LA-UR-16-26424
Approved for public release; distribution is unlimited.

Title: Programming Models in HPC

Author(s): Shipman, Galen M.

Intended for: Make presentation available to LANL Parallel Computing summer school participants (externally).
Web

Issued: 2016-08-22

Los Alamos
NATIONAL LABORATORY
EST. 1943

# Programming Models in HPC

Parallel Computing Summer School

**Galen Shipman**

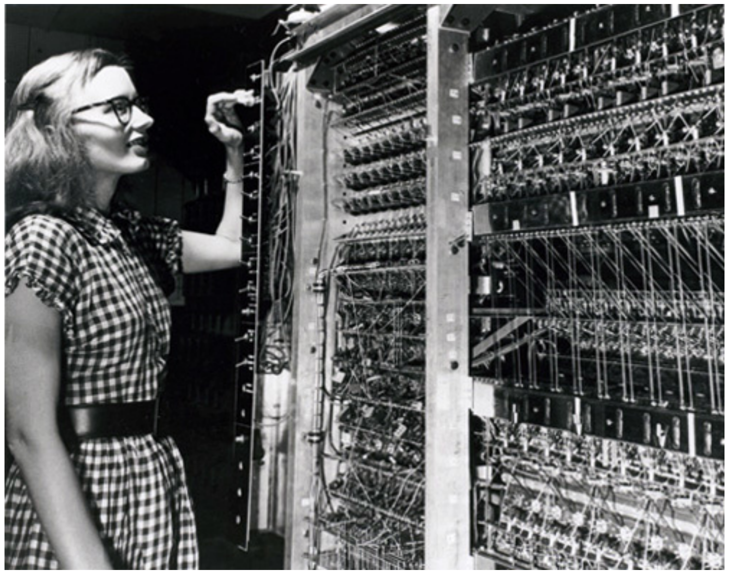
6/13/2016

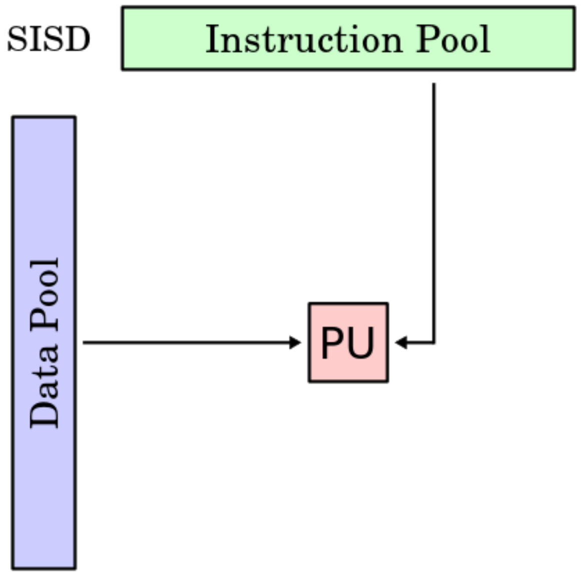
## Flynn’s Taxonomy of computer architectures
### *This taxonomy focuses on control mechanisms*

| | Single Instruction stream | Multiple Instruction streams |
|---|---|---|
| **Single data stream** | SISD | MISD |
| **Multiple data streams** | SIMD | MIMD |

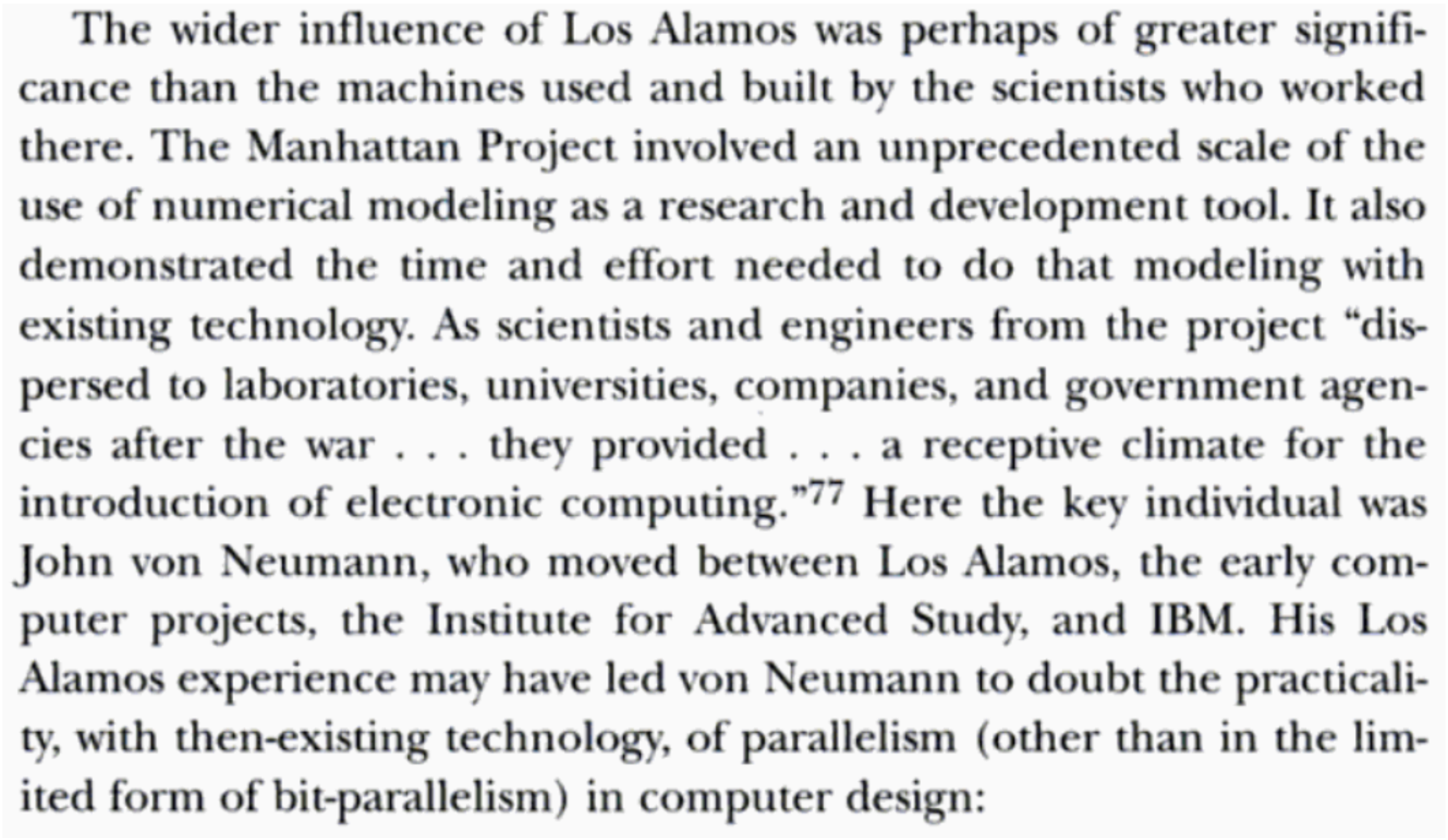
- **Single instruction stream, single data stream (SISD)**
  - Traditional uniprocessors
- **Single instruction stream, multiple data streams (SIMD)**
  - Vector units, GPU stream processors
- **Multiple instruction streams, single data stream (MISD)**
  - Generally specialized systems (e.g. Space shuttle flight control system)
- **Multiple instruction streams, multiple data streams (MIMD)**
  - Multi-core systems

