CIS 631
Parallel Computing
Task-based Parallel Programming

Department of Computer and Information Science
Spring 2018

UNIVERSITY OF OREGON
Logistics

- How did the programming meetup go?
  - Keep them going

- Lecture include discussion from this paper:
  
Parallel Programming Term Project

- Major programming project for the course
  - Non-trivial parallel application
  - Apply parallel programming and include performance analysis

- Project teams
  - 2-5 person teams
  - 2-3 teams

- Project dates
  - Proposal on Monday, May 7, 18:00pm, or earlier
  - Interim status meeting on Monday, May 21, with each team
  - Presentation on Thursday, 8:00, June 14 (finals period)
  - Report due on Thursday, 8:00, June 14 (finals period)

- Need to get system accounts!!!
  - NIC: https://systems.nic.uoregon.edu/account
    - indicate that this is for CIS 631 in Spring 2018
  - Talapas: should have been contacted by system administration
Term Paper

- Investigate parallel computing topic of interest
  - More in depth review
  - Individual choice
  - Summary of major points

- Requires minimum of ten references
  - Book and other references has a large bibliography
  - Google Scholar, Keywords: parallel computing
  - NEC CiteSeer Scientific Literature Digital Library

- Paper topic Friday, May 4, 18:00pm
  - Abstract and 8 research references
  - Final term paper due Friday, June 8 at 18:00
Better Programming Models

- Higher-level programming models can help insulate algorithms from parallel implementation details
  - Yet, without necessarily abdicating control

- Build in productive programming abstractions ...
  - Concurrency constructs
  - Memory sharing constructs
  - Synchronization mechanism

- ... while providing power execution models
  - Task parallel
  - Data parallel

- Target evolving parallel computing systems
Big Picture of Parallel Programming

Decomposition
- Task Decomposition
- Data Decomposition

Dependence Analysis
- Group Tasks
- Order Tasks
- Data Sharing

Design Evaluation
Types of Parallel Programs

- Flavors of parallelism
  - Data parallelism
    - all processors do same thing on different data
  - Task parallelism
    - processors are assigned tasks that do different things

- Parallel computation models
  - Data parallel
  - Producer-Consumer
  - Task graph
  - Work pool
  - Master-Worker
Granularity

- Granularity can be with respect to tasks and data
- Task granularity
  - Equivalent to choosing the number of tasks
  - Fine-grained decomposition results in large number of tasks
  - Large-grained decomposition has smaller number of tasks
  - Translates to data granularity after number of tasks chosen
    - consider matrix multiplication
- Data granularity
  - Think of in terms of amount of data needed in operation
  - Relative to data as a whole
  - Decomposition decisions based on input, output, input-output, or intermediate data
Task Parallelism and Programming

- Task-based parallel is a methodology for parallel execution.
- Efficient techniques are known and have theoretical foundation:
  - Graph-based expression of parallel algorithm
  - Include capturing of inputs/outputs and dependencies
- Task-based parallelism can be both a programming model with an API and a runtime system.
- Focus is on how to define tasks and their interaction.
- How to implement runtime support for execution.
Task-based Programming

- A different approach to writing your code
- Think of your program as a tree of tasks
  - Each task will depend on other tasks
  - Can have child tasks
  - Not the same on all nodes
- Want to have a large # tasks to generate high levels of concurrency
  - Fill in execution gaps as small as possible
  - Keep breaking tasks into smaller tasks
  - Scheduler can fill gaps
- Limit of task size/granularity is a function of overheads
- How do you handle the overheads?
  - Time to create, context switch queue, dequeue a task
Task Decomposition

- Breaking a program into tasks should be straightforward
- More functional
  - Tasks should accept inputs and return results
  - Modifying global state should be avoided
  - Race conditions and other thread related problems
- Leaf nodes are the smallest bits of work
- How to program for tasking?
- How to support API in runtime system?
**Task-base Parallel Programming**

- Cilk
- Chapel
- X10
- OpenMP
- OmpSs
- Charm++
- HPX
- Legion
- ParSEC
Understanding Task-based Environments

- Task-based parallel programming models for shared memory and distributed memory
  - Include parallel, manycore, heterogeneous, …
  - Many targeting high-performance computing (HPC)

- Taxonomy of languages, programming methodologies and interfaces, and runtime systems

- Use in classifying state-of-the-art environments

- Task-based parallel programming paradigm
  - In contrast with loop-based and message passing
Task-based Parallel Programming

- **Definition**
  - A *task* is a sequence of instructions within a program that can be processed concurrently with other tasks in the same program.
  - Execution of tasks constrained by control- and data-flow dependencies.

