Parallel Programming Models

Zhengxiong Hou

Fall, 2022

IV Parallel Programming Models

- 1. Overview
- 2. Shared Memory Model
  - -Threads Model
- 3. Distributed Memory Model
  - -Message Passing Model
- 4. Hybrid Model
- 5. GPGPU(General Purpose Graphics Processing Unit) Programming Model
- 6. Data Parallel Model
- 7. SPMD(Single Program Multi-Data) and MPMD
- 8. Data Intensive Computing Model

1. Overview

- **Parallel programming models exist as an abstraction** above hardware and memory architectures**.**
- A parallel programming model specifies the programmer's view on parallel computer by defining **how the programmer can code an algorithm**.
- This view is influenced by the architectural design and the language, compiler, or the runtime libraries and, thus, there exist many different parallel programming models even for the same architecture.

Parallel Programming Models-overview

- **What do the programmers use for code?**
  - **To determine communication and synchronization**
  - **Communication primitives**
  - **Examples:**

Multiprogramming model

- A set of independence tasks, no communication or synchronization at program level, e.g. web server sending pages to browsers

Shared address space (shared memory) programming

- Tasks operate and communicate via shared data, like bulletin boards

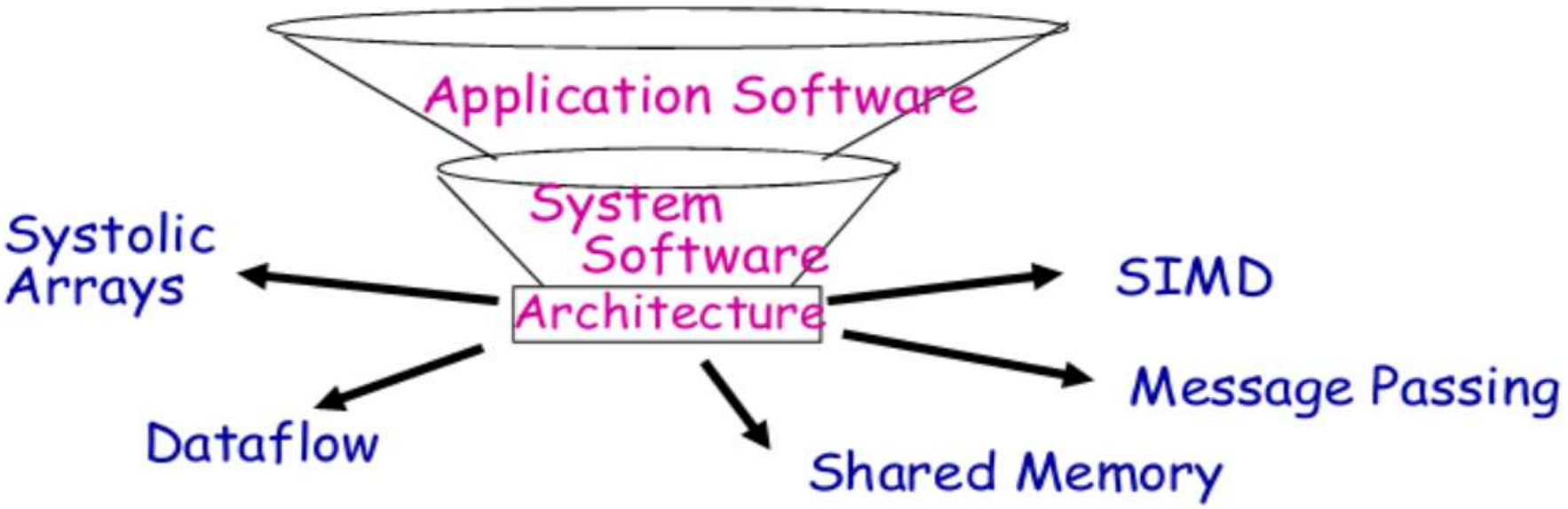
Message passing programming

- Explicit point-to-point communication, like phone calls (connection oriented) or email (connectionless, mailbox posts)

- **Message passing**
  - Independent tasks encapsulating local data
  - Tasks interact by exchanging messages
- **Shared memory**
  - Tasks share a common address space
  - Tasks interact by reading and writing this space asynchronously
- **Data parallelization**
  - Tasks execute a sequence of **independent operations**
  - Data usually evenly partitioned across tasks
  - Also referred to as “embarrassingly parallel”

- **Historically, parallel architectures tied to programming models**

  -Divergent architectures, with no predictable pattern of growth



- Today, Extension of “computer architecture” to support communication and cooperation**?**
  - -OLD: Instruction Set Architecture
  - -NEW: Communication Architecture
- Defines
  - -Critical abstractions, boundaries, and primitives (interfaces)
  - -Organizational structures that implement interfaces (hw or sw)
- Compilers, libraries and OS are important bridges

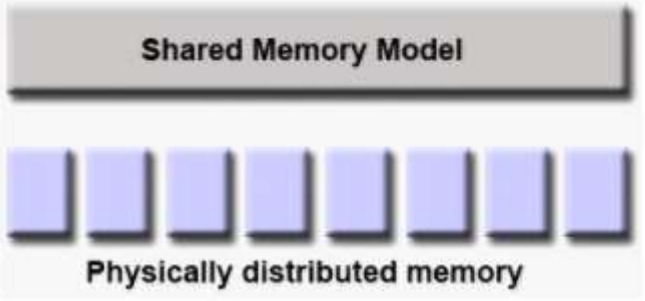
- **Which model to use?**

  This is often a combination of what is available and personal choice. There is no "best" model, although there certainly are better implementations of some models over others.

- Although it might not seem apparent, these models are **NOT** specific to a particular type of machine or memory architecture. In fact, any of these models can (theoretically) be implemented on any underlying hardware. Two examples from the past are discussed below.

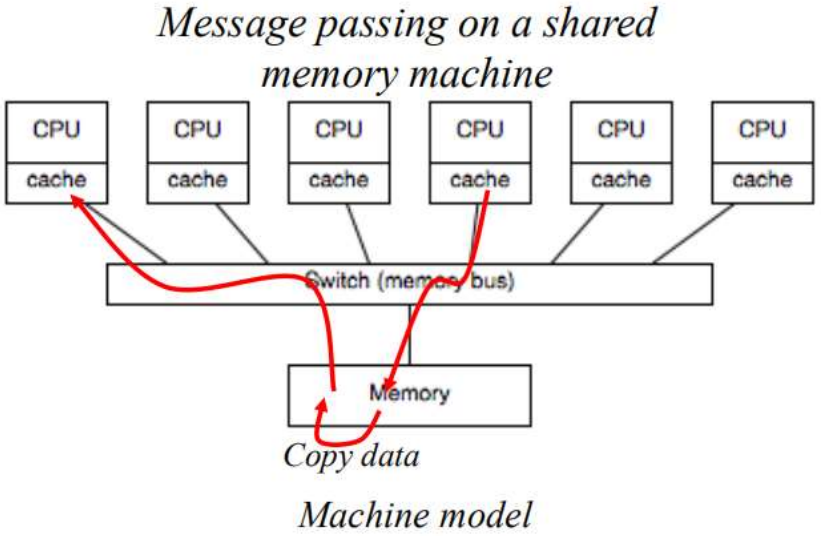
SHARED memory model on a DISTRIBUTED memory machine:

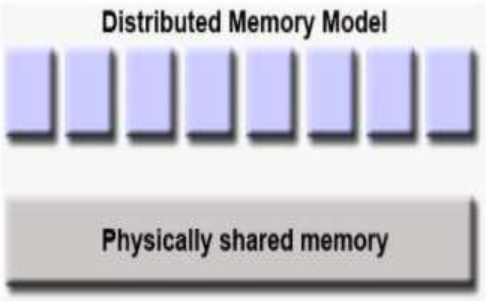
- Kendall Square Research (KSR) ALLCACHE approach. Machine memory was physically distributed across networked machines, but appeared to the user as a single shared memory global address space. Generically, this approach is referred to as "virtual shared memory".



DISTRIBUTED memory model on a SHARED memory machine:

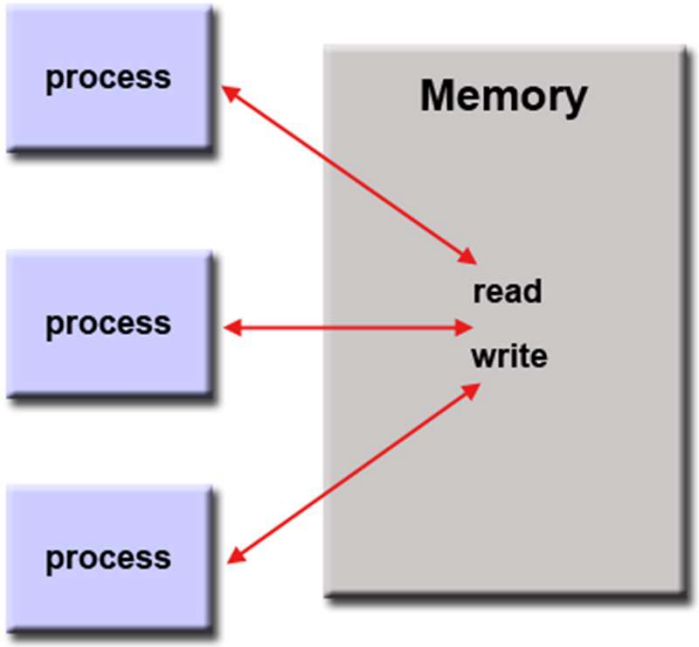
- Message Passing Interface (MPI) on SGI Origin 2000. The SGI Origin 2000 employed the CC-NUMA type of shared memory architecture, where every task has direct access to global address space spread across all machines. However, the ability to send and receive messages using MPI, as is commonly done over a network of distributed memory machines, was implemented and commonly used.





2. Shared Memory Model (without threads)

- In this programming model, *processes/tasks* share a common address space, which they read and write to asynchronously.
- Various mechanisms such as locks / semaphores are used to control access to the shared memory, resolve contentions and to prevent race conditions and deadlocks.
- This is perhaps the simplest parallel programming model.



Advantage & disadvantage

- An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. All processes see and have equal access to shared memory. Program development can often be simplified.
- An important disadvantage in terms of performance is that it becomes more difficult to understand and manage ***data locality***:
  - Keeping data local to the process that works on it conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processes use the same data.
  - Unfortunately, controlling data locality is hard to understand and may be beyond the control of the average user.

Implementations:

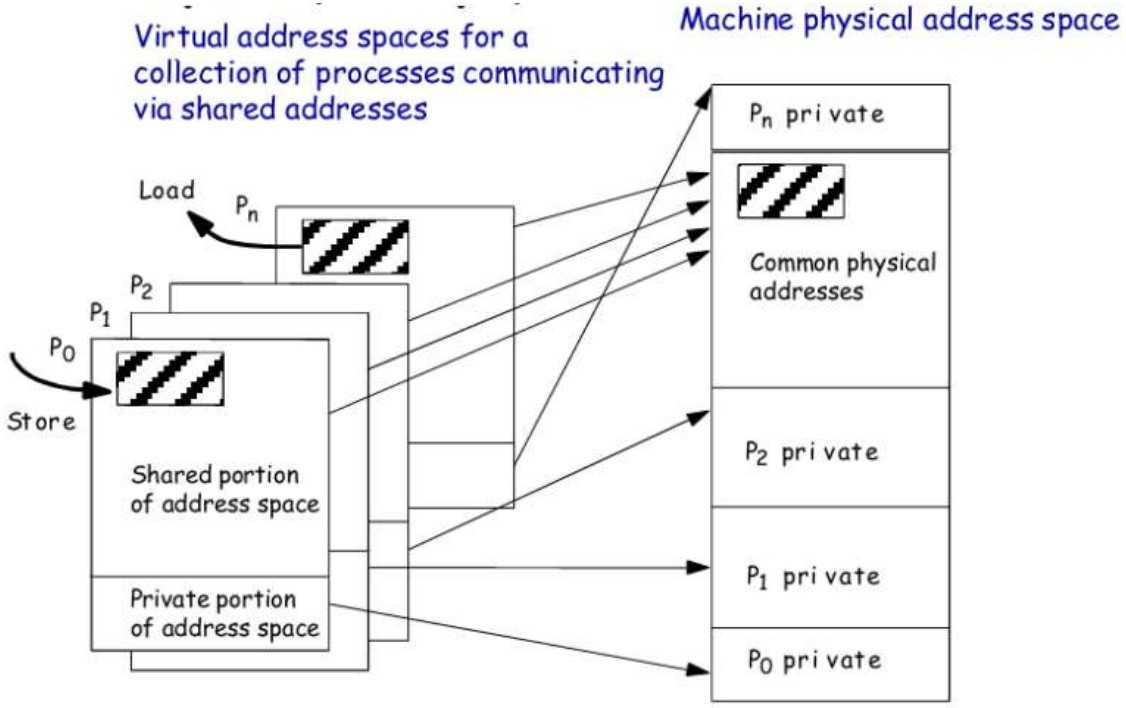
- On stand-alone shared memory machines, native operating systems, compilers and/or hardware provide support for shared memory programming. For example, the POSIX standard provides an API for using shared memory, and UNIX provides shared memory segments (shmget, shmat, shmctl, etc).
- On distributed memory machines, memory is physically distributed across a network of machines, but made global through specialized hardware and software. A variety of SHMEM implementations are available: http://en.wikipedia.org/wiki/SHMEM.