## Single Instruction Single Data

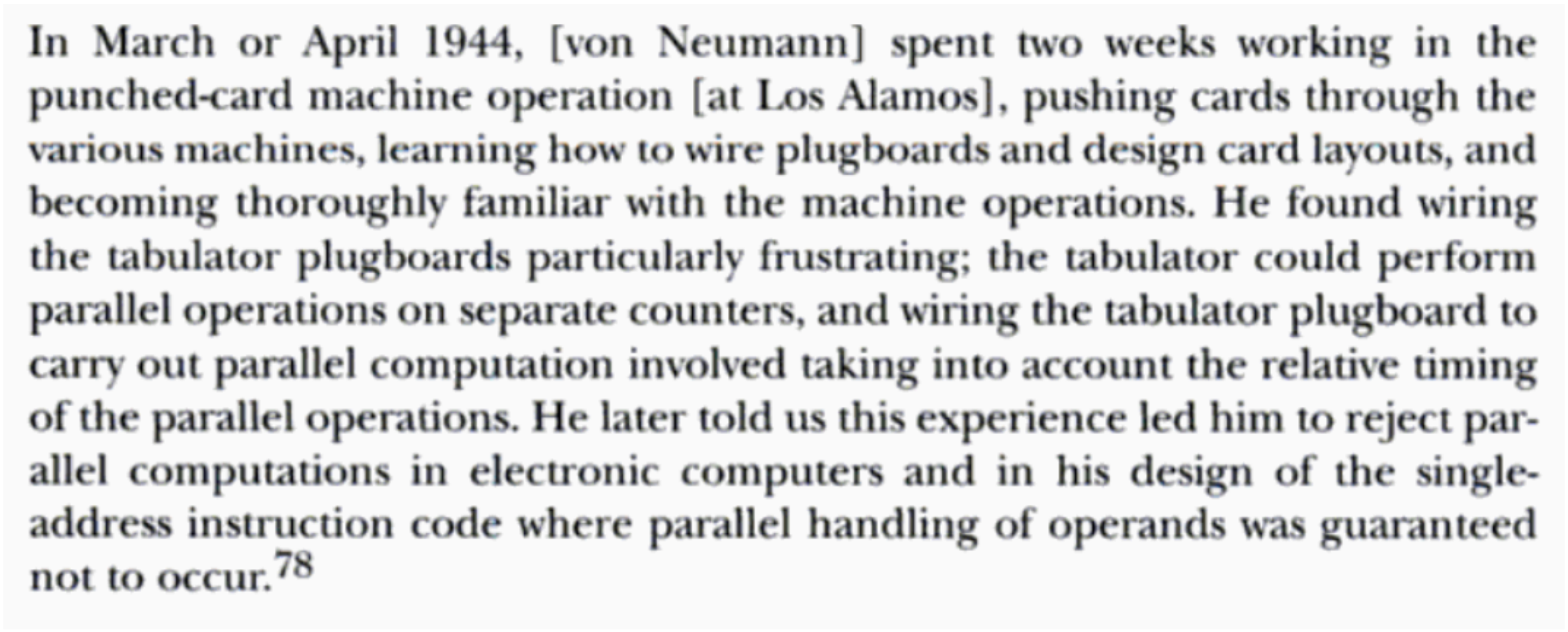




MANIAC at LANL –
John von Neumann & Nicholas
C. Metropolis



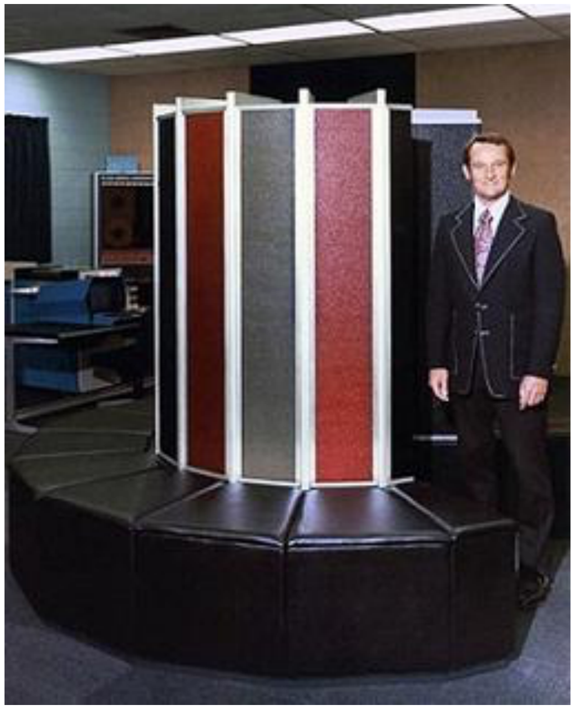
The wider influence of Los Alamos was perhaps of greater significance than the machines used and built by the scientists who worked there. The Manhattan Project involved an unprecedented scale of the use of numerical modeling as a research and development tool. It also demonstrated the time and effort needed to do that modeling with existing technology. As scientists and engineers from the project "dispersed to laboratories, universities, companies, and government agencies after the war . . . they provided . . . a receptive climate for the introduction of electronic computing."[77] Here the key individual was John von Neumann, who moved between Los Alamos, the early computer projects, the Institute for Advanced Study, and IBM. His Los Alamos experience may have led von Neumann to doubt the practicality, with then-existing technology, of parallelism (other than in the limited form of bit-parallelism) in computer design:



In March or April 1944, [von Neumann] spent two weeks working in the punched-card machine operation [at Los Alamos], pushing cards through the various machines, learning how to wire plugboards and design card layouts, and becoming thoroughly familiar with the machine operations. He found wiring the tabulator plugboards particularly frustrating; the tabulator could perform parallel operations on separate counters, and wiring the tabulator plugboard to carry out parallel computation involved taking into account the relative timing of the parallel operations. He later told us this experience led him to reject parallel computations in electronic computers and in his design of the single-address instruction code where parallel handling of operands was guaranteed not to occur.[78]

***MacKenzie, Donald, "Knowing Machines: Essays on Technical Change"***

## Single Instruction Multiple Data



Seymour Cray and the Cray-1, first installation was at LANL in 1976
Brought vector processing into wide-spread HPC use

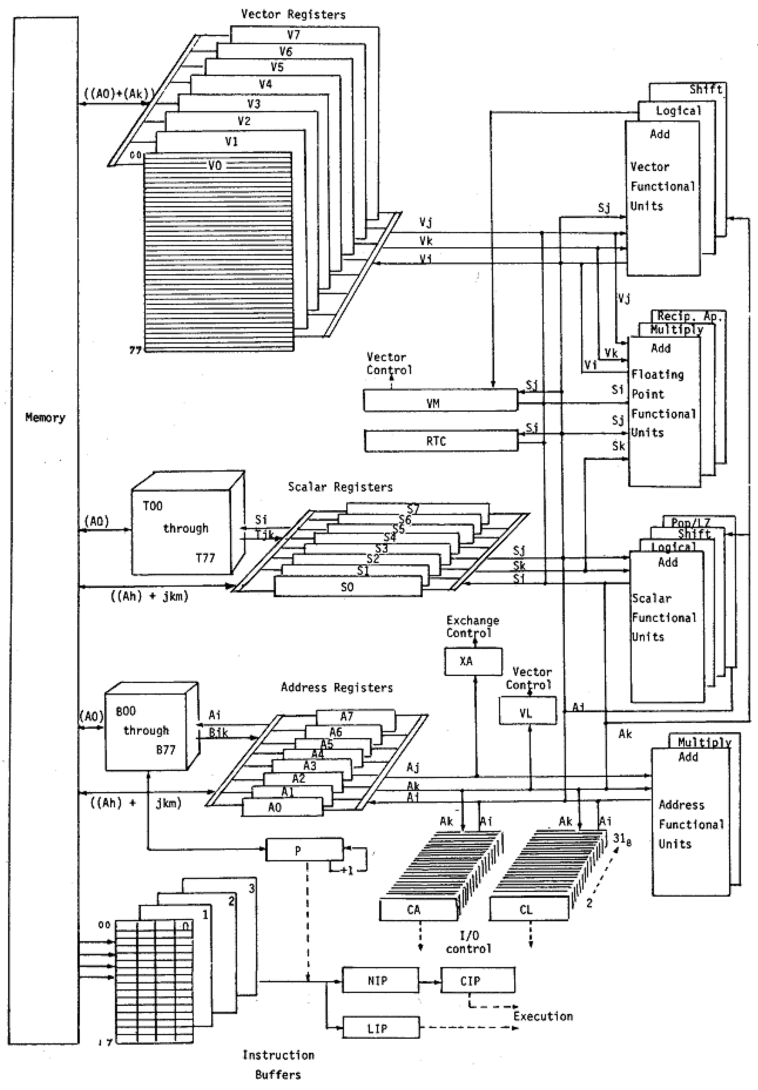
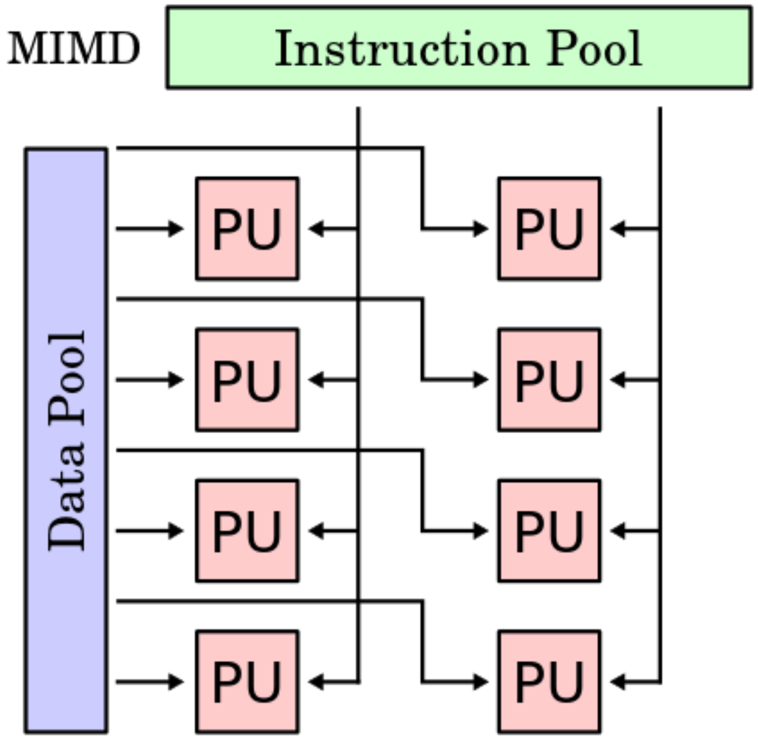