- **Task-based parallel execution is supported in programming languages**
  - C++11 thread support library
  - Cilk language with efficient task scheduling via work stealing
  - OpenMP via language extension and runtime (since v3.0)

- **Parallel libraries also support task-based parallelism**
  - Intel Cilk Plus or Intel TBB
  - HPX

- **Runtime systems can be based on task parallelism to improve shared memory performance of existing language extensions**
  - Qthreads, Argobots, …
  - Important for manycore processors and lightweight runtimes

- **Also supports accelerators**
  - StarPU
Distributed Task-based Parallelism

- Task-based parallelism on distributed memory systems
  - Important target for HPC
- Tasking is combined with GAS …
  - GASPI
- … and scheduling across distributed processes
  - Distributed execution of a single task-parallel program
- Languages: Chapel and X10
- Libraries: HPX, Charm++
  - With asynchronous GAS runtimes
- Runtimes with additional parallel concepts
  - Legion with data-centric parallel programming
Motivation for a Taxonomy

- Understanding with respect to omnipresent MPI programming model
  - MPI+X
  - Get a more concise picture of alternatives

- Consider that each task-based environment has two central components:
  - a *programming interface* (API)
  - *runtime system* encompassing underlying mechanisms
Task-based Parallel Programming Interfaces

- Defines way an application developer describes:
  - Parallelism
  - Dependencies
  - Work mapping / data distribution

- Define characterizing features of APIs

- Four broad categories
  - Architectural characteristics targeted
  - Task system offered
  - Work management
  - Engineering aspects
Architectural

- Communication model
  - *Shared memory* (smem)
  - *Message passing* (msg)
  - *Global address space* (gas)

- Distributed memory
  - *No* support
  - *Explicit* support (e.g., message passing)
  - *Implicit* support (e.g., automatic data migration)

- Heterogeneity
  - Execution on accelerators
  - Explicit support means the developers provisions the tasks on the accelerator using a distinct API
Task System

- Graph structure
  - Type of task graph dependency structure supported by API
  - Possibilities: tree, acyclic graph (dag), or an arbitrary graph

- Task partitioning
  - Whether each task is atomic (only be scheduled as a single unit)
  - Can be subdivided/split

- Result handling
  - Explicit handling of the results of task computations
  - Return types accessed as futures

- Task cancellation
  - No cancellation support
  - Supported cooperatively (only at task scheduling points)
  - Supported preemptively
Management

- Worker management
  - Threads/processes are started/maintained by user (*explicit*)
  - Provided automatically by the environment (*implicit*)

- Resilience management
  - Such as fine-grained checkpointing and restart

- Work mapping
  - Mapped to the existing hardware resources
  - Possibilities: *explicit, implicit, pattern-based*

- Synchronization
  - By regions or function scope (*implicit*)
  - By application developer (*explicit*)
Engineering

- Technology readiness
  - Matureness of implementation
  - Technology readiness according to standards

- Implementation type
  - How the API is implemented
  - How program addresses API
  - Is it a library?
  - Is it a language extension?
  - Is it an entire language with task integration
Taxonomy of Task-parallel APIs

Target Architecture
- Shared memory
- Distributed memory

Data Distribution
- Explicit
- Implicit

Scheduling
- Targets
  - time
  - energy
  - multiple
- Methods
  - static
  - dynamic
  - hybrid

Performance Monitoring
- Online
  - analysis
  - introsp.
  - hist. data
- Offline
  - analysis

Fault Tolerance
- None
- Task faults
- Process faults

Many-task runtimes may require performance introspection and monitoring (Sect. 3.2) to facilitate the implementation of different scheduling policies, that is, using online performance information to assist the decision making process of the scheduler.

