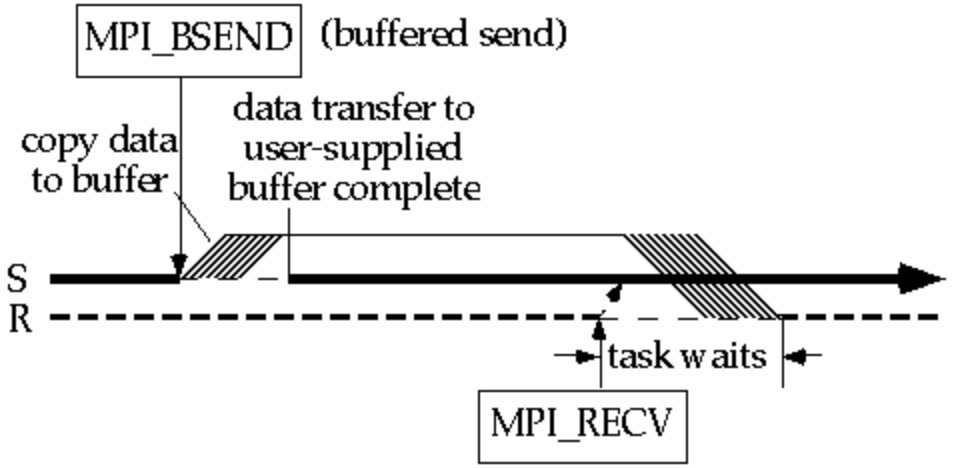
Standard Send (MPI_Send)

- It is up to MPI to decide whether outgoing messages will be buffered.
- Completes once the message has been sent, which may or may not imply that the message has arrived at its destination
- Can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted.
- Has non-local completion semantics, since successful completion of the send operation may depend on the occurrence of a matching receive.



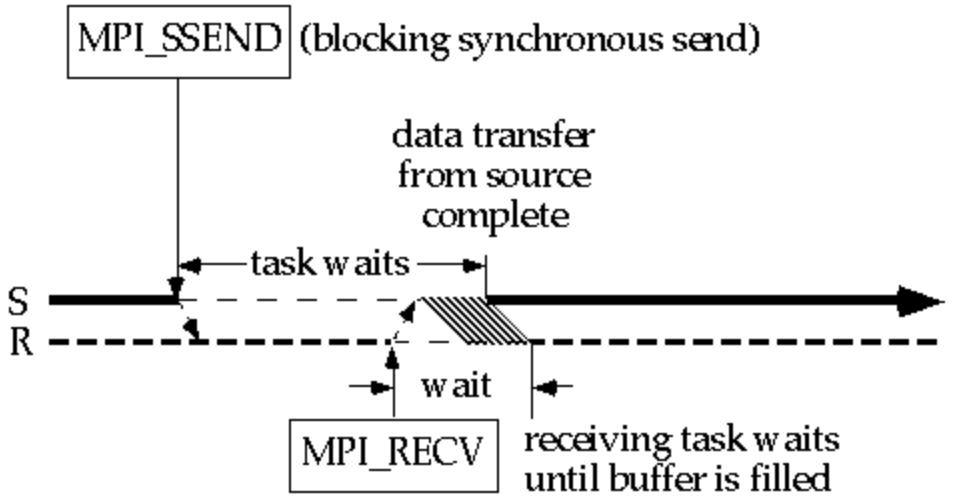
Buffered Send (MPI_Bsend)

- Can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted.
- Has local completion semantics: its completion does not depend on the occurrence of a matching receive.
- In order to complete the operation, it may be necessary to buffer the outgoing message locally. For that purpose, buffer space is provided by the application.