Figure 3-1. Computation section

## Multiple Instruction Multiple Data



CM-5 by Thinking Machines Corp. Composed of SPARC processors interconnected in a fat-tree network

## Address Space Organization

- Message-passing architecture
- Shared-address-space architecture

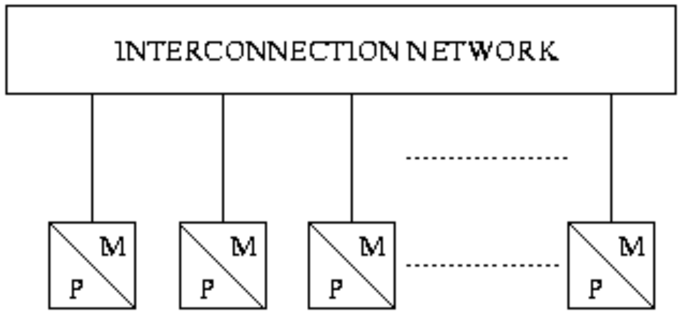


**Figure 2.4** A typical message-passing architecture.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

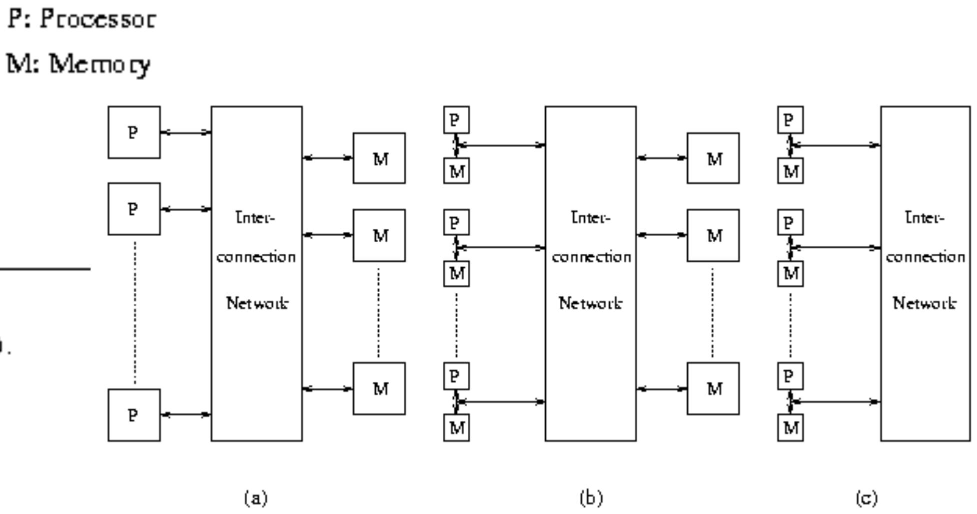


**Figure 2.5** Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Non-uniform-memory-access shared-address-space computer with local and global memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

## What is Trinity?

- **The Intel Xeon-Phi is a MIMD architecture**
  - Each Phi processing core is capable of operating on single or multiple data
    - Vector processing units (AVX-512) provide SIMD processing per core
  - Multiple cores can operate on different instructions and different memories concurrently (MIMD)
  - Within the Phi cores share an address space  (shared-address-space architecture)
- **Intel Xeon-Phi processors are interconnected in a flattened butterfly network**
  - Network and runtime systems provide either a shared-address-space view of these processors (Global Address Space runtimes) or a message-passing view (Message Passing Interface)
- **Control mechanisms: SISD, SIMD, and MIMD**
- **Address space organization: shared-address-space and message-passing**
- **So what programming model does this support?**
  - Many!

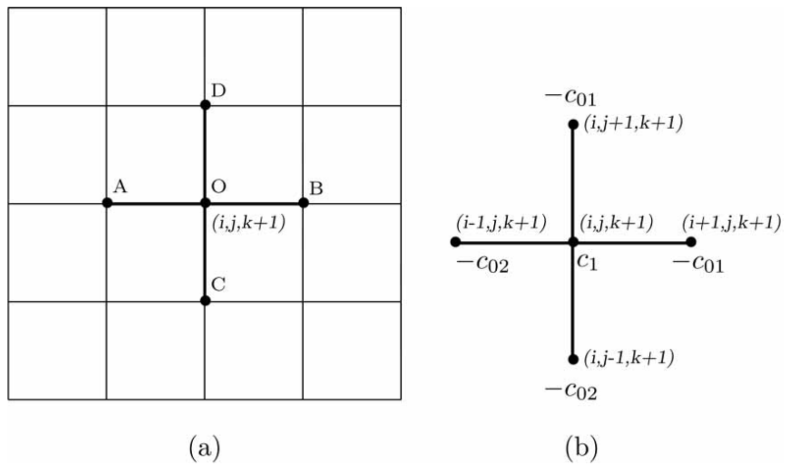
## Flynn's Taxonomy of computer architectures

| | Single Instruction stream | Multiple Instruction streams | Single Program | Multiple Programs |
|---|---|---|---|---|
| Single data stream | SISD | MISD | | |
| Multiple data streams | SIMD | MIMD | SPMD | MPMD |

- **Single instruction stream, single data stream (SISD)**
  - Traditional uniprocessors
- **Single instruction stream, multiple data streams (SIMD)**
  - Vector units, GPU stream processors
- **Multiple instruction streams, single data stream (MISD)**
- **Multiple instruction streams, multiple data streams (MIMD)**
  - Multi-core systems
- **Single program, multiple data streams (SPMD)**
  - Multiple cooperating processes running a single program
- **Multiple programs, multiple data streams (MPMD)**
  - Multiple cooperating processes running multiple programs