Shared Memory Model

- Any processor can **directly** reference any memory location
  -Communication occurs implicitly as result of loads and stores
- Convenient
  -Location transparency
  -Similar programming model to time-sharing on uniprocessors
  • Except processes run on different processors
  • Good throughput on multi programmed workloads
- Popularly known as *shared memory* machines or model
  -Ambiguous: memory may be physically distributed among processors

Shared Memory Model-processes

- Process: virtual address space plus one or more threads of control
- Portions of address spaces of processes are shared



- Writes to shared address visible to other threads, processes
- Natural extension of uniprocessor model: conventional memory operations for comm.; special atomic operations for synchronization

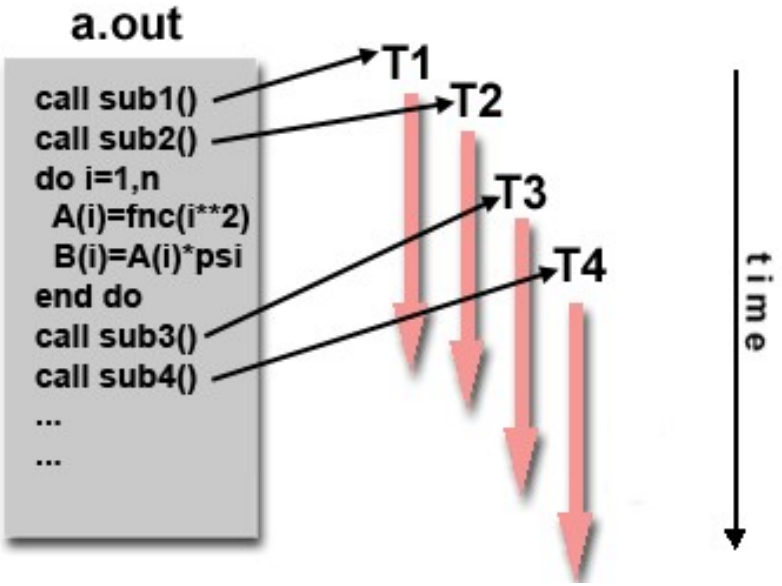
Shared Memory Model-Threads Model

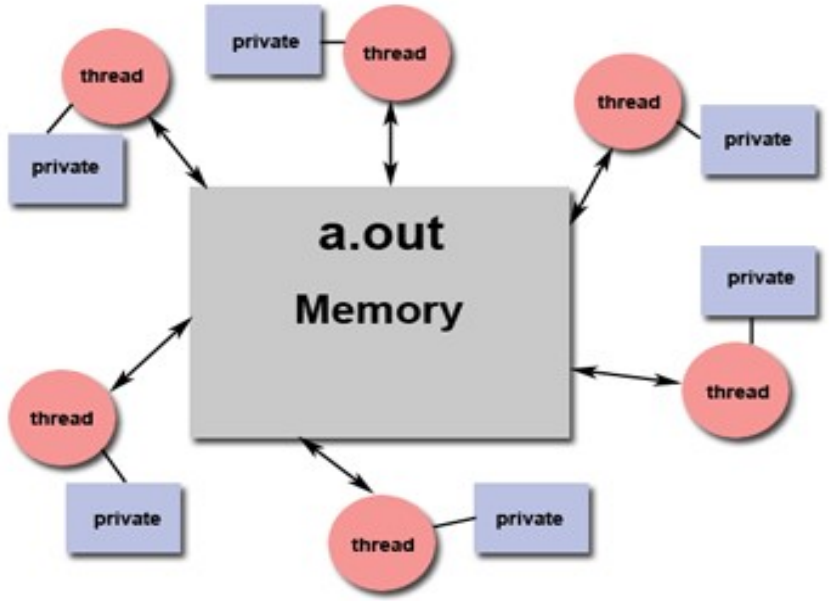
- This programming model is **a main type of shared memory programming**.
- In the threads model of parallel programming, a single "heavy weight" process can have multiple "light weight", concurrent execution paths.
- Perhaps *the most simple analogy* that can be used to describe threads is the concept of a single program that includes a number of subroutines

- **For example:**

  - The main program **a.out** is scheduled to run by the native operating system. **a.out** loads and acquires all of the necessary system and user resources to run. This is the "heavy weight" process.
  - **a.out** performs some serial work, and then creates a number of tasks (threads) that can be scheduled and run by the operating system concurrently.
  - Each thread has local data, but also, shares the entire resources of **a.out**. This saves the overhead associated with replicating a program's resources for each thread ("light weight"). Each thread also benefits from a global memory view because it shares the memory space of **a.out**.

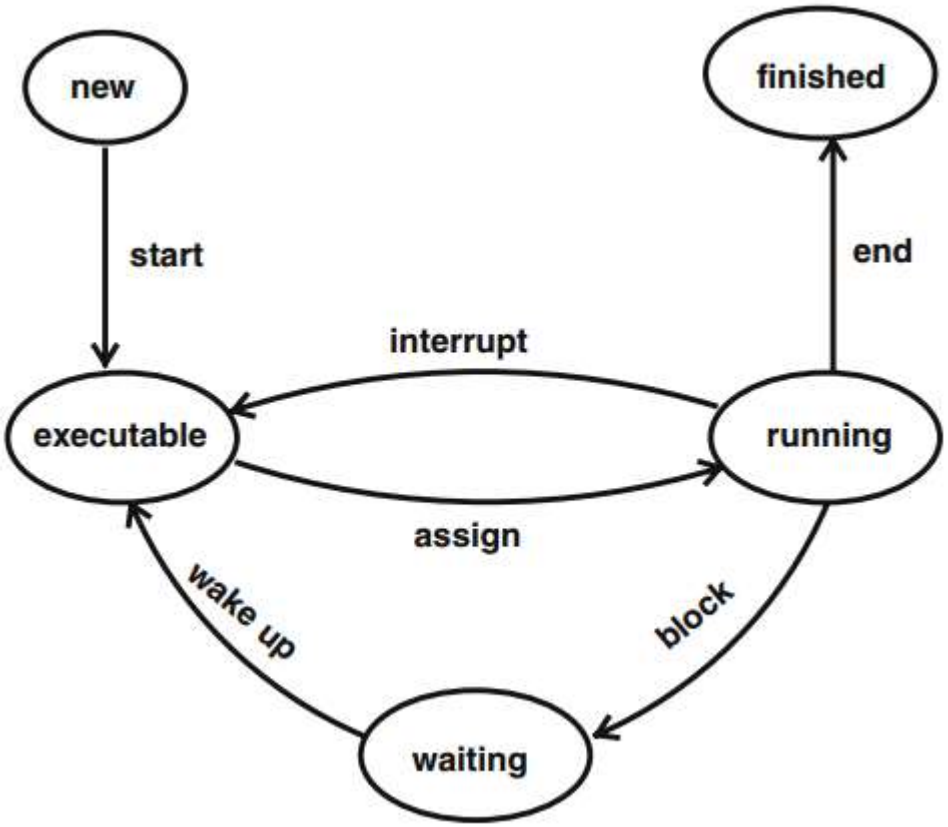
- A thread's work may best be described as a subroutine within the main program. Any thread can execute any subroutine at the same time as other threads.
- Threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to ensure that more than one thread is not updating the same global address at any time.
- Threads can come and go, but **a.out** remains present to provide the necessary shared resources until the application has completed.





States of a thread

- The *nodes* of the diagram show the possible states of a thread
- The *arrows* show possible transitions between them



Implementations:

- From a programming perspective, threads implementations commonly comprise:
  - A library of subroutines that are called from within parallel source code
  - A set of compiler directives imbedded in either serial or parallel source code

- In both cases, the programmer is responsible for determining the parallelism (although compilers can sometimes help).

Implementations-II

- Threaded implementations are not new in computing. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- Unrelated standardization efforts have resulted in two very different implementations of threads:

  POSIX Threads *and* ***OpenMP****.*

POSIX Threads

- Specified by the IEEE POSIX 1003.1c standard (1995). C Language only
- Part of Unix/Linux operating systems
- Library based
- Commonly referred to as **Pthreads**
- Very explicit parallelism; requires significant programmer attention to detail
- POSIX Threads tutorial:

  computing.llnl.gov/tutorials/pthreads

- Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library - though this library may be part of another library, such as libc, in some implementations.

Creating and Terminating Threads

- **Routines:**

```
pthread_create (thread,attr,start_routine,arg)
pthread_exit (status)
pthread_cancel (thread)
pthread_attr_init (attr)
pthread_attr_destroy (attr)
```

Joining and Detaching Threads

- **Routines:**

```
pthread_join (threadid,status)
pthread_detach (threadid)
pthread_attr_setdetachstate (attr,detachstate)
pthread_attr_getdetachstate (attr,detachstate)
```

Stack Management

- **Routines:**

```
pthread_attr_getstacksize (attr, stacksize)
pthread_attr_setstacksize (attr, stacksize)
pthread_attr_getstackaddr (attr, stackaddr)
pthread_attr_setstackaddr (attr, stackaddr)
```

Pthread Creation and Termination Example

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
Hello World! It's me, thread #2!
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

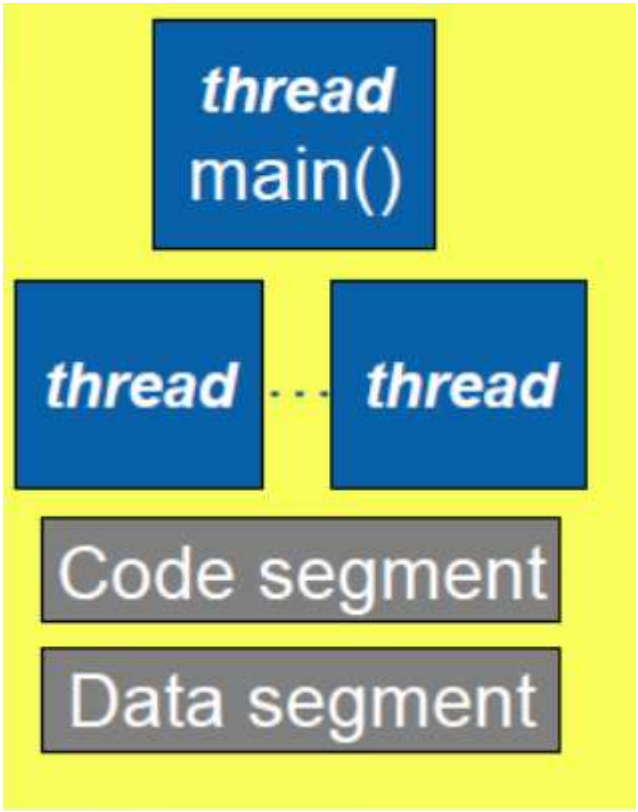
```
void* Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
   }

   return NULL;
}  /* Pth_mat_vect */
```