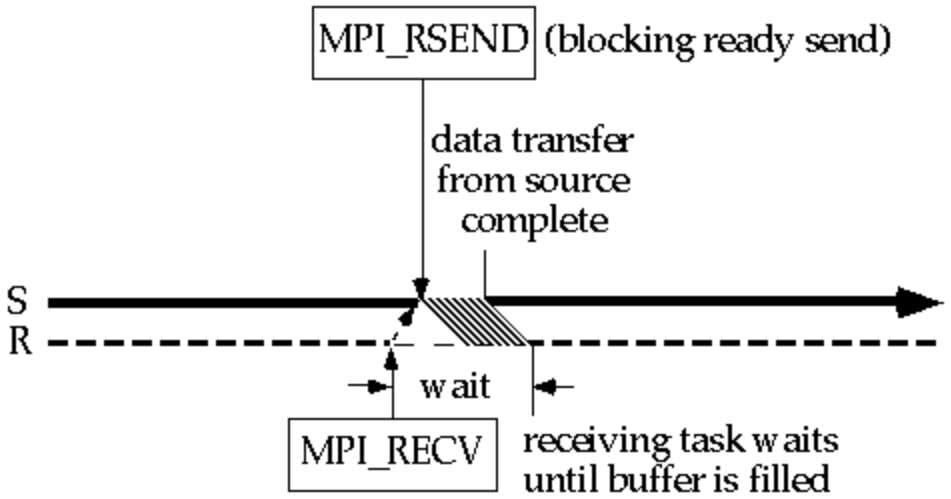
Synchronous Send (MPI_Ssend)

- Can be started whether or not a matching receive was posted
- Will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send.
- Provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication.



Ready Send (MPI_Rsend)

- Completes immediately
- Can be started only if the matching receive has already been posted.
- Saves on overhead by avoiding handshaking and buffering



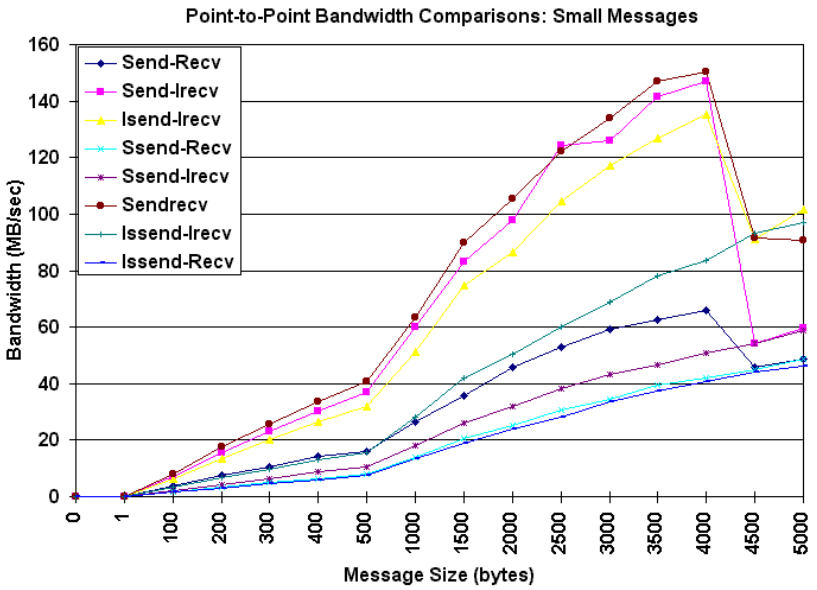
Non Blocking Communication

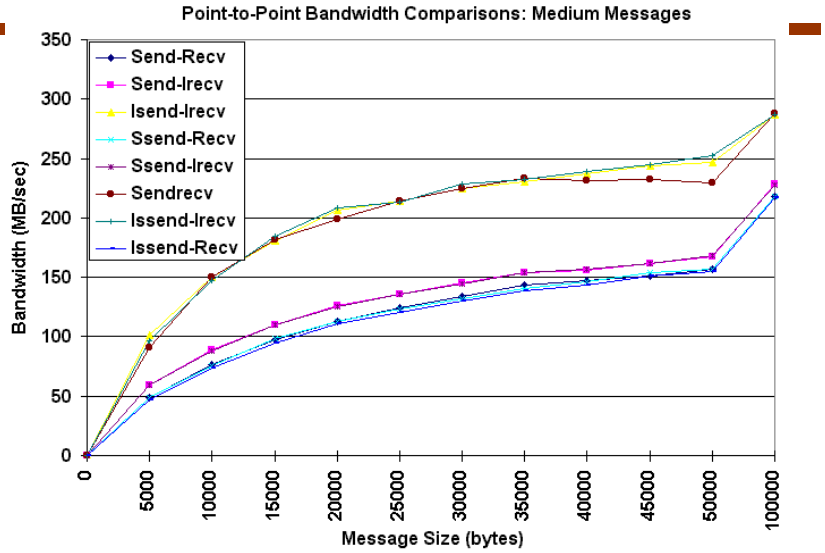
- To overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations ("I" stands for "Immediate")
  - MPI_Isend() will return "immediately" even before source location is safe to be altered
    - `int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
  - MPI_Irecv() will return even if no message to accept
    - `int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)`