Fault tolerance is another key factor that is important in many-task runtime systems in the context of Exascale requirements. As detailed in Sect. 3.3, a runtime may have no resilience capabilities, or it may target task faults or even process faults.

3.1 Scheduling in many-task runtime systems

3.1.1 Scheduling targets
Depending on the capabilities of the underlying many-task runtime system, its scheduling domain is usually limited to a single shared memory homogeneous compute node, after heterogeneous distributed memory systems of interconnected compute nodes, or in a most generic form to heterogeneous distributed memory systems. By supporting different types of heterogeneous architectures, the runtime can facilitate source code portability and support transparent interaction between different types of computation units for application developers.

Traditionally, execution time has been the main objective to minimize for different scheduling policies. However, the increasing scale of HPC systems makes it necessary to take the energy and power budgeting of the target system into account as well. Therefore, some many-task runtime systems have already started providing energy-aware scheduling policies. In a simple scenario, it is assumed that the application can provide an energy consumption model which can be used by a scheduling policy as part of its objective function. In more advanced cases, the runtime provides offline or online profiling data. These data are used to build a look-up table that maps each
Many-task Runtime System

- Serve as a basis for implementing APIs
- A major difference is the target architecture
  - Evolution from *homogeneous shared memory* ...
  - ... to *heterogeneous shared memory* ...
  - ... to *distributed memory* platforms
- Support for distributed memory varies significantly
- Issues of how data distribution is done
  - *Implicitly*: handled by runtime
  - *Explicitly*: specified by the programmer
Task Scheduling

- Different scheduling methods
  - Utilize information such as execution time per tasks, dependencies, resource usage, task communication, synchronization

- Categories
  - Static: distribute a global task list to different compute units
  - Dynamic: scheduling at runtime for load balancing
  - Hybrid: combination of static and dynamic

- Automatic scheduling
  - Within a shared memory machine only
    - Application developer responsible for data distribution
  - Uniform scheduling policies across nodes

- May require performance introspection and monitoring
- Fault tolerance also comes into play
Dynamic Task Scheduling

- Used if the information is not available before execution or it is too complex
- Goal is to achieve good load balance
- Tradeoff overhead for dynamic adaptability (e.g., load balancing)
- Work stealing:
  - Most widely used load balancing technique
  - Distribute tasks in per-processor work queues, where each processor operates on its local queue
  - Processors can steal tasks from other queues to perform load balancing
- Work sharing:
  - Schedules each task onto a processor when it is spawned
  - Usually implemented by a centralized task pool
  - Whenever a worker spawns a new task, the scheduler migrates it to a new worker to improve load balancing
Taxonomy of Many-task Runtime Systems

Task-parallel API Characteristics

Architectural
- Communication Model
  - smem
  - msg
  - gas
- Distributed Memory
  - exp.
  - imp.
- Heterogeneity
  - exp.
  - imp.
  - none

Task System
- Graph Structure
  - tree
  - dag
  - graph
- Task Partitioning
  - atom.
  - yes
- Result Handling
  - exp.
  - imp.
- Task Cancellation
  - coop.
  - pre.

Management
- Worker Management
  - exp.
  - imp.
- Resilience
  - avail.
  - none
- Work Mapping
  - exp.
  - imp.
  - patt.
- Synchronization
  - exp.
  - imp.