- **Pthreads matrix-vector multiplication**

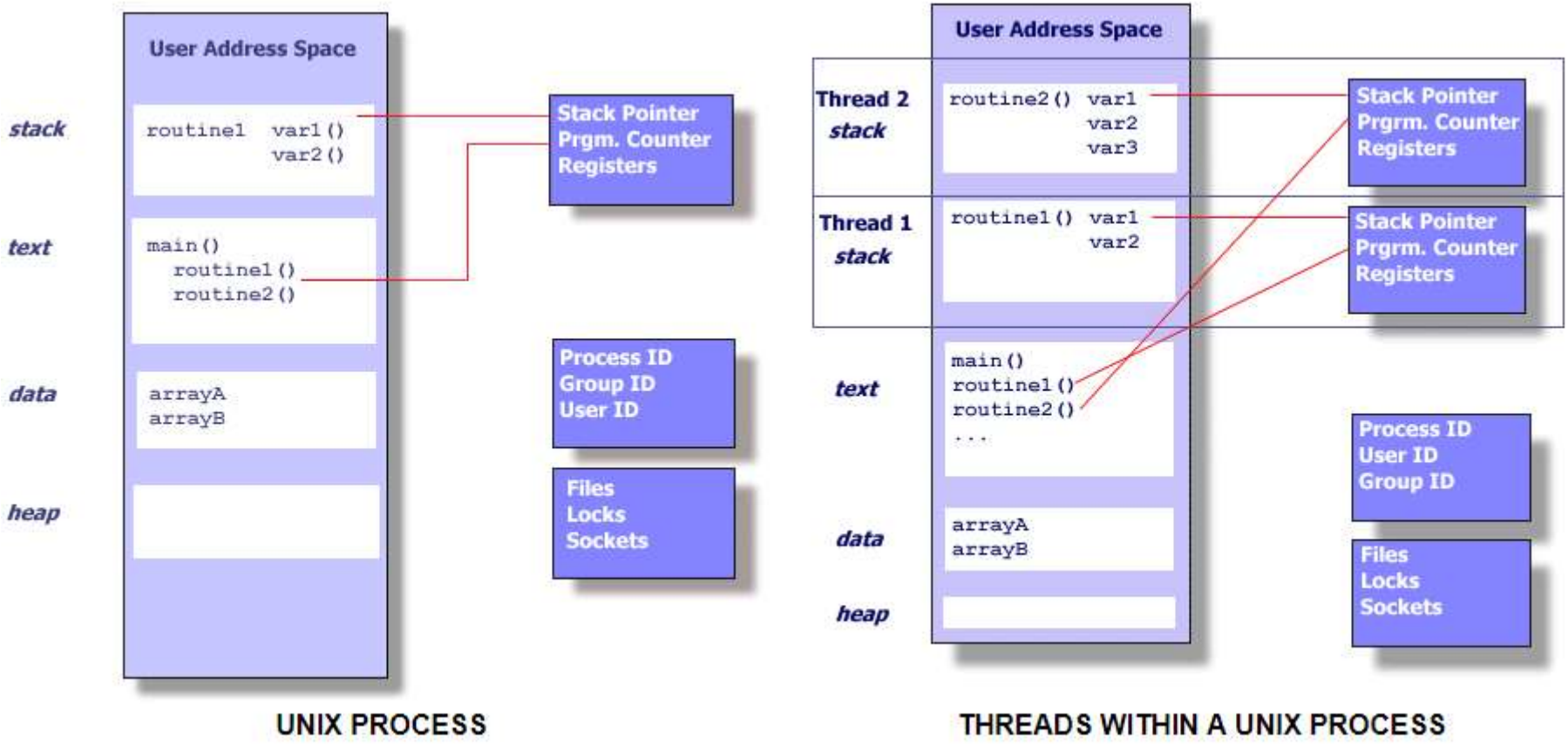
OpenMP

- Industry standard, jointly defined and endorsed by a group of major computer hardware and software vendors, organizations and individuals.
- Compiler directive based
- Portable / multi-platform, including Unix and Windows platforms
- Available in C/C++ and Fortran implementations
- Can be very easy and simple to use - provides for "incremental parallelism". Can begin with serial code.
- OpenMP tutorial: computing.llnl.gov/tutorials/openMP

Processes and Threads



- Modern operating systems load programs as processes
  - Resource holder
  - Execution
- A process starts executing at its entry point as a thread
- Threads can create other threads within the process
- All threads within a process share code & data segments

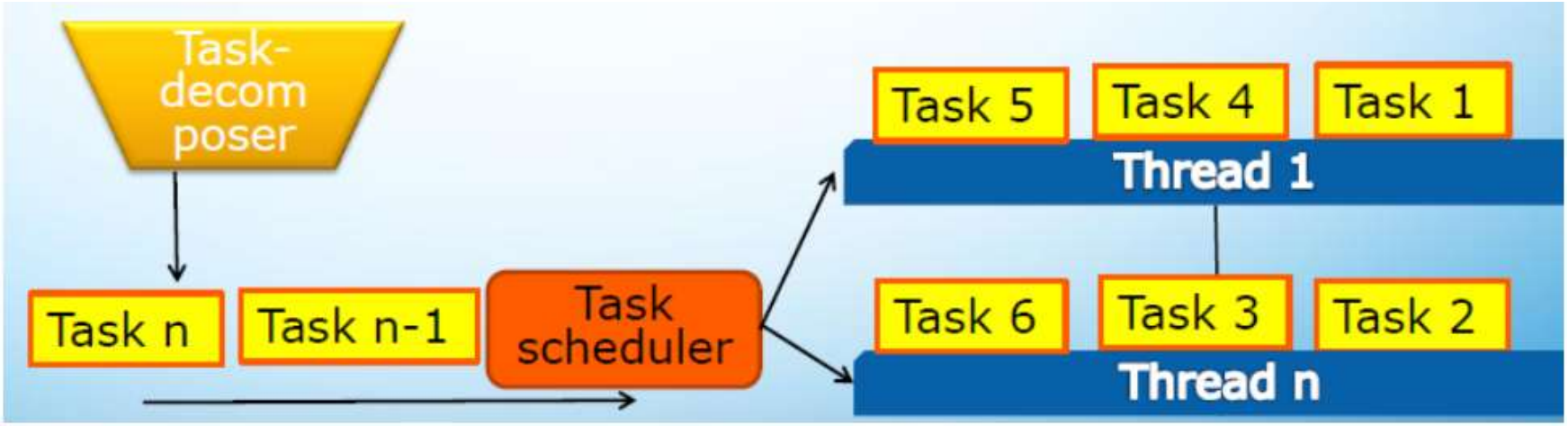

User Address Space
stack
routine1 var1()
var2()
text
main()
routine1()
routine2()
data
arrayA
arrayB
heap
Stack Pointer
Prgm. Counter
Registers
Process ID
Group ID
User ID
Files
Locks
Sockets
UNIX PROCESS
User Address Space
Thread 2
stack
routine2() var1
var2
var3
Thread 1
stack
routine1() var1
var2
text
main()
routine1()
routine2()
...
data
arrayA
arrayB
heap
Stack Pointer
Prgrm. Counter
Registers
Stack Pointer
Prgrm. Counter
Registers
Process ID
User ID
Group ID
Files
Locks
Sockets
THREADS WITHIN A UNIX PROCESS

Fork–Join

- The fork–join construct is a simple concept for the creation of processes or threads, it was originally developed for process creation, but the pattern can also be used for threads
  - Using the concept, an existing thread *T* creates a number of child threads *T*1*, . . . , Tm* with a fork statement. The child threads work in parallel and execute a given program part or function.
  - The creating parent thread T can execute the same or a different program part or function and can then wait for the termination of T1, . . . , Tm by using a join call

Task and Thread

- A task consists the data and its process, and task scheduler will attach it to a thread to be executed
- Task operation is much cheaper than threading operation
- Ease to balance workload among threads by stealing
- Suit for list, tree, map data structure



Task and Thread-II

- Considerations

  -Many more tasks than threads
  - More flexible to schedule the task
  - Easy to balance workload

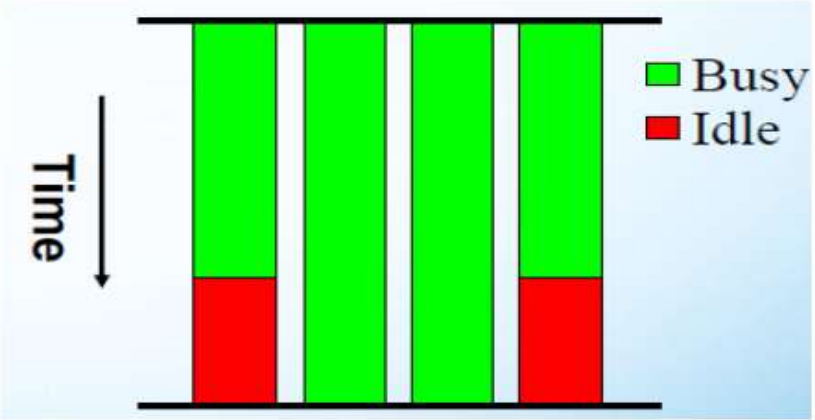
  -Amount of computation within a task must be large enough to offset overhead of managing task and thread

  -Static scheduling
  - Tasks are collections of separate, independent function calls or are loop iterations

  -Dynamic scheduling
  - Task execution length is variable and is unpredictable
  - May need an additional thread to manage a shared structure to hold all tasks

Imbalanced Workload

- All threads process the data in same way, but one thread is assigned more work, thus require more time to complete it and impact overall performance



- Considerations
  - Parallelize the inner loop
  - Incline to fine-grained
  - Choice the proper algorithm
  - Divide and conquer, master and worker, work-stealing

Granularity

- A extent to which a larger entity is subdivided
- Coarse-grained means fewer and larger components
- Fine-grained means more and smaller components
- Consideration
  - Fine-grained will increase the workload for task scheduler
  - Coarse-grained may cause the workload imbalance
  - Benchmark to set the proper granularity

Race Conditions

- Threads "*race*" against each other for resources
  -Execution order is assumed but cannot be guaranteed
- Storage conflict is most common
  -Concurrent access of same memory location by multiple threads, at least one thread is writing
- Determinacy race and data race
- May not be apparent at all times
- Considerations

  -Control shared access with critical regions
  - Mutual exclusion and synchronization, critical session, atomic

  -Scope variables to be local to threads
  - Have a local copy for shared data
  - Allocate variables on thread stack

Deadlock

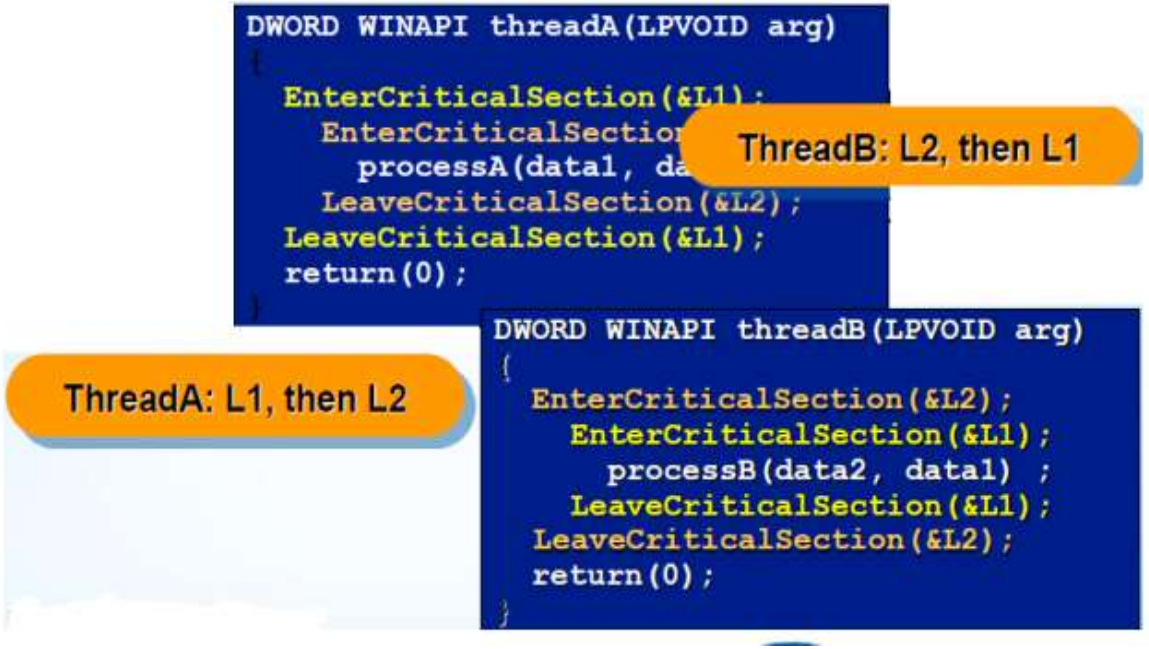
- 2 or more threads wait for each other to release a resource
- A thread waits for a event that never happen, like suspended lock
- Most common cause is locking hierarchies
- Considerations
  - Always lock and un-lock in the same order, and avoid hierarchies if possible
  - Use atomic

Thread T_1

```
lock(s1);
lock(s2);
do_work();
unlock(s2)
unlock(s1)
```

Thread T_2

```
lock(s2);
lock(s1):
do_work();
unlock(s1)
unlock(s2)
```

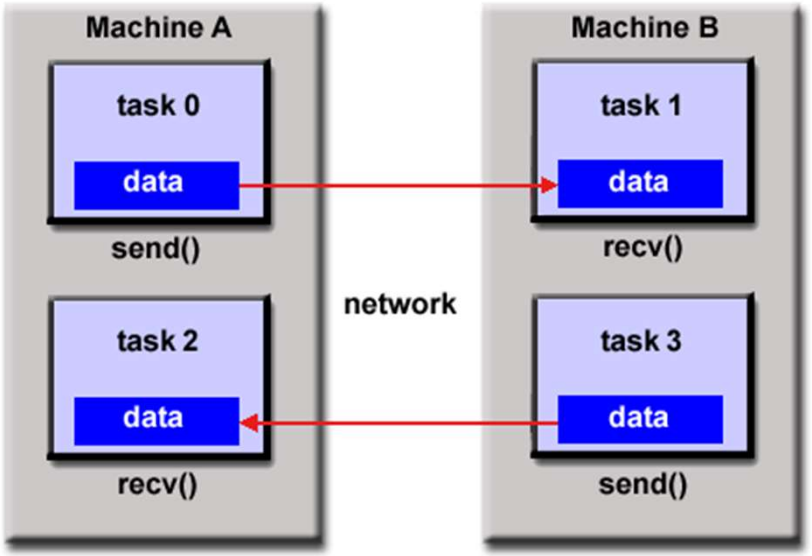


Summary for Threads Model

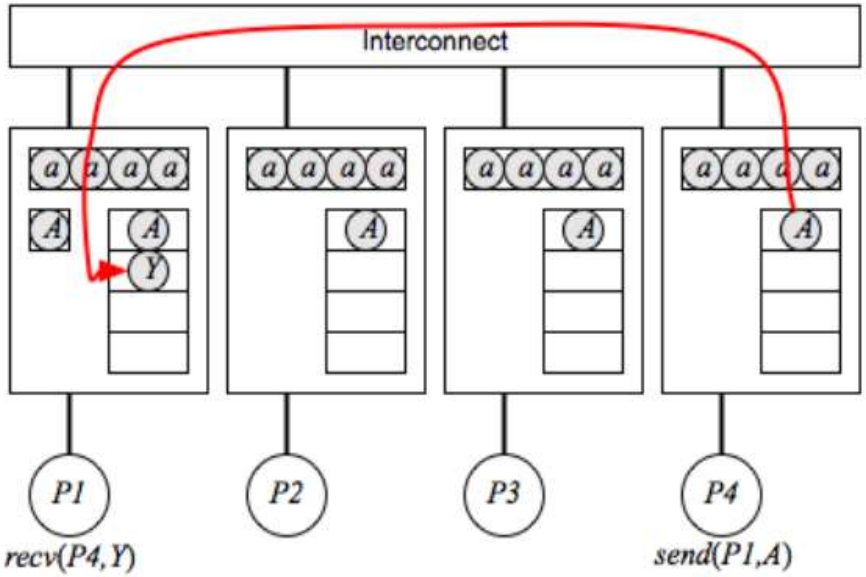
- Threading applications require multiple iterations of designing, debugging, and performance tuning steps
- Use tools to improve productivity
- Unleash the power of dual-core and multi-core processors