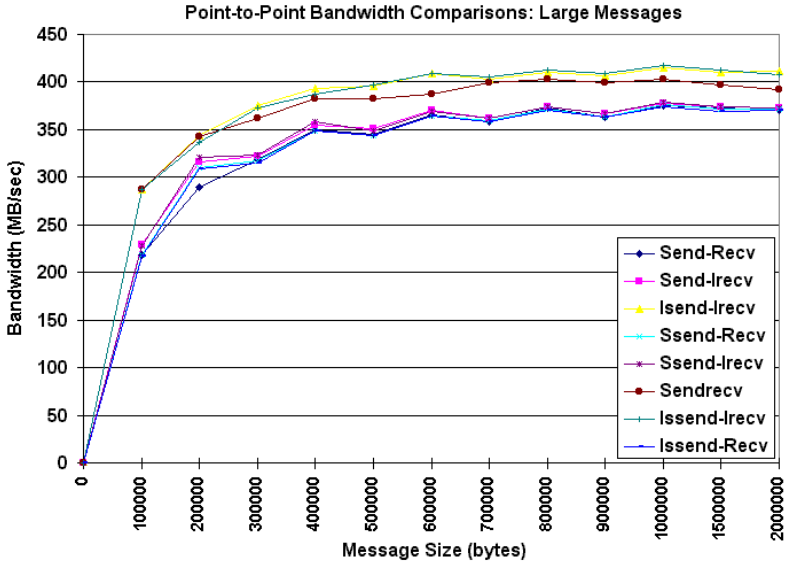
Non Blocking Communication

- These operations return before the operations have been completed.
- Function MPI_Test tests whether or not the non- blocking send or receive operation identified by its request has finished.
  - `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
- MPI_Wait waits for the operation to complete.
  - `int MPI_Wait(MPI_Request *request, MPI_Status *status)`

Performance of Point-to-Point







Non-blocking operations perform best

Performance gains depend upon message size

The greatest gains occur with smaller message sizes

Collective Communication

- Point-to-point communications involve pairs of processes.
- Many message passing systems provide operations which allow larger numbers of processes to participate
  - Broadcast
  - Scatter
  - Gather
  - Reduce

Broadcast

```
MPI_Bcast(&buffer, count, datatype, root,
  communicator);
```

- One node (root) sends a message all others receive the message

Broadcast

```
#include <stdio.h>
#include "mpi.h"
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
    int i,myid, numprocs;
    int source,count;
    int buffer[4];
    MPI_Status status;
    MPI_Request request;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    source=0;
    count=4;
    if(myid == source){
      for(i=0;i<count;i++)
        buffer[i]=i;
    }
    MPI_Bcast(buffer,count,MPI_INT,source,MPI_COMM_WORLD);
    for(i=0;i<count;i++)
      printf("%d ",buffer[i]);
    printf("\n");
    MPI_Finalize();
}
```

Scatter

- Similar to Broadcast but sends a section of an array to each processors

- `int MPI_Scatter(&sendbuf, sendcnts, sendtype, &recvbuf, recvcnts, recvtype, root, comm );`

  - Sendbuf is an array of size (number processors*sendcnts)
  - Sendcnts number of elements sent to each processor
  - Recvcnts number of elements obtained from the root processor
  - Recvbuf elements obtained from the root processor, may be an array

Gather

- Used to collect data from all processors to the root, inverse of scatter
- Data is collected into an array on root processor

- `int MPI_Gather(&sendbuf,sendcnts, sendtype, &recvbuf, recvcnts,recvtype,root, comm )`

  - Sendcnts # of elements sent from each processor
  - Sendbuf is an array of size sendcnts
  - Recvcnts # of elements obtained from each processor
  - Recvbuf of size Recvcnts*number of processors

Scatter/Gather

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int numnodes,myid,mpi_err;
#define mpi_root 0

int main(int argc,char *argv[])
{
    int *myray,*send_ray,*back_ray;
    int count;
    int size,mysize,i,k,j,total;
     MPI_Init(argc,argv);
    MPI_Comm_size( MPI_COMM_WORLD, &numnodes );
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

/* each processor will get count elements from the root */
     count=4;
     myray=(int*)malloc(count*sizeof(int));

/* create the data to be sent on the root */
     if(myid == mpi_root){
           size=count*numnodes;
            send_ray=(int*)malloc(size*sizeof(int));
            back_ray=(int*)malloc(numnodes*sizeof(int));
            for(i=0;i<size;i++)
                        send_ray[i]=i;
            }
```

Scatter/Gather (CONT….)