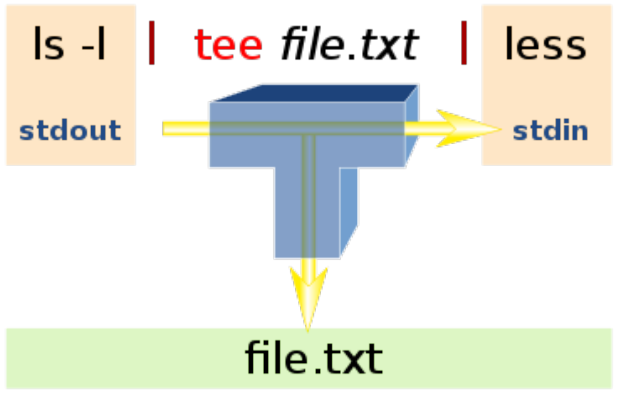
## Single Program Multiple Data

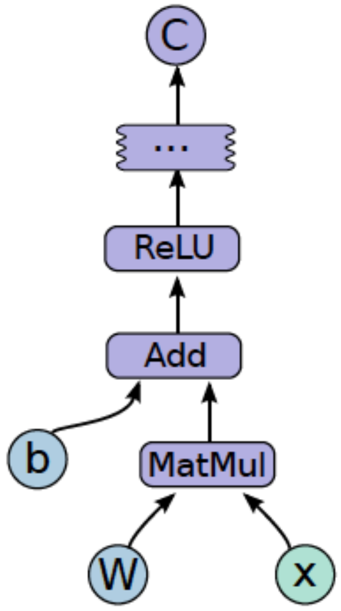
- **By far the most widely used programming model in high-performance computing**
- **Well-suited for message-passing architectures**
- **Each process runs the same program concurrently, exchanging data as needed to cooperatively complete a computation**
  - Often uses a stencil computation
  - Data that must be shared is often "ghosted" explicitly (copied) between cooperating processes
  - Finite-element, finite-volume and particle methods are often expressed as SPMD



## Multiple Program Multiple Data

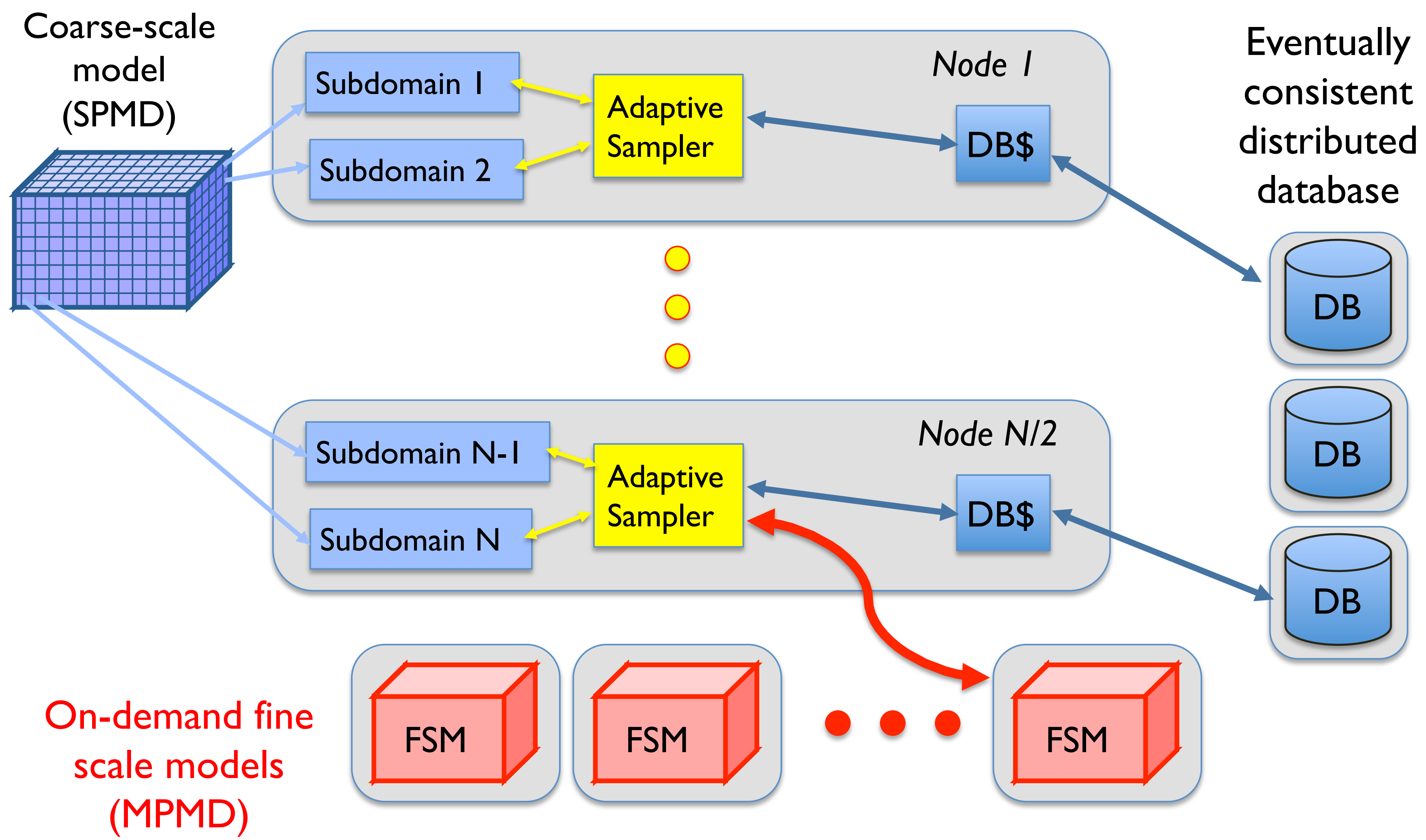
- **Less widely used in HPC, broadly used in “big data” applications**
- **Also well suited for message-passing architectures**
- **Each process runs a different program concurrently, exchanging data as needed to cooperatively complete a computation**
  - Often used in data-flow computations
  - Producer-consumer models of computation
    - Think pipes & filters in Unix





  - Some HPC applications utilize this model
    - Multi-scale applications such as ExMatEx’s Tabasco application