3. Distributed Memory / Message Passing Model

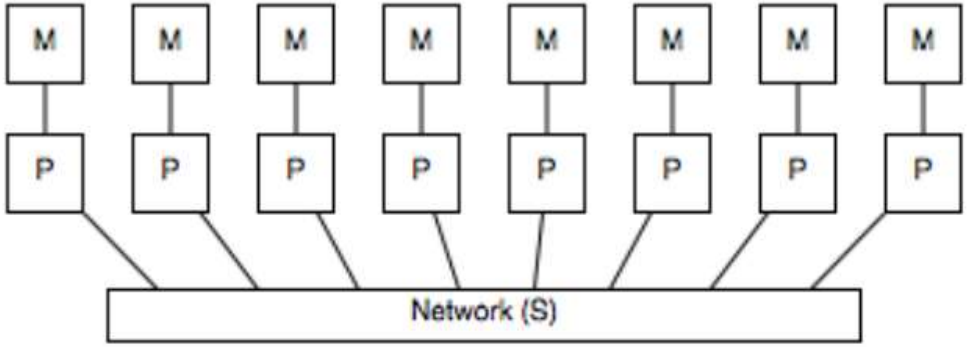
- A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.
- Tasks exchange data through communications by sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.



- Each node has a network interface
  - Communication and synchronization via network
  - Message latency and bandwidth is dependent on network topology and routing algorithms

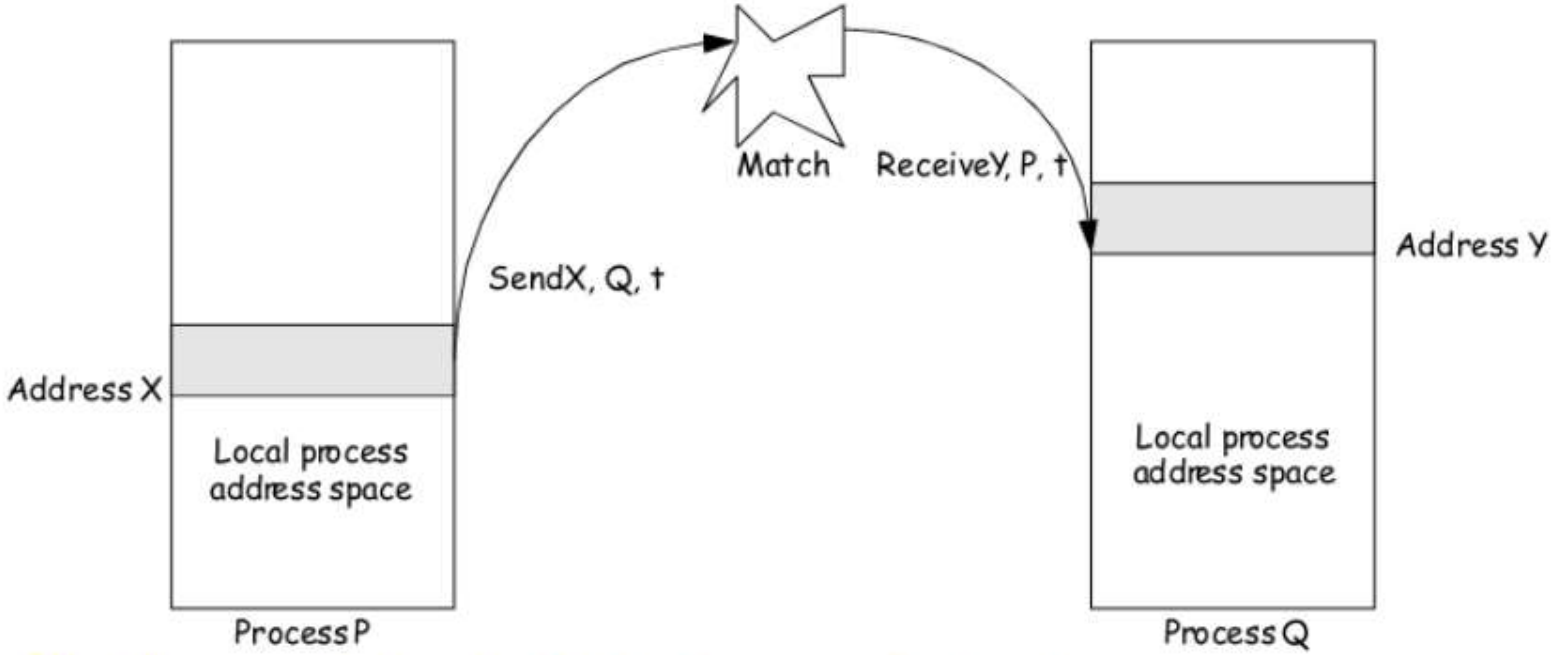


Programming model



Machine model

Message Passing Abstraction



- Send specifies buffer to be transmitted and sending process
- Recv specifies receiving process and application storage to receive into
- Memory to memory copy, but need to name processes
- Optional tag on send and matching rule on receive
- Many overheads: copying, buffer management, protection

Implementations:

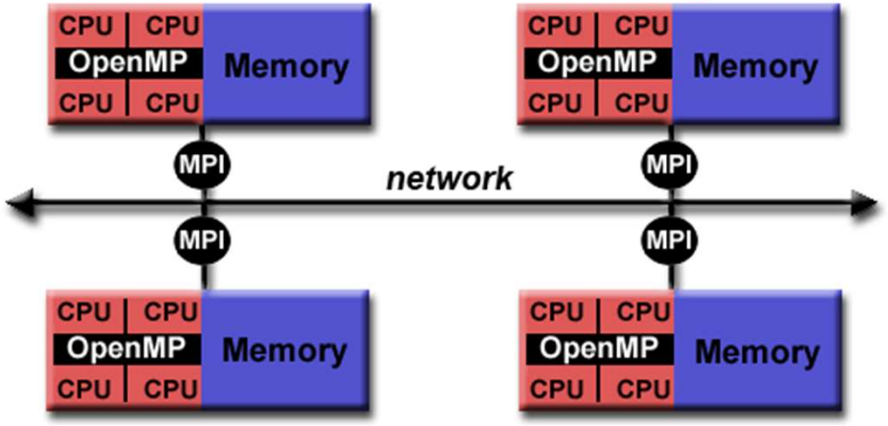
- From a programming perspective, message passing implementations usually comprise a library of subroutines. Calls to these subroutines are imbedded in source code. The programmer is responsible for determining all parallelism.
- Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.

Implementations-II

- Part 1 of the **Message Passing Interface (MPI)** was released in **1994**
- Part 2 (MPI-2) was released in **1998**
- MPI-3 in **2012**
- **MPI-4 in 2021**
- All MPI specifications are available on the web at http://www.mpi-forum.org/docs/.
- MPI is the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work.
- MPI implementations exist for virtually all popular parallel computing platforms. Not all implementations include everything in MPI-1, MPI-2, MPI-3 or MPI-4

4. Hybrid Model

- A hybrid model combines more than one of the previously described programming models.
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with the threads model (OpenMP).
  - Threads perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI
- This hybrid model lends itself well to the most popular (currently) hardware environment of clustered multi/many-core machines.



- Another similar and increasingly popular example of a hybrid model is using MPI with CPU-GPU (Graphics Processing Unit) programming.
  - MPI tasks run on CPUs using local memory and communicating with each other over a network.
  - Computationally intensive kernels are off-loaded to GPUs on-node.
  - Data exchange between node-local memory and GPUs uses CUDA (or something equivalent).
- Other hybrid models are common:
  - MPI with Pthreads
  - MPI with non-GPU accelerators
  - ...

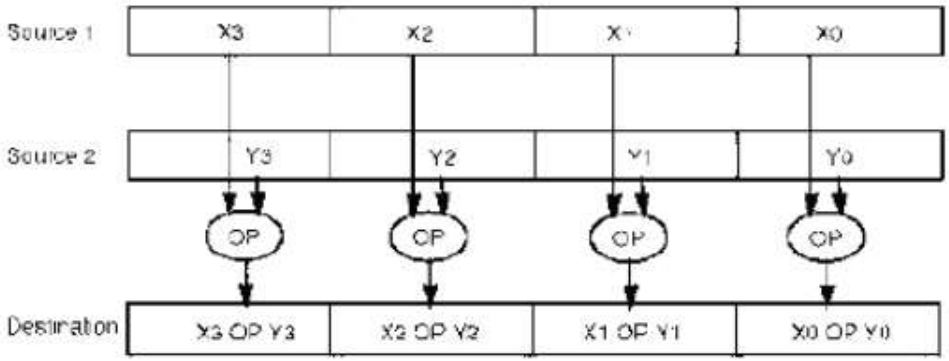
5. GPGPU Programming Model

- **CUDA(Compute Unified Device Architecture)**
  -NVIDIA
- **OpenCL(Open Computing Language)**
  -Nvidia, AMD, IBM, Intel
- **OpenACC**
- **DPC++ (Data Parallel C++)**
  -Intel

```
#pragma acc data copy(A) create(Anew)
while ( error > tol  &&  iter  <  iter_max )  {
  error = 0.0;
#pragma acc kernels {
#pragma acc loop independent collapse(2)
  for (  int  j = 1; j < n-1;  j++ )  {
    for (  int  i = 1; i < m-1; i++ )  {
      Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] +
                              A [j-1] [i] + A [j+1] [i]);
      error = max ( error, fabs (Anew [j] [i] - A [j] [i]));
    }
  }
 }
}
```

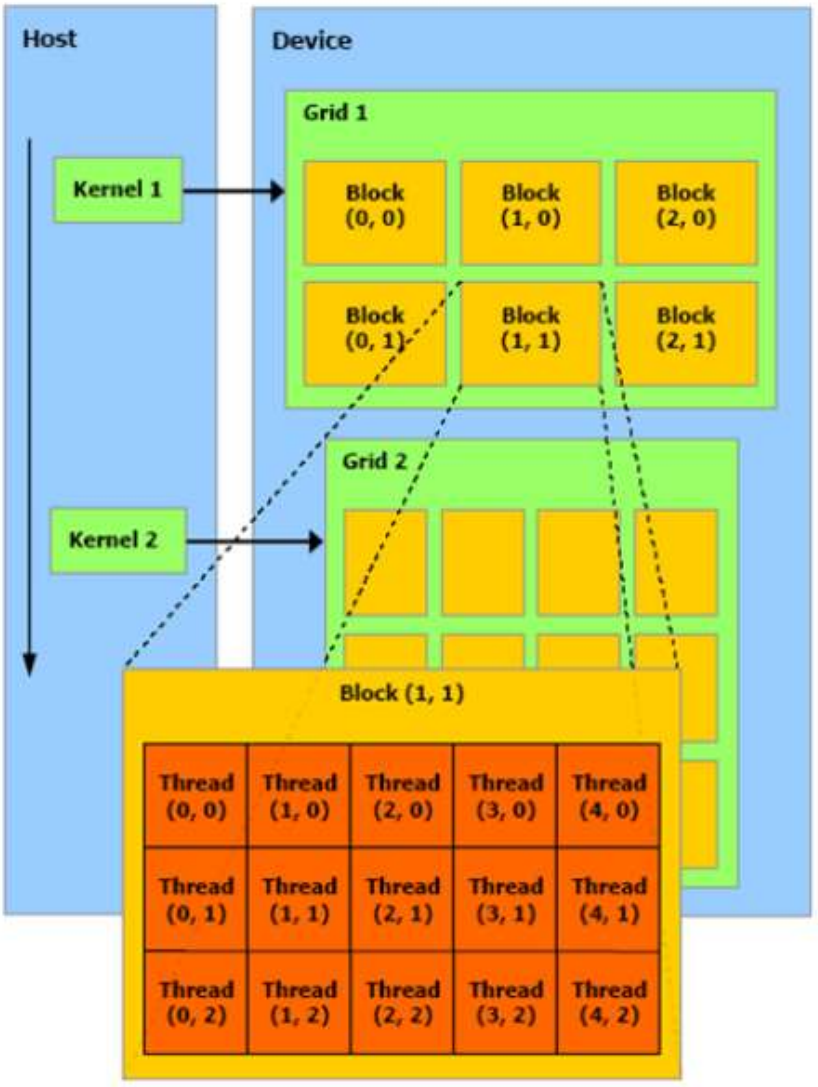
CUDA Goals: SIMD Programming

4 way SIMD (SSE) 16 way SIMD (LRB)