```
/* send different data to each processor */
    MPI_Scatter(send_ray, count, MPI_INT, myray, count, MPI_INT,
                            mpi_root, MPI_COMM_WORLD);


/* each processor does a local sum */
    total=0;
    for(i=0;i<count;i++)
        total=total+myray[i];
    printf("myid= %d total= %d\n ",myid,total);

/* send the local sums back to the root */
    MPI_Gather(&total,1, MPI_INT,back_ray, 1, MPI_INT,
                      mpi_root, MPI_COMM_WORLD);

/* the root prints the global sum */
    if(myid == mpi_root){
      total=0;
      for(i=0;i<numnodes;i++)
        total=total+back_ray[i];
      printf("results from all processors= %d \n ",total);
    }
    mpi_err = MPI_Finalize();
}
```

Reduce

- Used to combine partial results from all processors. Result returned to root processor
- `int MPI_Reduce(&sendbuf, &recvbuf, count, datatype, operation,root, communicator)`
  - Operation is a type of mathematical operation. Several types of operations available

MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

Scatter/Reduce

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int numnodes,myid,mpi_err;
#define mpi_root 0

void init_it(int  *argc, char ***argv);

void init_it(int  *argc, char ***argv) {
     mpi_err = MPI_Init(argc,argv);
    mpi_err = MPI_Comm_size( MPI_COMM_WORLD, &numnodes );
    mpi_err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
}

int main(int argc,char *argv[]){
     int *myray,*send_ray,*back_ray;
     int count;
     int size,mysize,i,k,j,total,gtotal;

     init_it(&argc,&argv);

/* each processor will get count elements from the root */
     count=4;
     myray=(int*)malloc(count*sizeof(int));
/* create the data to be sent on the root */
     if(myid == mpi_root){
         size=count*numnodes;
             send_ray=(int*)malloc(size*sizeof(int));
             back_ray=(int*)malloc(numnodes*sizeof(int));
             for(i=0;i<size;i++)
                         send_ray[i]=i;
             }
```

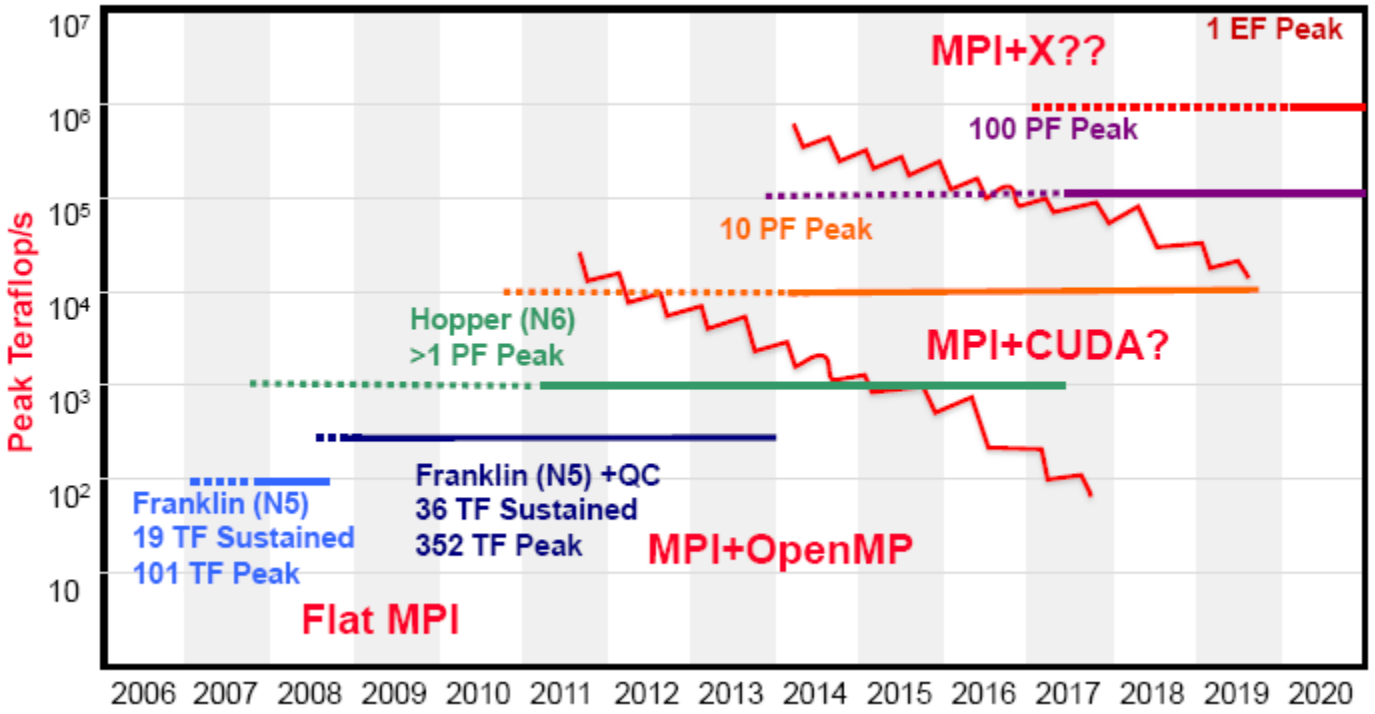
Scatter/Reduce (cont …)

```
/* send different data to each processor */
   MPI_Scatter(send_ray, count, MPI_INT, myray, count,
               MPI_INT, mpi_root, MPI_COMM_WORLD);

/* each processor does a local sum */
   total=0;
   for(i=0;i<count;i++)
       total=total+myray[i];
   printf("myid= %d total= %d\n ",myid,total);

/* send the local sums back to the root */
    MPI_Reduce(&total, &gtotal,1, MPI_INT,
                  MPI_SUM, mpi_root, MPI_COMM_WORLD);

/* the root prints the global sum */
   if(myid == mpi_root){
     printf("results from all processors= %d \n ",gtotal);
   }
    mpi_err = MPI_Finalize();
}
```