Engineering
- Technology Readiness
  - TRL 1
  - ...
  - TRL 9
- Implementation Type
  - lang.
  - ext.
  - lib.
Classification: Task-based APIs

- If an API supports a given feature, it must not require the user to resort to 3rd-party libraries or implementation-specific details

- Clarifications:
  - In C++ STL, we consider the entity launched by `std::async` to represent a task
  - HPX is an implementation of the C++ tasking API providing additional features such as distributed memory support and task dependencies
  - While StarPU offers shared memory parallelism, it is capable of generating MPI communication from a given task graph and data distribution
  - Thus, StarPU is marked with explicit support for distributed memory using a message-based communication model
  - PaRSEC includes both a task-based runtime that works on user-specified task graph and data distribution information, as well as a compiler that accepts serial input and generates this data
# Feature Comparison of Task-parallel APIs

<table>
<thead>
<tr>
<th>Architectural</th>
<th>Task System</th>
<th>Management</th>
<th>Eng.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication Model</td>
<td>Distributed Memory</td>
<td>Heterogeneity</td>
<td>Graph Structure</td>
</tr>
<tr>
<td>C++ STL</td>
<td>smem</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>TBB</td>
<td>smem</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>HPX</td>
<td>gas</td>
<td>i</td>
<td>e</td>
</tr>
<tr>
<td>Legion</td>
<td>gas</td>
<td>i</td>
<td>e</td>
</tr>
<tr>
<td>PaRSEC</td>
<td>msg</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>OpenMP</td>
<td>smem</td>
<td>×</td>
<td>i</td>
</tr>
<tr>
<td>Charm++</td>
<td>gas</td>
<td>i</td>
<td>e</td>
</tr>
<tr>
<td>OmpSs</td>
<td>smem</td>
<td>×</td>
<td>i</td>
</tr>
<tr>
<td>AllScale</td>
<td>gas</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>StarPU</td>
<td>msg</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>Cilk Plus</td>
<td>smem</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Chapel</td>
<td>gas</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>X10</td>
<td>gas</td>
<td>i</td>
<td>i</td>
</tr>
</tbody>
</table>

Following the API taxonomy defined in Sect. 2, Table 1 classifies existing task-parallel APIs. Note that for an API to qualify as supporting a given feature, this API must not require the user to resort to third-party libraries or implementation-specific details of the API. For instance, some APIs offer arbitrary task graphs via manual task reference counting [12]. This does not qualify as support in our classification. Also note that all APIs marked as featuring task cancellation do so in a non-preemptive manner due to the absence of OS-level preemption capabilities.

Some entries require additional clarification. In C++ STL, we consider the entity launched by `std::async` to represent a task. At the same time, HPX is an implementation of the C++ tasking API providing additional features such as distributed memory support and task dependencies. Also, while StarPU offers shared memory parallelism, it is capable of generating MPI communication from a given task graph and data distribution [2]; hence, it is marked with explicit support for distributed memory using a message-based communication model. Furthermore, PaRSEC includes both a task-based runtime system and a user-specified task graph and data distribution information, as well as a compiler that accepts serial input and generates this data. As the latter is limited to loops, we only consider the runtime in this work.

Several observations can be made from the data presented in Table 1. First, all APIs targeting distributed memory also support heterogeneity in some form. APIs offering implicit distributed memory support generally employ a global address space and implement task partitioning. Second, among APIs lacking distributed memory, only OmpSs offers resilience (via its Nanos++ runtime), and distributed memory APIs only recently started to include resilience support [9]—likely driven by the continuous increase in machine sizes and, hence, decreased mean time between failures. Finally, all APIs listed above support task cancellation in some form.
Observations

- APIs offering implicit distributed memory support generally employ a global address space and task partitioning.
- Some form of heterogeneity is provided in almost all modern APIs.
  - Often requires explicit heterogeneous task provisioning by the user.
- Among APIs lacking distributed memory, only OmpSs offers resilience.
- Distributed memory APIs only recently started to support resiliency.
# Comparison of Task Parallel Runtimes