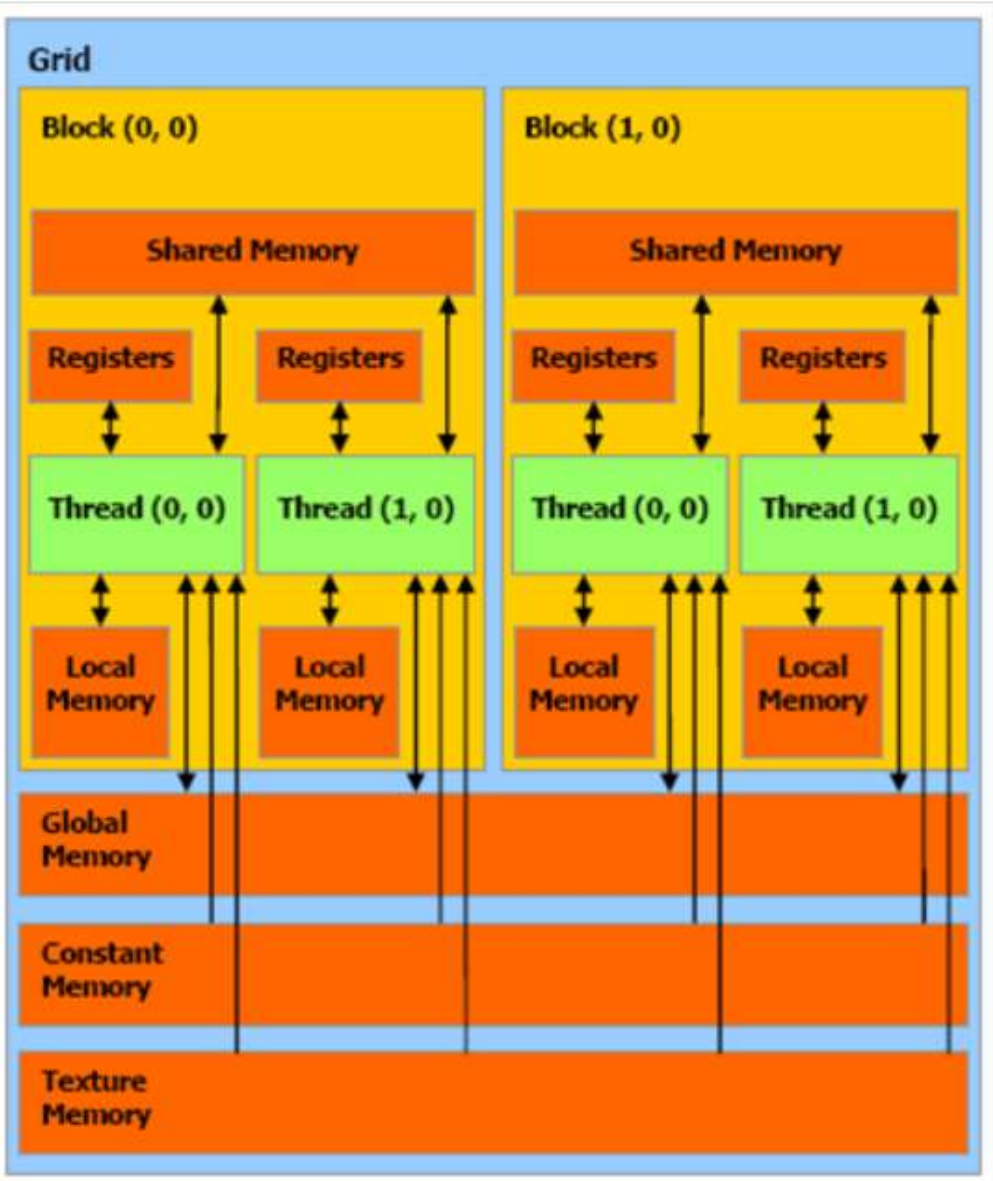


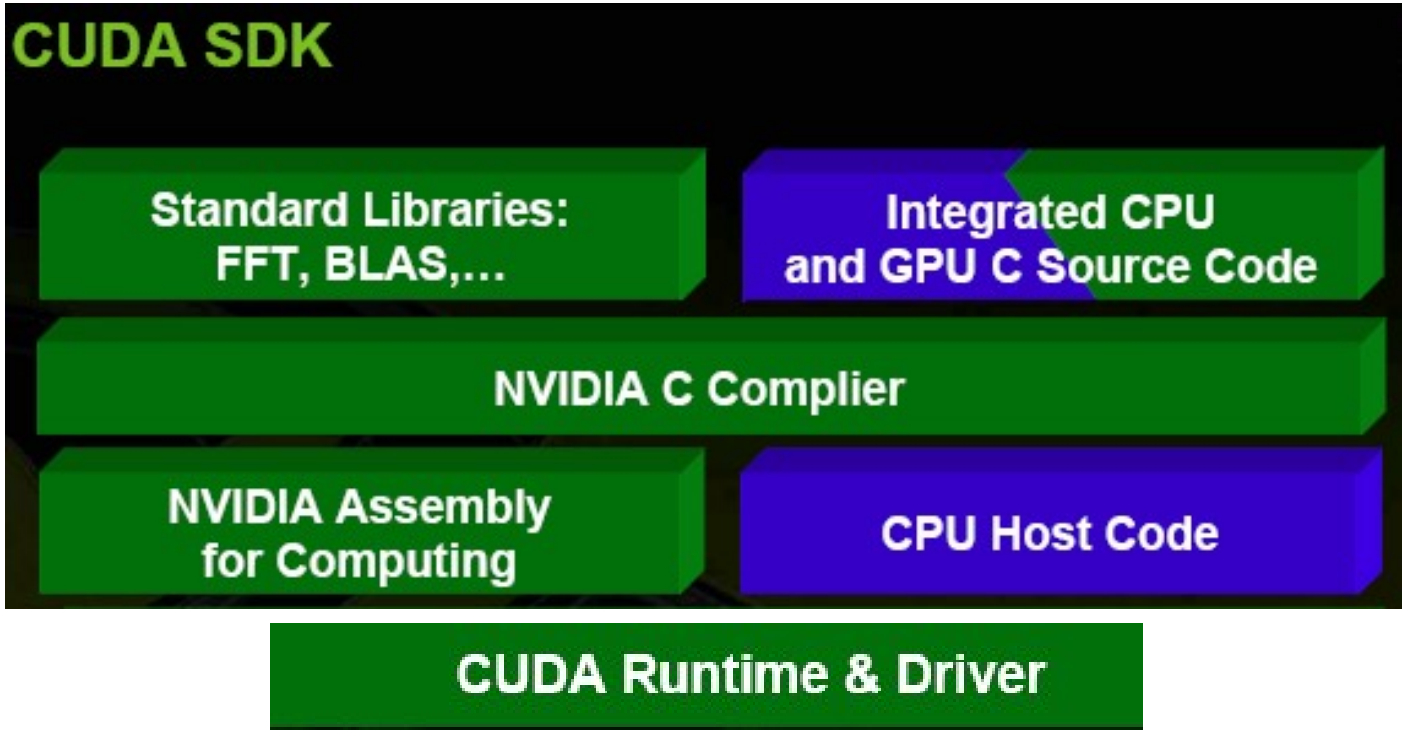
- Hardware architects love SIMD, since it permits a very space and energy-efficient implementation
- However, standard SIMD instructions on CPUs are inflexible, and difficult to use, difficult for a compiler to target
- CUDA thread abstraction will provide programmability at the cost of additional hardware

CUDA Programming Model



The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks

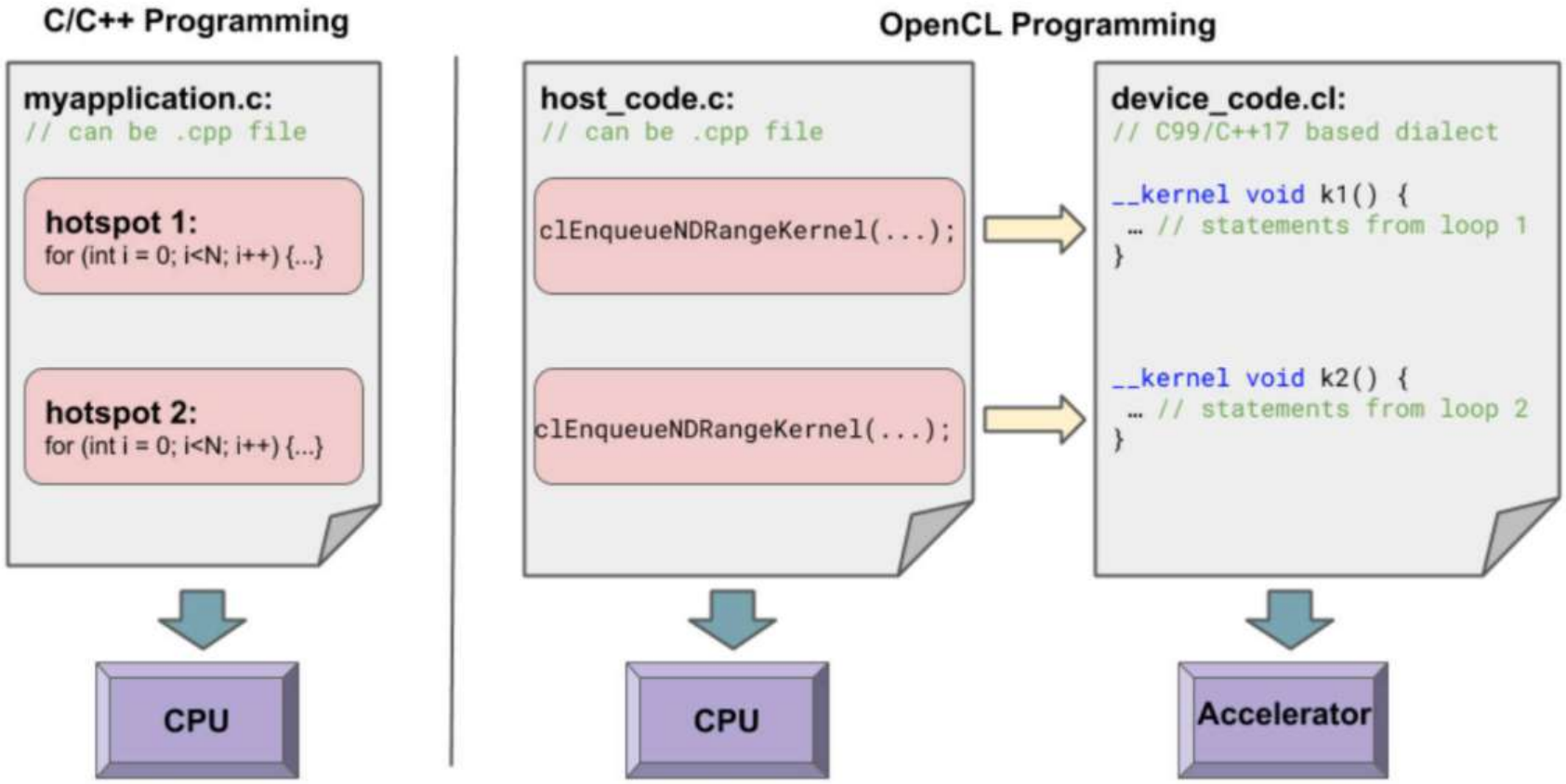





```
(base) [houzx@gpu1 ~]$ ls /usr/local/cuda-10.1/
bin   cuda-10.0  EULA.txt  include  libnsight  nsight-compute-2019.4.0  nsight-systems-2019.3.7.5  nvvm     share  targets  version.txt
cuda  doc        extras    lib64    libnvvp    nsightee_plugins         nvml                       samples  src    tools
(base) [houzx@gpu1 ~]$ ls /usr/local/cuda-10.1/bin/
bin2c        cuda-gdb                  cuda-uninstaller  nsight                      nsys-exporter  nvlink           nvprune
computeprof  cuda-gdbserver            cuobjdump         nsight_ee_plugins_manage.sh  nvcc           nv-nsight-cu     nvvp
crt          cuda-install-samples-10.1.sh  fatbinary      nsight-sys                   nvcc.profile   nv-nsight-cu-cli ptxas
cudafe++     cuda-memcheck             gpu-library-advisor  nsys                      nvdisasm       nvprof
(base) [houzx@gpu1 ~]$
```