MPI + OpenMP: Threads per process

```
#include <omp.h>
#include <mpi.h>
#include <stdio.h>

int main (int argc, char *argv[]){
   int p, myrank, c;

   MPI_Init(&argc, &argv);

   #pragma omp parallel reduction(+:c)
   {
     c=get_number_of_threads();
   }
   printf("%d\n", c);

MPI_Finalize();

return 0;
}
```

The Future of MPI



Source: Kathy Jelick, UCB

MPICH2 Multicore Optimizations

- Using MPI on multicore
  - One MPI process per core – fast communication (214 ns latency)
  - Multithreaded – Hybrid model with OpenMP, UPC, pthreads
    - Nemesis: New "multi-method" communication subsystem for MPICH2
      - uses lock-free queues
      - One receive queue per process
      - Implemented using atomic assembly instructions: no lock overhead
      - Optimized to reduce cache misses

Multithreaded MPI Programming

- Pros
  - Hybrid programming model
  - Use shared-memory algorithms where appropriate
  - One copy of large array shared by all threads
- Cons
  - In general threaded programming can be difficult to write, debug and verify (e.g., using pthreads)
- OpenMP and UPC make threaded programming easier
  - Language constructs to parallelize loops, etc.

MPI Supported Thread Levels

- MPI_THREAD_SINGLE
  - Only one user thread is allowed
- MPI_THREAD_FUNNELED
  - May have one or more threads, but only the “main” thread may make MPI calls
- MPI_THREAD_SERIALIZED
  - May have one or more threads, but only one thread can make MPI calls at a time. It is the application developer’s responsibility to guarantee this.
- MPI_THREAD_MULTIPLE
  - May have one or more threads. Any thread can make MPI calls at any time (with certain conditions).

- MPICH2 supports MPI_THREAD_MULTIPLE

Using Multiple Threads in MPI

- The main thread must call MPI_Init_thread()
  - App requests a thread level
  - MPI returns the thread level actually provided
  - These values need not be the same on every process
  - Hint: Request only the level you need to avoid unnecessary overhead for higher thread levels.
- MPI_Init_thread()
  - Called in place of MPI_Init()
  - Only the main thread should call this
  - The main thread (and only the main thread) must call MPI_Finalize
    - there is no MPI_Finalize_thread()
- MPI does not provide routines to create threads
  - That's left to the user
    - E.g., use pthreads OpenMP, etc

Summary

- Definitions related to concurrency
- Introduction to parallel programming
  - Flynn's Taxonomy (parallel architectures)
  - Memory access models
  - Parallel programming models
- Implementation Tour
  - Threads: OpenMP
  - Data Parallel: CUDA
  - Task Parallel: TBB
  - Message Passing: MPI