# ExMatEx Workflow Overview



## So what is a programming model?

## What is a programming model?

**Not Programming Models**

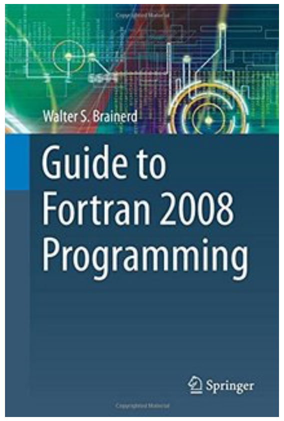
Walter S. Brainerd

Guide to Fortran 2008 Programming

Springer

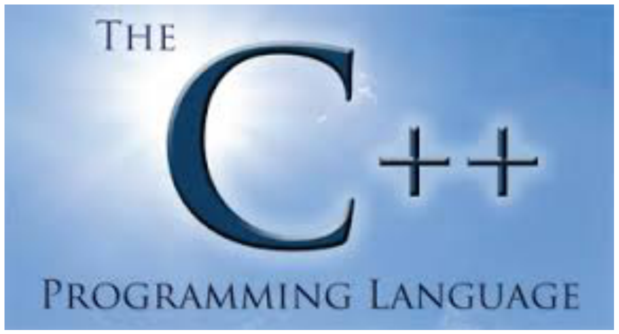
THE C++ PROGRAMMING LANGUAGE

## Programming languages


Walter S. Brainerd

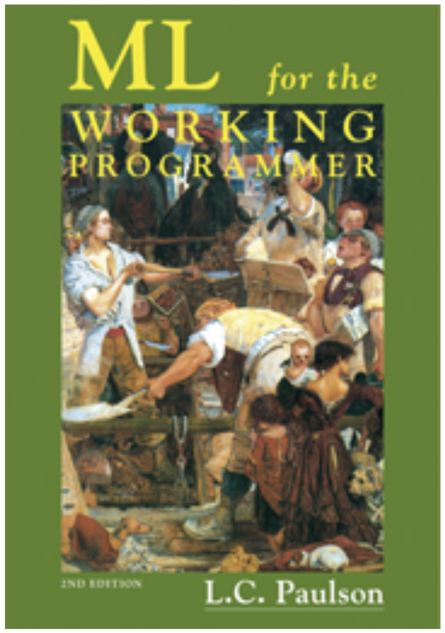
Guide to Fortran 2008 Programming

Springer


THE C++ PROGRAMMING LANGUAGE

THE C PROGRAMMING LANGUAGE

HASKELL


ML for the WORKING PROGRAMMER

2ND EDITION

L.C. Paulson

## Runtime Systems

MPI

OpenMP™

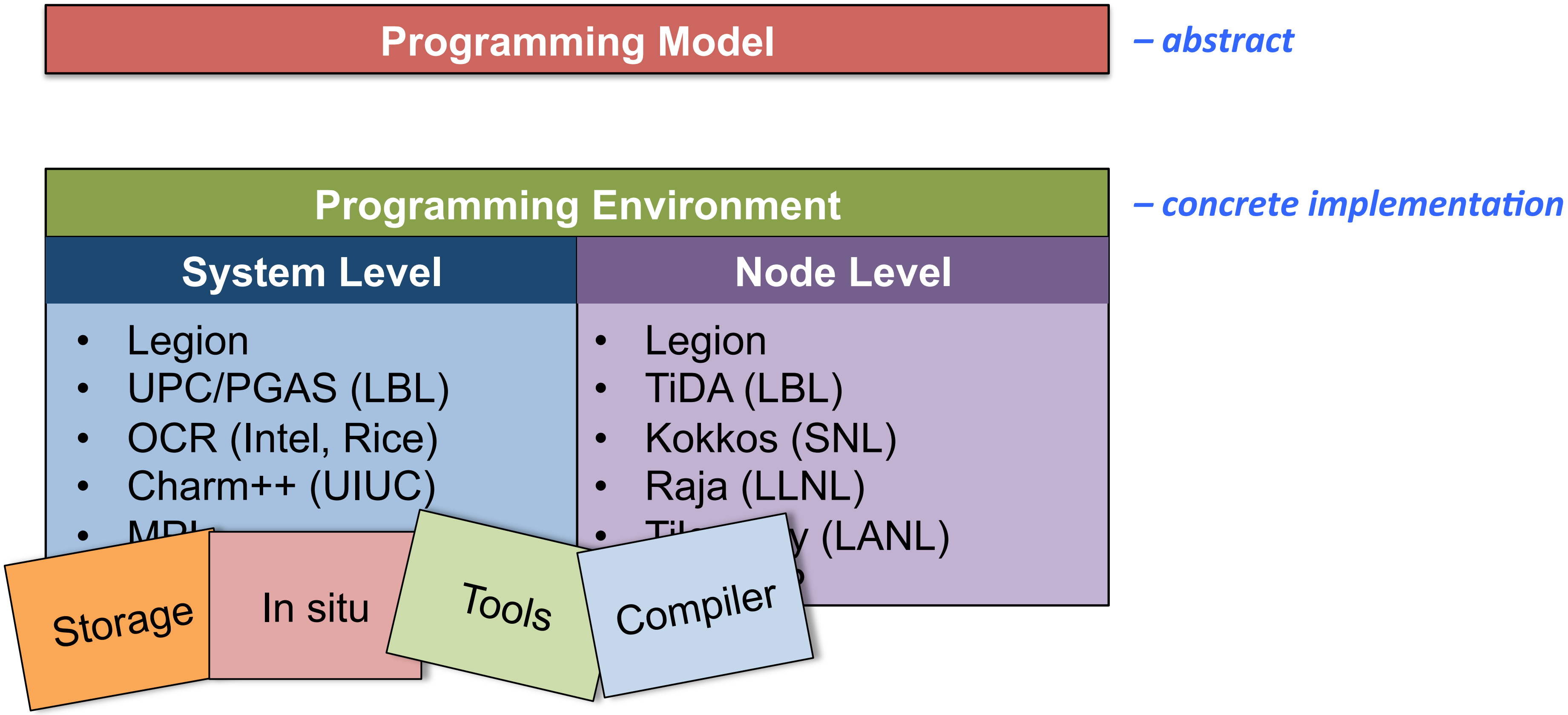
Charm++

OpenACC
Directives for Accelerators

**Legion**
*A Data-Centric Parallel Programming System*

PGAS

## Programming Model and Environments



*The programming model is an abstraction of the underlying computer system, allowing the expression of algorithms and data structures*

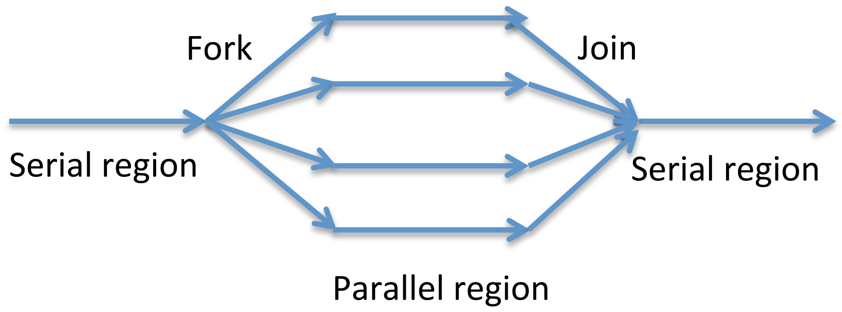
*- Languages and runtimes provide implementations of these abstractions*

*- A programming model exists independent of choice of language and runtime*

## MPI

- MPI (The Message Passing Interface)
  - Provides abstractions to implement either SIMD or MIMD using a message-passing architecture
- Each process in an MPI application is assigned a rank (0 – N) existing within a global MPI_COMM_WORLD
- Two-sided communication
  - MPI_SEND/MPI_RECEIVE
- One-sided communication
  - MPI_PUT/MPI_GET
- Collective communication
  - MPI_BCAST, MPI_GATHER, MPI_ALLTOALL
- Reductions
  - MPI_REDUCE
- Process granularity can be controlled by the user
  - Single process per node, single process per core
- Provides a well-documented API with C, C++, and FORTRAN bindings

```c
/******************************************************************************
* FILE: mpi_helloBsend.c
* DESCRIPTION:
*   MPI tutorial example code: Simple hello world program that uses blocking
*   send/receive routines.
* AUTHOR: Blaise Barney
* LAST REVISED: 06/08/15
******************************************************************************/
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define  MASTER		0

int main (int argc, char *argv[])
{
int  numtasks, taskid, len, partner, message;
char hostname[MPI_MAX_PROCESSOR_NAME];
MPI_Status status;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
/* need an even number of tasks  */
if (numtasks % 2 != 0) {
   if (taskid == MASTER)
      printf("Quitting. Need an even number of tasks: numtasks=%d\n", numtasks);
   }
else {
    if (taskid == MASTER)
      printf("MASTER: Number of MPI tasks is: %d\n",numtasks);
   MPI_Get_processor_name(hostname, &len);
   printf ("Hello from task %d on %s!\n", taskid, hostname);
   /* determine partner and then send/receive with partner */
   if (taskid < numtasks/2) {
     partner = numtasks/2 + taskid;
     MPI_Send(&taskid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD);
     MPI_Recv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &status);
     }
   else if (taskid >= numtasks/2) {
     partner = taskid - numtasks/2;
     MPI_Recv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &status);
     MPI_Send(&taskid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD);
     }
   /* print partner info and exit*/
   printf("Task %d is partner with %d\n",taskid,message);
   }
MPI_Finalize();
}
```



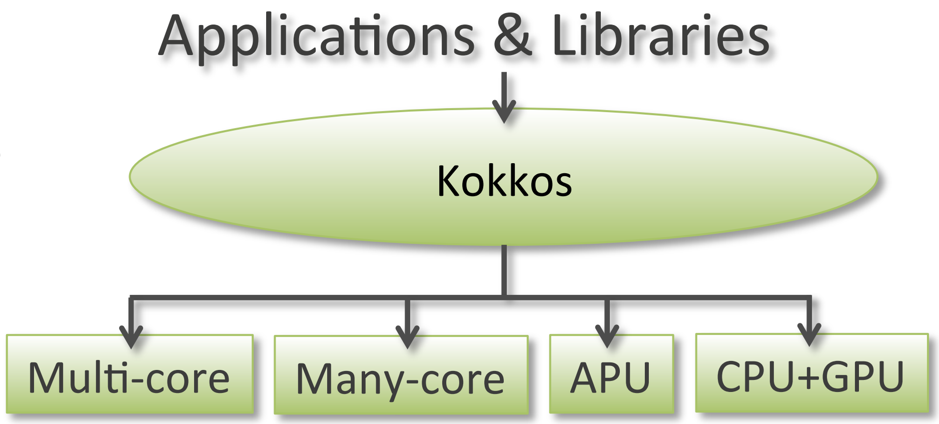
- Provides abstractions to implement either SIMD or MIMD using a shared-memory architecture
- Uses the fork-join model of parallelism
  - A master thread forks multiple parallel threads to execute a parallel region
- Provides constructs for parallel-for, tasks, and reductions
- Pragma based: OMP directives are inserted in the application code to describe parallel regions and reduction operations