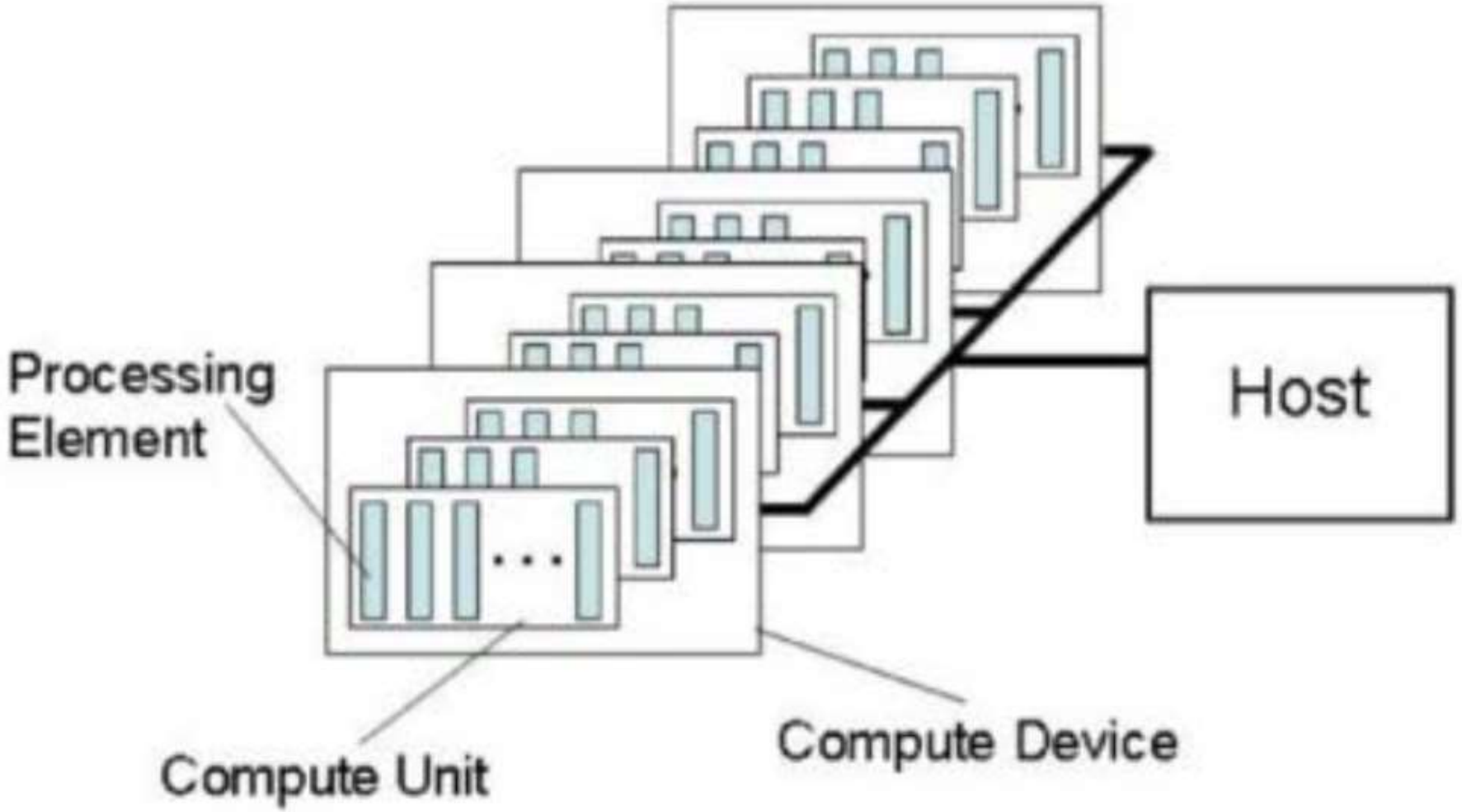
OpenCL Programming Model

- Data Parallel - SPMD
  - Work-items in a work-group run the same program
  - Update data structures in parallel using the work-item ID to select data and guide execution
- Task Parallel
  - One work-item per work group … for coarse grained task-level parallelism
  - Native function interface: trap-door to run arbitrary code from an OpenCL command-queue