<table>
<thead>
<tr>
<th>Target Architecture</th>
<th>Data Distribution</th>
<th>Scheduling Methods (sm)</th>
<th>Scheduling Methods (d)</th>
<th>Performance Monitoring</th>
<th>Fault Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP runtimes(*)</td>
<td>sm</td>
<td>×</td>
<td>m</td>
<td>×</td>
<td>off/on</td>
</tr>
<tr>
<td>Intel TBB</td>
<td>sm</td>
<td>×</td>
<td>ws</td>
<td>×</td>
<td>off</td>
</tr>
<tr>
<td>Intel Cilk Plus</td>
<td>sm</td>
<td>×</td>
<td>ws</td>
<td>×</td>
<td>off</td>
</tr>
<tr>
<td>StarPU</td>
<td>sm</td>
<td>e</td>
<td>m</td>
<td>×</td>
<td>off/on</td>
</tr>
<tr>
<td>Nanos++</td>
<td>d</td>
<td>i</td>
<td>m</td>
<td>×</td>
<td>off/on</td>
</tr>
<tr>
<td>Charm++</td>
<td>d</td>
<td>i</td>
<td>m</td>
<td>m</td>
<td>off/on</td>
</tr>
<tr>
<td>X10</td>
<td>d</td>
<td>i</td>
<td>ws</td>
<td>×</td>
<td>off</td>
</tr>
<tr>
<td>Chapel</td>
<td>d</td>
<td>i</td>
<td>ws</td>
<td>×</td>
<td>off</td>
</tr>
<tr>
<td>HPX</td>
<td>d</td>
<td>i</td>
<td>m</td>
<td>×</td>
<td>off/on</td>
</tr>
<tr>
<td>AllScale</td>
<td>d</td>
<td>i</td>
<td>ws</td>
<td>l</td>
<td>off/on</td>
</tr>
<tr>
<td>ParSEC</td>
<td>d</td>
<td>e</td>
<td>m</td>
<td>l</td>
<td>off</td>
</tr>
<tr>
<td>Legion</td>
<td>d</td>
<td>i</td>
<td>ws</td>
<td>ws</td>
<td>off/on</td>
</tr>
</tbody>
</table>

(*) Such as Intel OpenMP, GOMP, Qthreads, and Argobots
Classification: Task-based RTS

- An API always translates into at least one runtime implementation, but is not always limited to one such implementation
  - For the majority of cases, except for the OpenMP API and the Cilk API, there is a 1:1 mapping between API and runtime
  - Most diverse example is offered by the OpenMP API, which has many runtime implementations
- Other runtimes with shared memory and distributed memory support (e.g. HPX) can also run as backends to OpenMP
- Nanos++ goes beyond OpenMP in its support for pragmas and for distributed execution
Classification: Task-based RTS (2)

- Work stealing is the most common method of scheduling
- No established method for inter-node scheduling
  - ParSEC only provides a limited inter-node scheduling based on remote completion notifications
  - Legion uses distributed work stealing
- Recent work in X10 extends the X10 scheduler with distributed work stealing algorithms across nodes
- Same applies to StarPU and OmpSs
  - New distributed memory scheduling policies are being developed for these runtimes, but they are not part of their main release yet
- Chapel, X10, and HPX, has automatic data distribution support (runtime feature)
  - However, these runtimes require explicit work mapping in distributed memory environments (API feature)
X10: Productivity and Performance at Scale

- >8 years of R&D by IBM Research supported by DARPA
- Bring Java-like productivity to HPC
  - Evolution of Java with input from Scala, ZPL, CCP, …
  - Imperative OO language, garbage collected, type/memory safe
  - Rich data types and type system
- Designed for scale
  - Multi-core, multi-processor, distributed, heterogeneous systems
  - Few simple constructs for concurrency and distribution
- Tool chain
  - Open source compilers, runtime, IDE
  - Debugger (not open source)
- X10 implementations
  - C++ based (“Native X10”)
  - JVM based (“Managed X10”)

http://x10-lang.org
X10 Compilation and Execution

X10 Compiler Front-End

- X10 Source
  - Parsing / Type Check
  - X10 AST
  - AST Optimizations AST Lowering

Managed X10

- Java Interop
  - Java Back-End
    - Java Code Generation
    - Java Source
    - XRJ
    - Java Compiler
    - Java Bytecode
    - JNIEnv
    - Java VMs
    - X10RT

Native X10

- C++ Back-End
  - C++ Code Generation
  - C++ Source
  - Cuda Source
  - Platform Compilers
  - XRX
  - Native executable
  - Existing Native (C/C++/etc) Application
  - XRC
  - Native Environment (CPU, GPU, etc)
**X10 Runtime**