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

int main()
{
   const int N = 10000000;
   const double L = 1.0;
   const double h = L / N;
   const double x_0 = 0.0;

   double pi;
   double t_1, t_2;

   int i;
   double sum = 0.0;

   t_1 = omp_get_wtime();
```

```
#pragma omp parallel for reduction(+: sum)
                                              schedule(static)
   for (i = 0; i < N; ++i)
   {
      double x = x_0 + i * h + h/2;
      sum += sqrt(1 - x*x);
   }

   t_2 = omp_get_wtime();

   pi = sum * h * 4.0;

   printf("omp_get_max_threads(): %d\n",

       omp_get_max_threads());
   printf("time: %f\n", t_2 - t_1);
   printf("pi ~ %f\n", pi);

   return 0;
}
```

## *Kokkos:* Performance Portable Thread-Parallel Programming Model

- **Open source, C++11-based library for node-level programming: application identifies parallelizable grains of *computations and data***



✓ **Multicore CPU** - including NUMA architectural concerns

✓ **Intel Xeon Phi (KNC)** – testbed prototype toward Trinity / ATS-1

✓ **NVIDIA GPU (Kepler)** – testbed prototype toward Sierra / ATS-2

✧ **IBM Power 8** – testbed prototype toward Sierra / ATS-2

✧ **AMD Fusion** – via collaboration with AMD

✓ Regularly and extensively tested
✧ Ramping up testing

https://github.com/kokkos/kokkos

## *Kokkos Abstractions: Patterns, Policies, and Spaces*

- **Parallel Pattern of user's computations**
  - parallel_for, parallel_reduce, parallel_scan, task-graph, ... (*extensible*)
- **Execution Policy tells how the computations will be executed**
  - Static scheduling, dynamic scheduling, thread-teams, ... (*extensible*)
- **Execution Space tells where the computations will execute**
  - Which cores, numa region, GPU, ... (*extensible*)
- **Memory Space tells where user data resides**
  - Host memory, GPU memory, high bandwidth memory, ... (*extensible*)
- **Layout (policy) tells how user data is laid out in memory**
  - Row-major, column-major, array-of-struct, struct-of-array … (*extensible*)

```
parallel_for( nrow, KOKKOS_LAMBDA( int i ){
  for ( int j = irow[i] ; j < irow[i+1] ; ++j )
    y[i] += A[j] * x[ jcol[j] ];
});
```

Lawrence Livermore National Laboratory

## RAJA: A Systematic Approach to Node-Level Portability and Tuning

- **Loops are the main conceptual abstraction in RAJA**
  - Based on loop structures and mesh traversal patterns in LLNL ASC codes (many loops O(10K) but only O(10) patterns – RAJA categorizes these patterns.
- **Lightweight, can be adopted incrementally, does not overburden maintenance, allows easy exploration of alternative parallel strategies**
- **Key abstractions:**
  - Traversals & execution policies (loop scheduling, execution, implementation details)
  - IndexSets (iteration space partition, data placement, dependency scheduling)
  - Reduction types (programming model portability)

More on RAJA: [http://1.usa.gov/1MJXlGd](http://1.usa.gov/1MJXlGd)

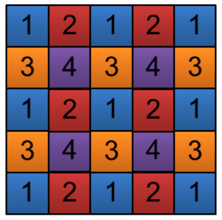
Lawrence Livermore National Laboratory

## IndexSets Allow Common Algorithms to Run Safely in Parallel Without Refactoring or Critical Sections

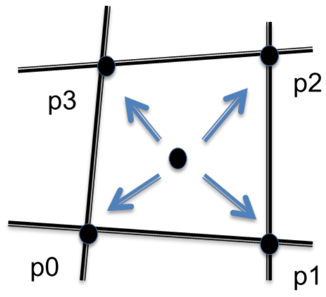
- **Allows loop traversals to execute groups of work in parallel that guarantee no race conditions in otherwise non thread-safe loops**
- **E.g. Element volume can be distributed to nodes without contention-heavy fine-grained synchronization such as critical sections, atomic operations, and temporary arrays**