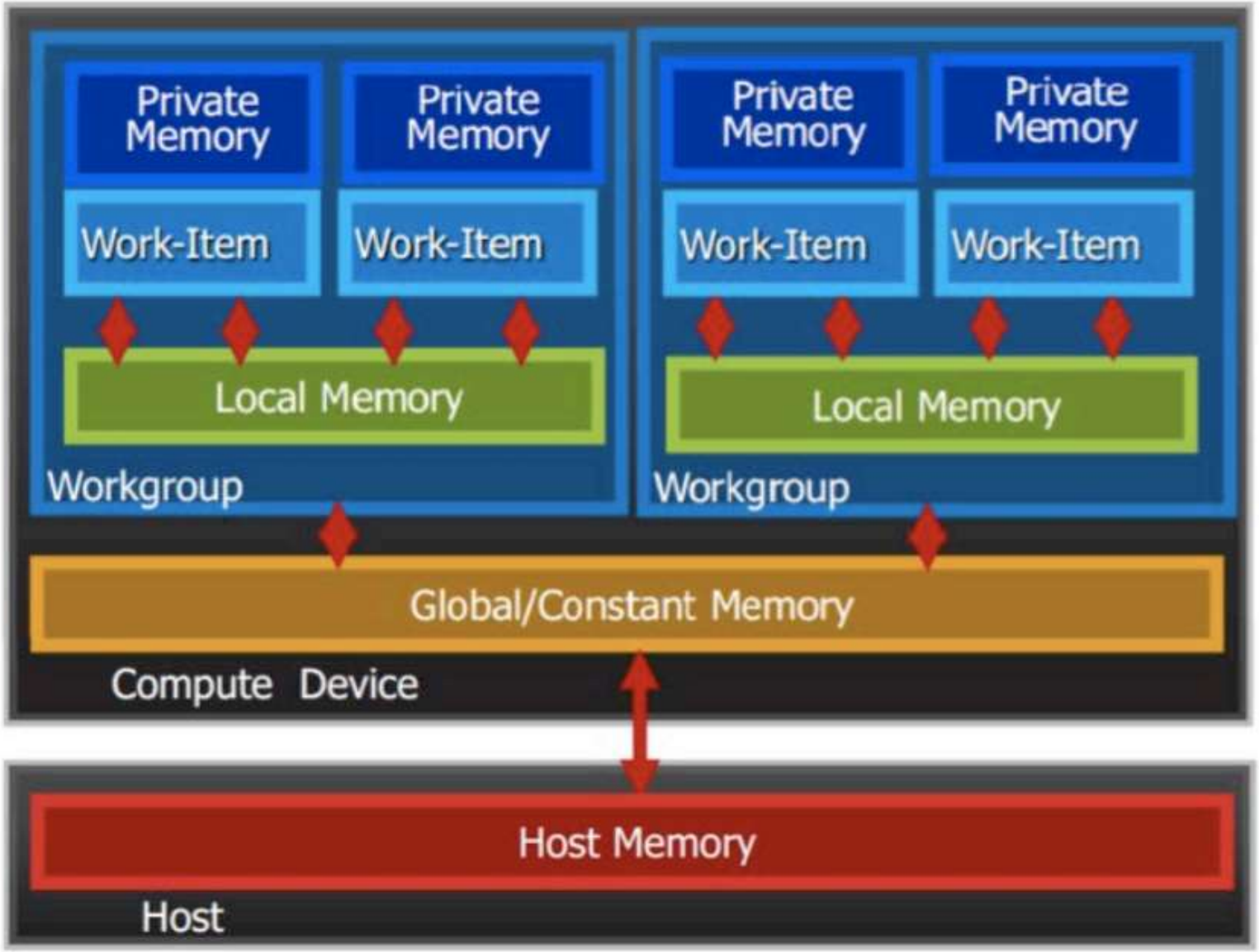


- **Traditional vs OpenCL programming paradigm**

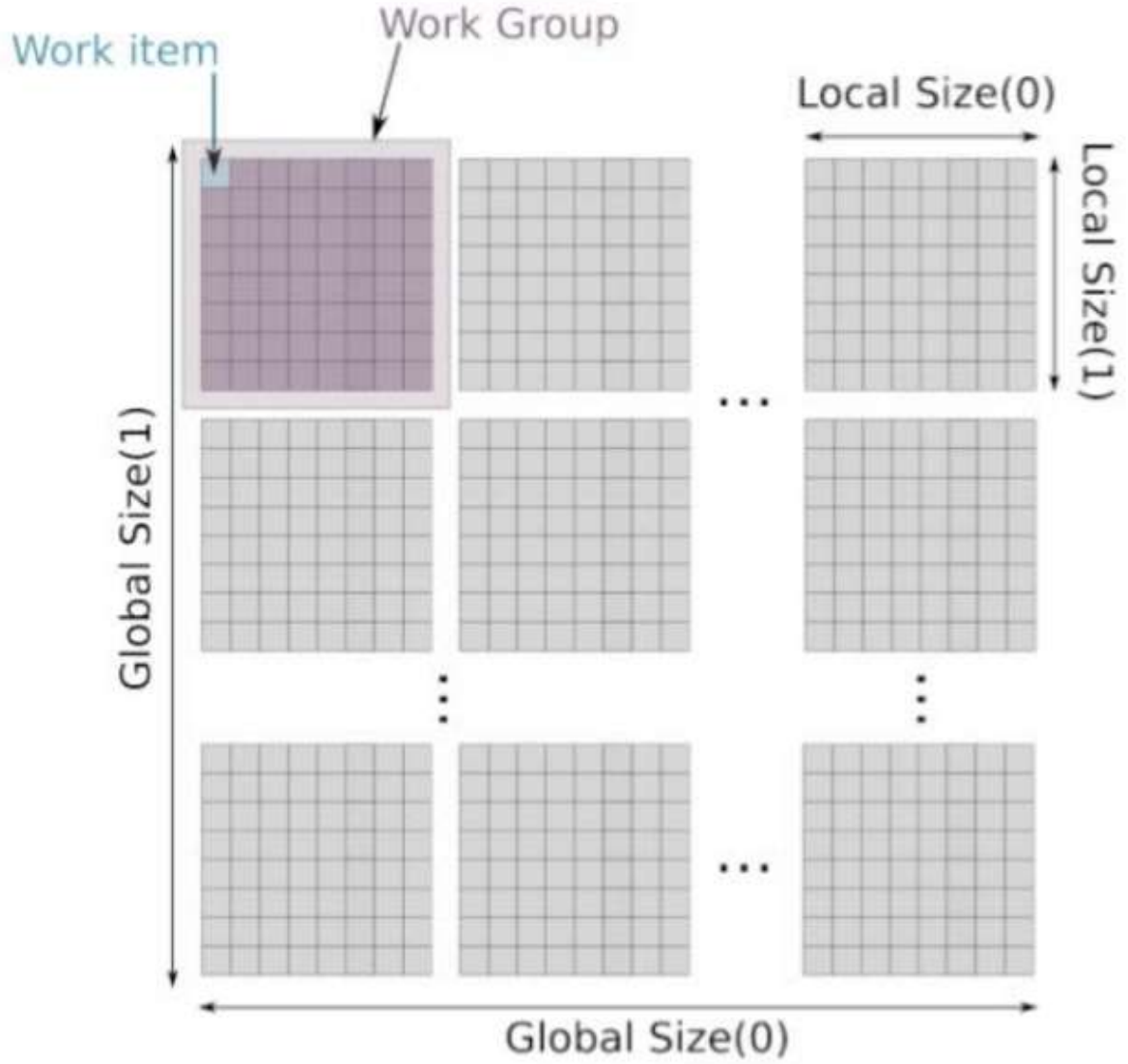
OpenCL Platform Model



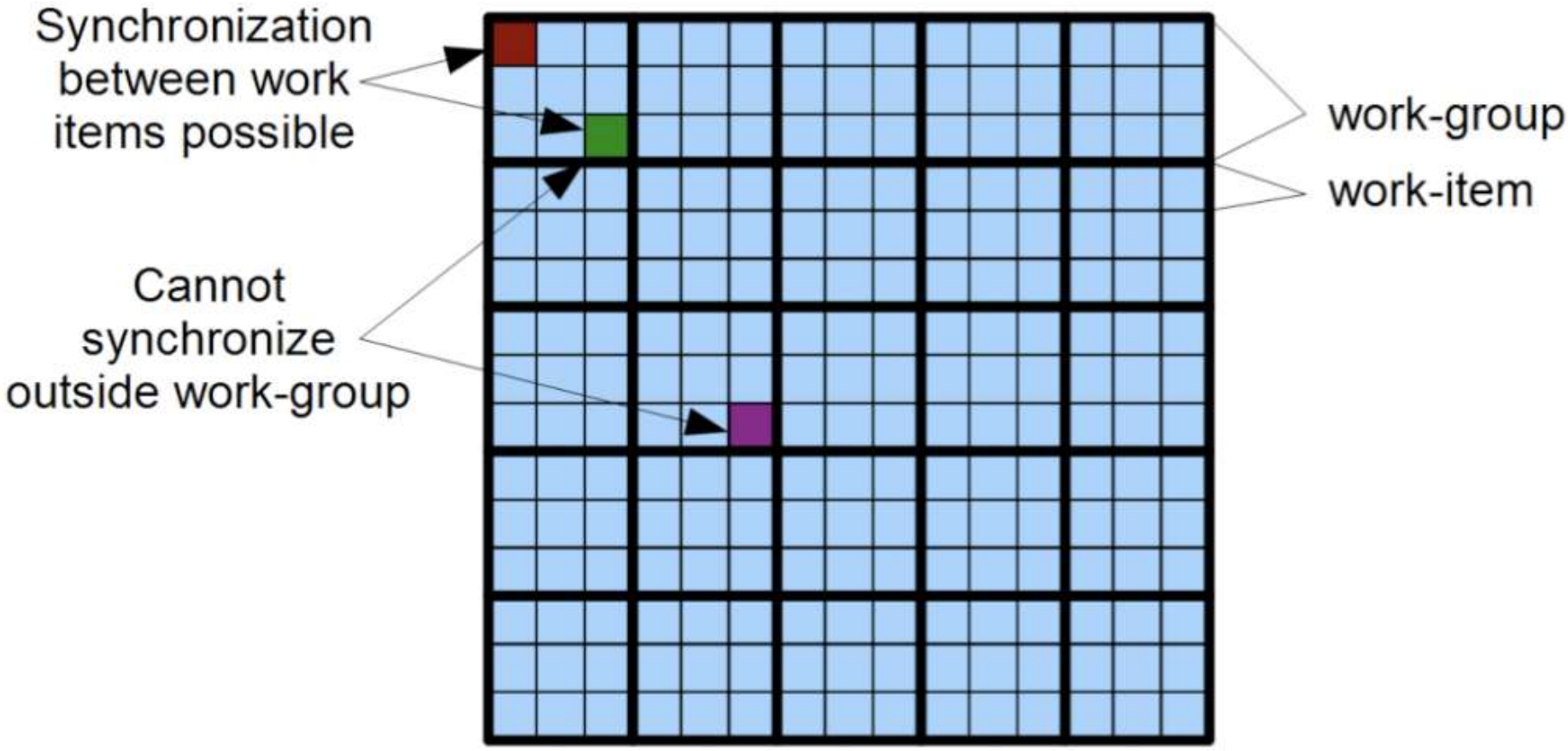
OpenCL Memory Model



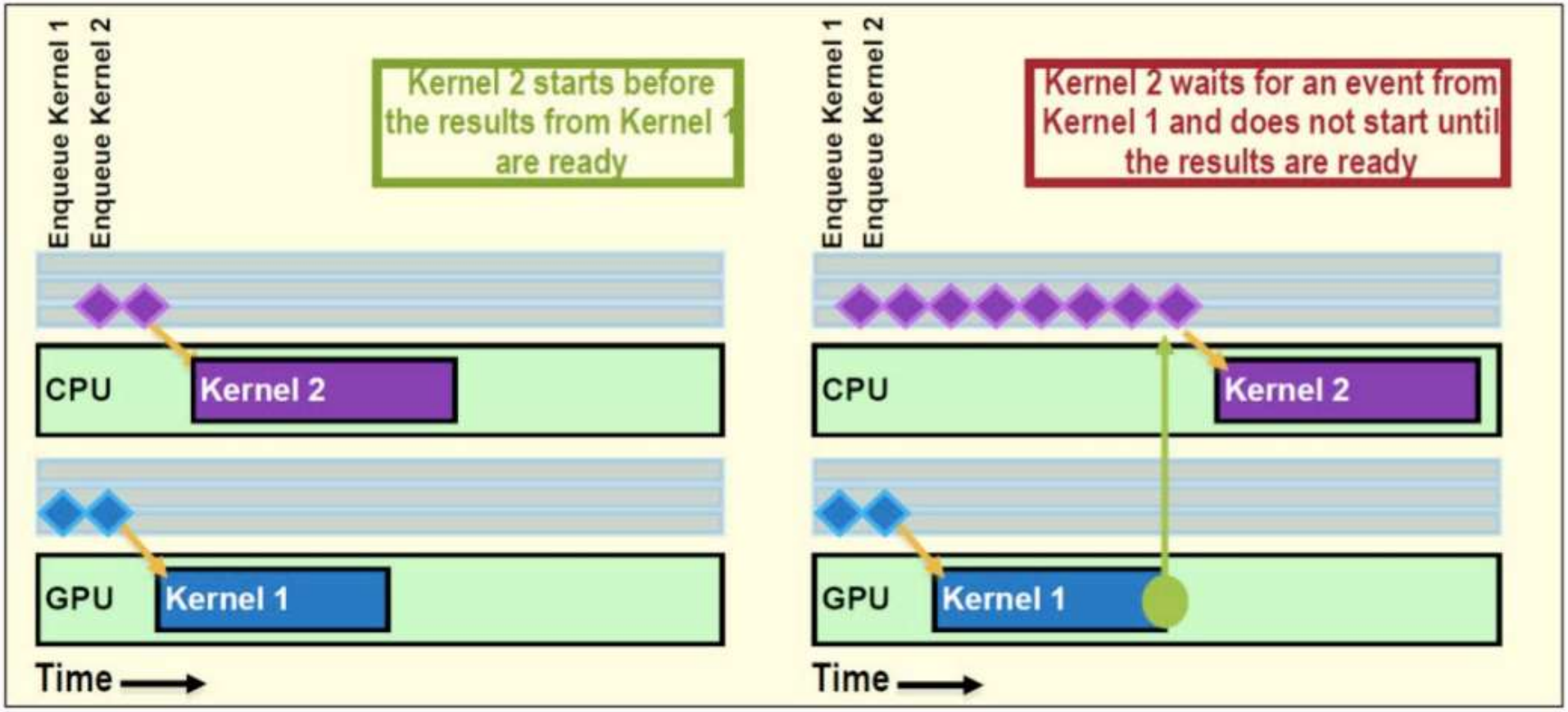
2D Data-Parallel execution in OpenCL



OpenCL Work-group / Work-unit Structure



Concurrency Control with OpenCL Event-Queueing



OpenCL's Two Styles of Data Parallelism

- Explicit SIMD data parallelism
  - The kernel defines one stream of instructions
  - Parallelism from using wide vector types
  - Size vector types to match native HW width
  - Combine with task parallelism to exploit multiple cores
- Implicit SIMD data parallelism (i.e. shader-style)
  - Write the kernel as a "scalar program"
  - Use vector data types sized naturally to the algorithm
  - Kernel automatically mapped to SIMD-compute-resources and cores by the compiler/runtime/hardware

Both approaches are viable CPU options

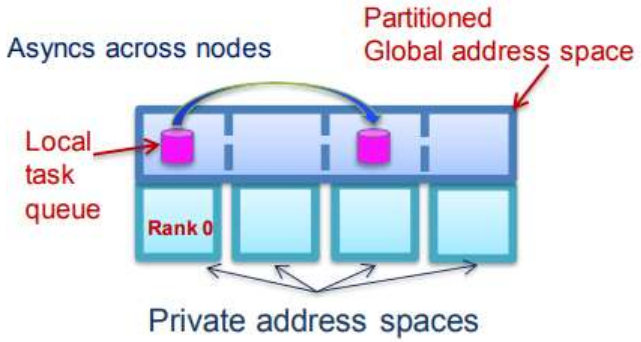
6. Data Parallel Model

Task versus Data Parallel

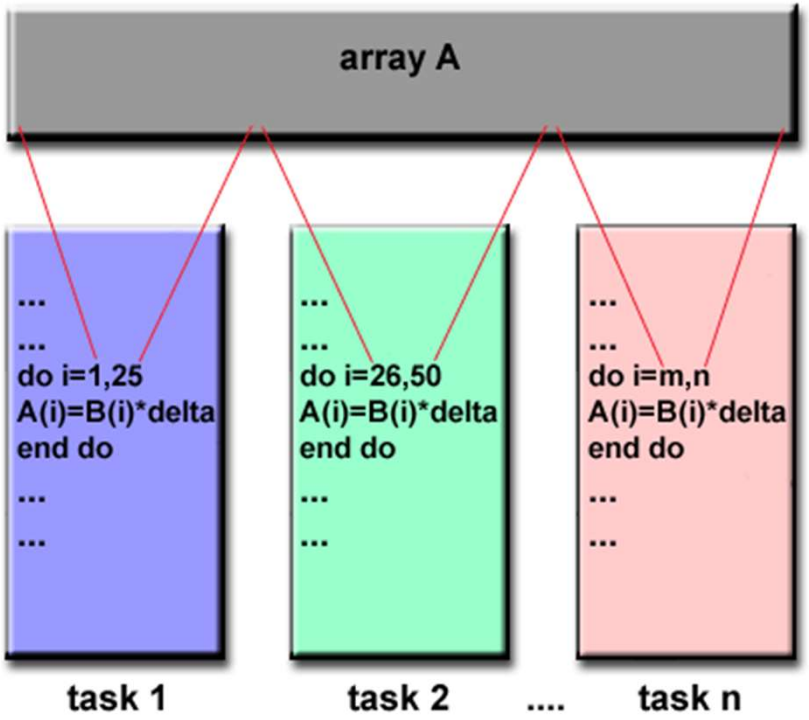
- Task parallel (maps to high-level MIMD machine model)
  - Task differentiation, like restaurant cook, waiter, and receptionist
  - Communication via shared address space or message passing
  - Synchronization is explicit (via locks and barriers)
  - Underscores operations on private data, explicit constructs for communication of shared data and synchronization
- Data parallel (maps to high-level SIMD machine model)
  - Global actions on data by tasks that execute the same code
  - Communication via shared memory or logically shared address space with underlying message passing
  - Synchronization is implicit (lock-step execution)
  - Underscores operations on shared data, private data must be defined explicitly or is simply mapped onto shared data space

Partitioned Global Address Space (PGAS) model

- Address space is treated globally
- Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
- A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
- Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".



- 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.



Implementations of PGAS:

- PGAS (Partitioned Global Address Space) languages attempt to combine the convenience of the global view of data with awareness of data locality

- Currently, there are several relatively popular, and sometimes developmental, parallel programming implementations based on the Data Parallel / PGAS model.

-Unified Parallel C (UPC): an extension to the C programming language for SPMD parallel programming. Compiler dependent. More information: http://upc.lbl.gov/

-Coarray Fortran: a small set of extensions to Fortran 95 for SPMD parallel programming. Compiler dependent. More information: http://www.co-array.org/

- **Global Arrays:** provides a shared memory style programming environment in the context of distributed array data structures. Public domain library with C and Fortran77 bindings. More information:http://www.emsl.pnl.gov/docs/global/

- **X10:** a PGAS based parallel programming language being developed by IBM at the Thomas J. Watson Research Center. More information: http://x10-lang.org/

- **Chapel:** an open source parallel programming language project being led by Cray. More information: http://chapel.cray.com/

7. SPMD and MPMD

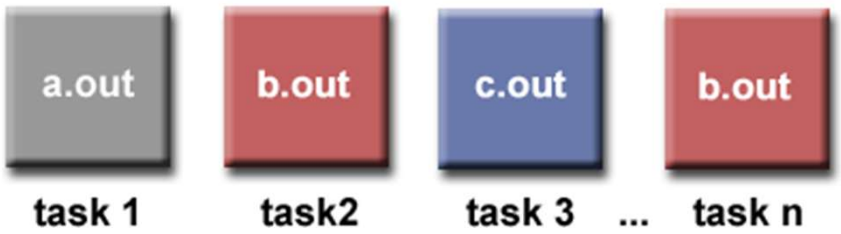
- **Single Program Multiple Data (SPMD):**
  - SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
  - SINGLE PROGRAM: All tasks execute their copy of the same program simultaneously. This program can be threads, message passing, data parallel or hybrid.
  - MULTIPLE DATA: All tasks may use different data

- SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.
- The SPMD model, using message passing or hybrid programming, is probably the most commonly used parallel programming model for multi-node clusters.