```
forall<colorset>(elemSet, [=] (int elem) {
   int p0 = elemToNodeMap[elem][0];
   int p1 = elemToNodeMap[elem][1];
   int p2 = elemToNodeMap[elem][2];
   int p3 = elemToNodeMap[elem][3];
   double volFrac = elemVol[elem]/4.0 ;
   nodeVol[p0] += volFrac ;
   nodeVol[p1] += volFrac ;
   nodeVol[p2] += volFrac ;
   nodeVol[p3] += volFrac ;
} ) ;
```

Parallel reductions



|  |  |  |  |  |
|---|---|---|---|---|
| 1 | 2 | 1 | 2 | 1 |
| 3 | 4 | 3 | 4 | 3 |
| 1 | 2 | 1 | 2 | 1 |
| 3 | 4 | 3 | 4 | 3 |
| 1 | 2 | 1 | 2 | 1 |



- **Indexsets allow for ‘contention-light’ coarse-grained locking**

# Quick Overview of the Legion Programming Model

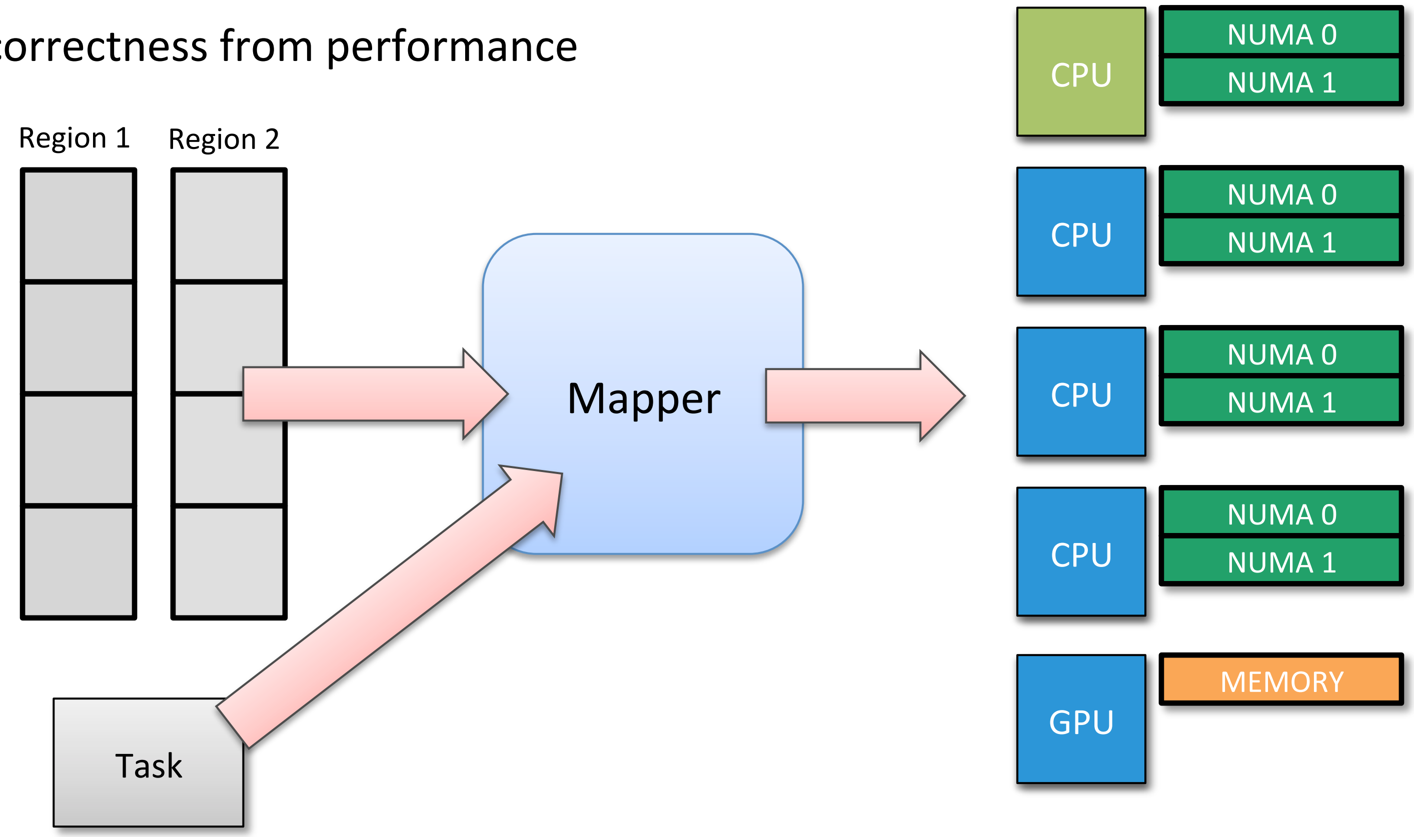
## Targets heterogeneous, distributed memory machines

- **Task:** unit of parallel execution
  - Task arguments are **regions** (collection of data w/ an index and field space)
  - Regions may be arbitrarily **partitioned** (by index space) and **sliced** by field (access)
- **Tasks must specify how they use their regions:**
  - **Privlieges**(read-only,write-only,read+write,reduce)
  - **Coherence**(exclusive,atomic,simultaneous) –w/ respect to “sibling” tasks
- **Tasks launches follow sequential semantics with relaxed execution order**

https://github.com/StanfordLegion/legion

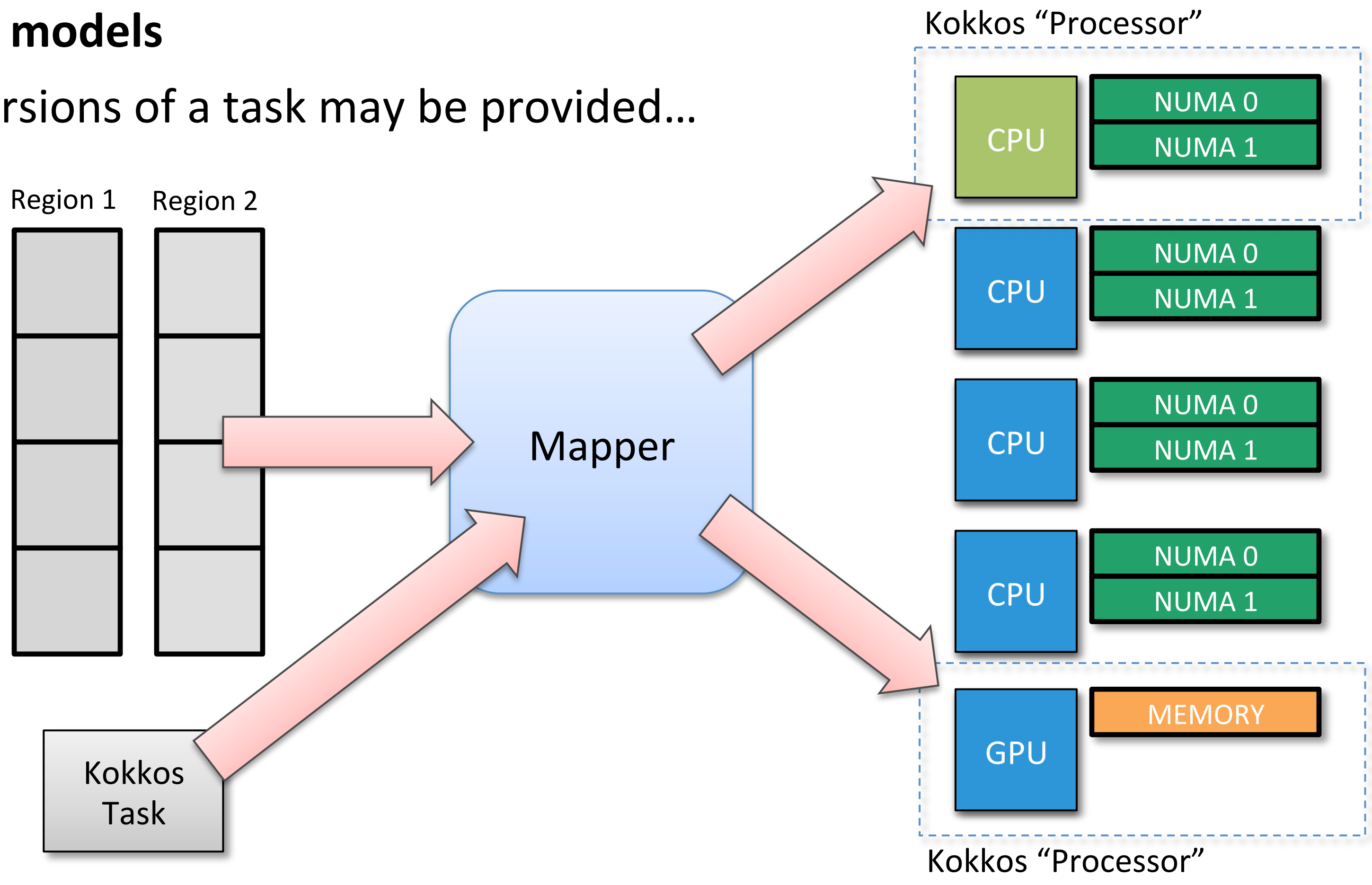
# Mapping Tasks and Data to Hardware Resources

- **Application selects:**
  - Where tasks run and where regions are placed
  - Computed dynamically
  - Decouple correctness from performance



## Mapping Tasks and Data to Hardware Resources
### Interoperability: Supporting Task-Level Models

- **Interoperability: Allow tasks to be written in different programming models**
  - Different versions of a task may be provided…

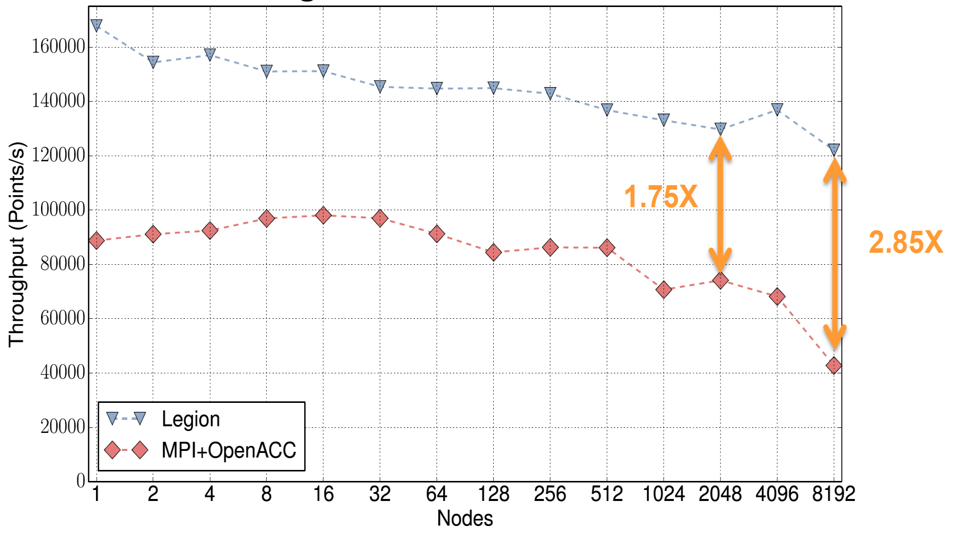


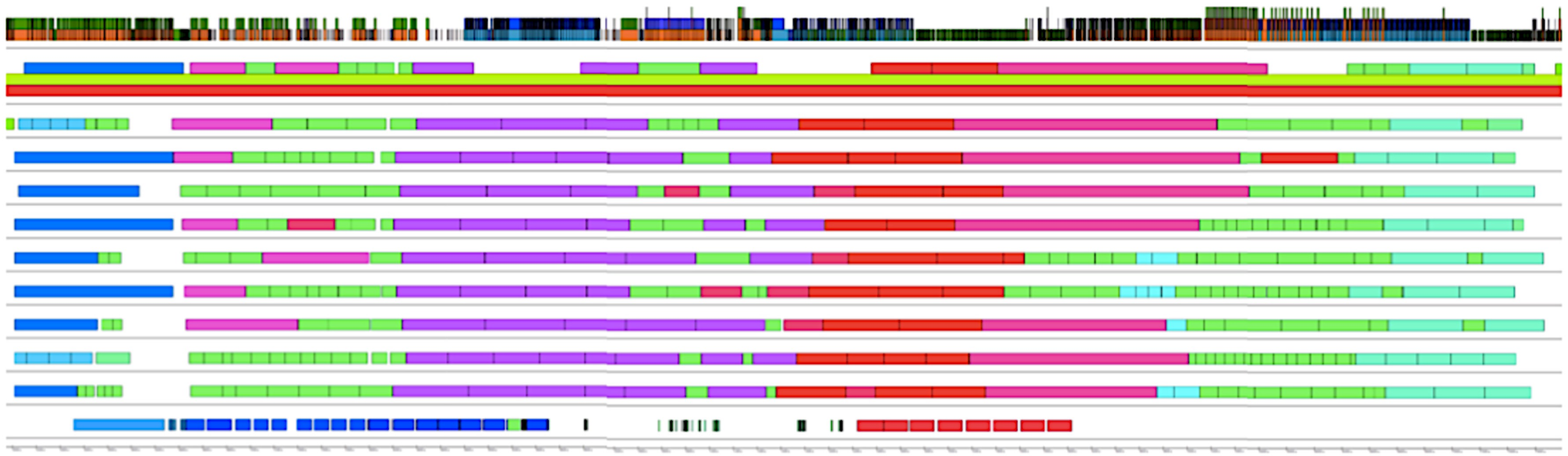
EXaCT CENTER FOR EXASCALE SIMULATION OF COMBUSTION IN TURBULENCE

## Legion S3D Execution and Performance Details

- **Mapping for $96^3$ Heptane**
  - Top line shows runtime workload
  - Different species required mapping changes (e.g., due to limited GPU memory size) – i.e. tuning is often not just app and system specific…

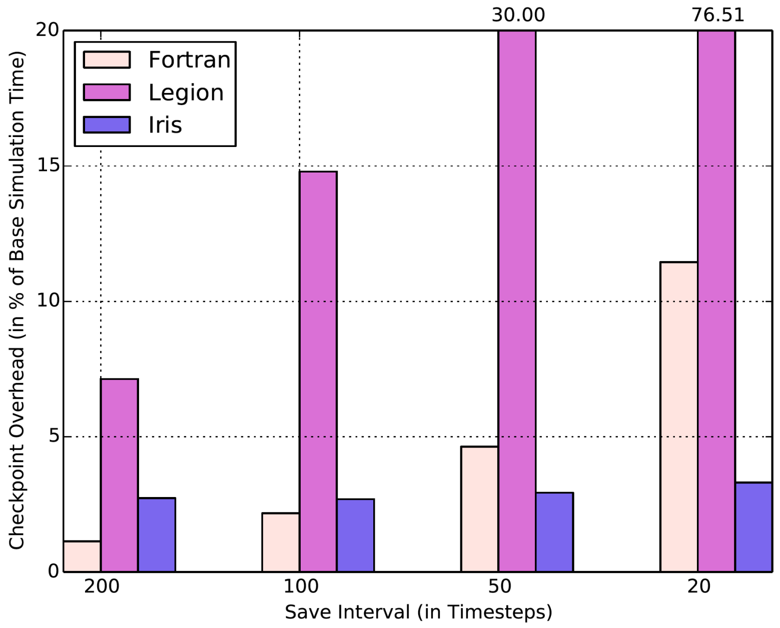
Weak scaling results on Titan out to 8K nodes





## Workflow: Integration of External Resources into the Programming Model

- **We can't ignore the full workflow!**
  - Amdahl's law sneaks in if we consider I/O from tasks – 15-76% overhead vs. 2-12% of original Fortran code!
- **Introduce new semantics for operating with external resources (e.g. storage, databases, etc.).**
  - Incorporates these resources into deferred execution model
  - Maintains consistency between different copies of the same data
  - Underlying parallel I/O handled by HDF5 but scheduled by runtime
- **Allow applications to adjust the snapshot interval based on available storage and system fault concerns instead of overheads.**



Performance of S3D checkpoints running on 64 nodes (i.e., 1,024 cores) of Titan.