- The SPMD approach is one of the most popular models for parallel programming
- MPI is mainly based on this approach

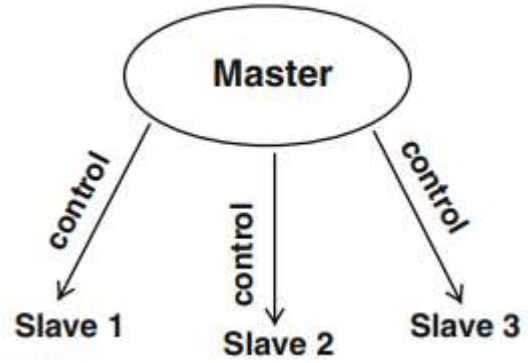
Multiple Program Multiple Data (MPMD):

- Like SPMD, MPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- MULTIPLE PROGRAM: Tasks may execute different programs simultaneously. The programs can be threads, message passing, data parallel or hybrid.
- MULTIPLE DATA: All tasks may use different data
- MPMD applications are not as common as SPMD applications, but may be better suited for certain types of problems, particularly those that lend themselves better to functional decomposition than domain decomposition

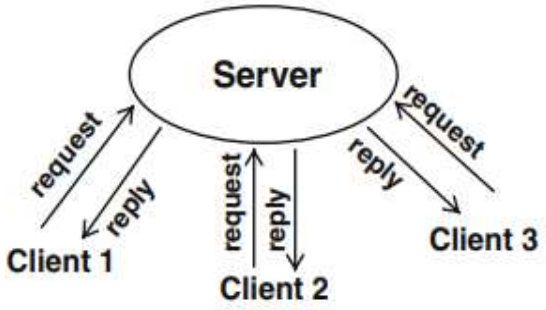


Master–Slave or Master–Worker

- In the SIMD and SPMD models, all threads have equal rights.
- In the master–slave model, also called master–worker model, there is one master which controls the execution of the program. The master thread often executes the main function of a parallel program and creates worker threads at appropriate program points to perform the actual computations

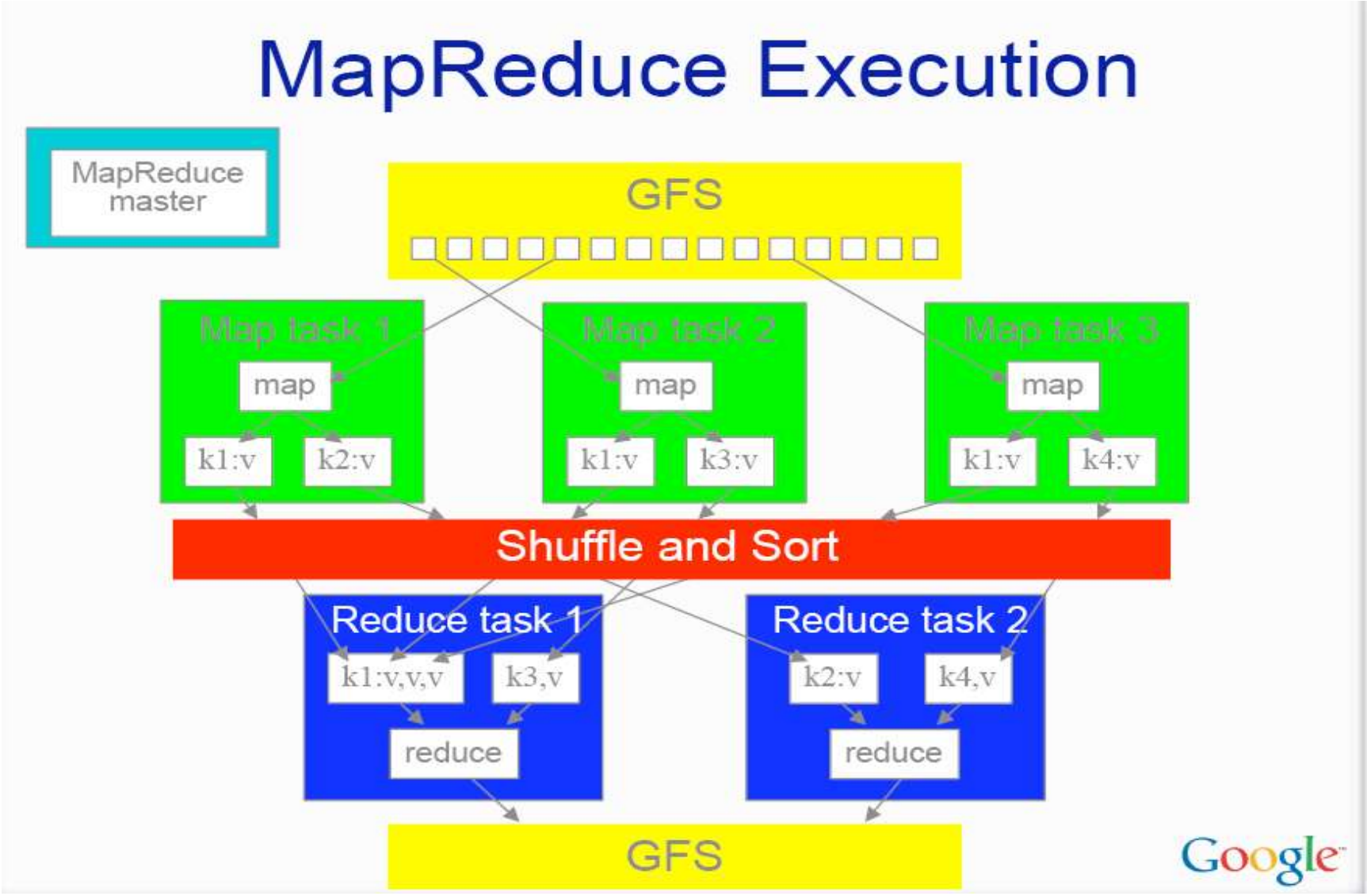


Client–Server



- The coordination of parallel programs according to the client–server model is similar to the general MPMD (**m**ultiple-**p**rogram **m**ultiple-**d**ata) model.
- The client–server model originally comes from distributed computing where multiple client computers have been connected to a mainframe which acts as a server and provides responses to access requests to a database
- After having processed a request of a client, the server delivers the result back to the client.
- The client–server model is important for parallel programming in heterogeneous systems and is also often used in grid computing and cloud computing

8. Data Intensive Computing Model



(Courtesy of Jeffrey Dean, Google, 2008)

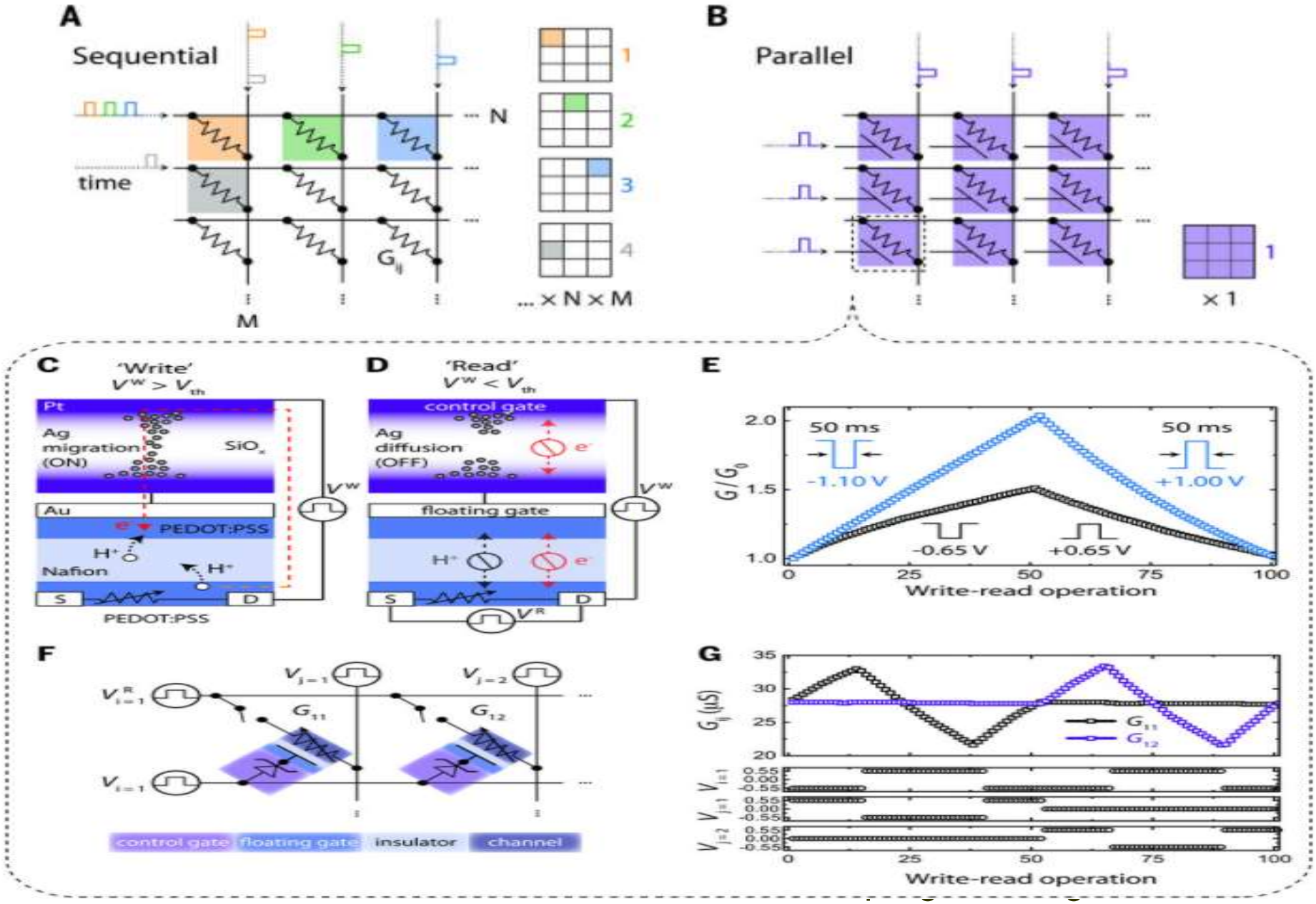
Parallel prog. model	Declarative	Deterministic	Efficient
Intel TBB	No	No	Yes
.Net Task Par. Lib.	No	No	Yes
Cilk	No	No	Yes
OpenMP	No	No	Yes
CUDA	No	No	Yes
Java Concurrency	No	No	Yes
Det. Parallel Java	No	Hybrid	Yes
High Perf. Fortran	Hybrid	No	Yes
X10	Hybrid	No	Yes
Linda	Hybrid	No	Yes
Asynch. Seq. Processes [3]	Yes	Yes	No
StreamIt	Yes	Yes	Yes
LabVIEW [11]	Yes	Yes	Yes
CnC	Yes	Yes	Yes

Programming Models for Parallel Computing

Criteria	Parallel Programming Models			
	MapReduce	*Cilk/Cilk++*	*OpenMP*	*MPI*
Organisation of the address space	Distributed / Shared	Shared	Shared	Distributed
Representation of parallelism	Implicit	Explicit	Explicit	Explicit
Data level parallelism	Yes	Yes	Yes	Yes
Task level parallelism	No	Yes	Yes	Yes
Data decomposition support	Yes	No	Yes	No
Incremental parallelism	No	Yes	Yes	No
Deterministic	Yes	No	No	No
Built-in load balancing	Yes	Yes	Yes	No
Supported languages	All	C/C++	Fortran, C/C++	Fortran, C/C++
Support for parallel programming	Library	Extension	Compiler directives	Library
Static / dynamic scheduling	Static	Dynamic	Static / Dynamic	Static / Dynamic
Static / dynamic mapping	Dynamic	Dynamic	Static / Dynamic	Static / Dynamic
Complexity	Simple	Complex	Simple	Complex
Level of Abstraction	High	Middle	Middle	Low

Rim Ketata, et al., Parallel Programming Models: A Survey, International Journal of Engineering Research & Technology, 2016.

ELLIOT J. FULLER, et al., Parallel programming of an ionic floating-gate memory array for scalable neuromorphic computing，Science, 2019.



Abstract

Neuromorphic computers could overcome efficiency bottlenecks inherent to conventional computing through parallel programming and readout of artificial neural network weights in a crossbar memory array. However, selective and linear weight updates and <10-nanoampere read currents are required for learning that surpasses conventional computing efficiency. We introduce an ionic floating-gate memory array based on a polymer redox transistor connected to a conductive-bridge memory (CBM). Selective and linear programming of a redox transistor array is executed in parallel by overcoming the bridging threshold voltage of the CBMs. Synaptic weight readout with currents <10 nanoamperes is achieved by diluting the conductive polymer with an insulator to decrease the conductance. The redox transistors endure >1 billion write-read operations and support >1-megahertz write-read frequencies.

Ionic floating-gate memories

Digital implementations of artificial neural networks perform many tasks, such as image recognition and language processing, but are too energy intensive for many applications. Analog circuits that use large crossbar arrays of synaptic memory elements represent a low-power alternative, but most devices cannot update the synaptic weights uniformly or scale to large array sizes. Fuller *et al.* developed an integrated device, ionic floating-gate memory, that has the gate terminal of a redox transistor electrically connected to a diffusive memristor. This low-power device enabled linear and symmetric weight updates in parallel over an entire crossbar array at megahertz rates over 10^9 write-read cycles.

References

- The main content expressed in this chapter comes from -Livermore Computing Center's training materials, (https://computing.llnl.gov/tutorials/parallel_comp/)
- Prof. Hai Jin

  from Huazhong University of Science and Technology

Thank you!