```
#include <cstdio>
#include <cassert>
#include <cstdlib>
#include "legion.h"
using namespace LegionRuntime::HighLevel;

enum TaskIDs {
  TOP_LEVEL_TASK_ID,
  FIBONACCI_TASK_ID,
  SUM_TASK_ID,
};

void top_level_task(const Task *task,
                    const std::vector<PhysicalRegion> &regions,
                    Context ctx, HighLevelRuntime *runtime) {
  int num_fibonacci = 7; // Default value
  const InputArgs &command_args = HighLevelRuntime::get_input_args();
  if (command_args.argc > 1) {
    num_fibonacci = atoi(command_args.argv[1]);
    assert(num_fibonacci >= 0);
  }
  printf("Computing the first %d Fibonacci numbers...\n", num_fibonacci);

  std::vector<Future> fib_results;
  for (int i = 0; i < num_fibonacci; i++) {
    TaskLauncher launcher(FIBONACCI_TASK_ID, TaskArgument(&i,sizeof(i)));
    fib_results.push_back(runtime->execute_task(ctx, launcher));
  }

  for (int i = 0; i < num_fibonacci; i++) {
    int result = fib_results[i].get_result<int>();
    printf("Fibonacci(%d) = %d\n", i, result);
  }

  fib_results.clear();
}
```

```
int fibonacci_task(const Task *task,
                   const std::vector<PhysicalRegion> &regions,
                   Context ctx, HighLevelRuntime *runtime) {
  assert(task->arglen == sizeof(int));
  int fib_num = *(const int*)task->args;
  if (fib_num == 0)
    return 0;
  if (fib_num == 1)
    return 1;

  // Launch fib-1
  const int fib1 = fib_num-1;
  TaskLauncher t1(FIBONACCI_TASK_ID, TaskArgument(&fib1,sizeof(fib1)));
  Future f1 = runtime->execute_task(ctx, t1);

  // Launch fib-2
  const int fib2 = fib_num-2;
  TaskLauncher t2(FIBONACCI_TASK_ID, TaskArgument(&fib2,sizeof(fib2)));
  Future f2 = runtime->execute_task(ctx, t2);

  TaskLauncher sum(SUM_TASK_ID, TaskArgument(NULL, 0));
  sum.add_future(f1);
  sum.add_future(f2);
  Future result = runtime->execute_task(ctx, sum);

  return result.get_result<int>();
}
```

```
int sum_task(const Task *task,
             const std::vector<PhysicalRegion> &regions,
             Context ctx, HighLevelRuntime *runtime) {
  assert(task->futures.size() == 2);
  Future f1 = task->futures[0];
  int r1 = f1.get_result<int>();
  Future f2 = task->futures[1];
  int r2 = f2.get_result<int>();

  return (r1 + r2);
}

int main(int argc, char **argv) {
  HighLevelRuntime::set_top_level_task_id(TOP_LEVEL_TASK_ID);
  HighLevelRuntime::register_legion_task<top_level_task>(TOP_LEVEL_TASK_ID,
      Processor::LOC_PROC, true/*single*/, false/*index*/);
  HighLevelRuntime::register_legion_task<int,fibonacci_task>(FIBONACCI_TASK_ID,
      Processor::LOC_PROC, true/*single*/, false/*index*/);
  HighLevelRuntime::register_legion_task<int,sum_task>(SUM_TASK_ID,
      Processor::LOC_PROC, true/*single*/, false/*index*/,
      AUTO_GENERATE_ID, TaskConfigOptions(true/*leaf*/), "sum_task");

  return HighLevelRuntime::start(argc, argv);
}
```

## Resources

- MPI: https://computing.llnl.gov/tutorials/mpi/
- OpenMP: https://computing.llnl.gov/tutorials/openMP/
- OpenACC: http://www.openacc.org
- Kokkos: https://github.com/kokkos/kokkos-tutorials
- Raja: http://software.llnl.gov/RAJA/
- Legion: http://legion.stanford.edu
- Charm++: http://charm.cs.illinois.edu/research/charm
- PGAS: http://www.pgas.org