- **X10RT (X10 runtime transport)**
  - Active messages, collectives, RDMAs
  - Implemented in C; emulation layer

- **Native runtime**
  - Processes, threads, atomic operations
  - Object model (layout, rtt, serialization)
  - Two versions: C++ and Java

- **XRX (X10 runtime in X10)**
  - Implements APGAS: async, finish, at
  - X10 code compiled to C++ or Java

- **Core X10 libraries**
  - x10.array, io, util, util.concurrent
Chapel

- Chapel is a modern programming language
  - First-class concepts for concurrency and parallelism
  - Designed with programmability and performance
  - Portable, scalable, and open source

Philosophy: Good, *top-down* language design can tease system-specific implementation details away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.

https://chapel-lang.org
Chapel’s Multiresolution Design

- Higher levels for programmability, productivity
- Lower levels for greater degrees of control
- Build higher-level concepts in terms of the lower
- Permit the user to intermix layers arbitrarily

Chapel language concepts
Chapel Concepts

- Task parallelism
  - Coforall
  - Cobegin

- Parallelism: Data-Driven Synchronization
  - Atomic variables: support atomic operations
  - Sync variables: store full-empty state along with value
  - Sync statements: join unstructured tasks
  - Serial statements: conditionally squash parallelism
Chapel Locale Type

Definition:
- Abstract unit of target architecture
- Supports reasoning about locality
- Defines “here vs. there” / “local vs. remote”
- Capable of running tasks and storing variables

Typically a compute node
- Has processors and memory
- Multicore processor or SMP

Communication is implicit
Data Parallel Features

- Rich domain/array types
  - Multidimensional: strided, sparse, associative

- Slicing
  - Refer to subarrays using ranges/domains

- Promotion
  - Call scalar functions with array arguments

- Reductions/Scans
  - Apply operations across collections
Charm++

- Machine independent parallel programming system
- Charm++ programs are written in C++
  - A few library calls and a interface description language
  - Multiple inheritance, late bindings, and polymorphism
- Design of the system is based on the following tenets:
  - Efficient portability: runs on shared and distributed memory
  - Latency tolerance: hides communication via efficient message-driven execution
  - Dynamic load balancing: dynamic creation and migration of work via load balancing monitoring and algorithms
  - Reuse and modularity: "module" construct and mechanisms for compositionality of modules without sacrificing the latency-tolerance

http://charm.cs.illinois.edu/research/charm
**Charm++ Programming Model**

- Charm++ is a concurrent object-oriented system
- Programs consist of potentially medium-grained processes (called *charses*)
  - Special type of replicated process
  - Processes interact with each other via messages
- Programs have collections of charses (few to many)
  - Can also implement distributed data structures
- Objects are mapped by runtime system to processors

![Diagram](image-url)
HPX

- HPX is a tasking runtime system
  - Implemented as a C++ library/framework
  - Provides threads/futures/locks/schedulers/…

- User programs the following:
  - Creates tasks
  - Synchronizes between/amongst them

- HPX differs from (most) other libraries because the same API and the same scheduling/runtime can be used for the whole hierarchy of tasks
  - Single framework based soundly on C++
  - Aim to replace OpenMP+MPI+accelerator
  - From top to bottom (of the task tree)

http://stellar-group.org/libraries/hpx/
https://github.com/STEllAR-GROUP/tutorials/tree/master/cscs2017
Legion

- Focus on high-end goals
  - High performance
  - Performance portability
  - Programmability

- Time step of mini-app task graph
  - Who generates it?
  - Who will schedule it?
  - Who will re-schedule it?

- Today it is the programmer’s responsibility
- It should be the programming system’s responsibility
- Legion is a project from Stanford and LANL

http://legion.stanford.edu
Legion: Tasks and Regions

- A task is the unit of parallel execution
  - A function

- Task arguments are regions
  - Collections
  - Rows are an index space
  - Columns are fields

- Tasks declare how they use their regions

```plaintext
task saxpy(is : ispace(int1d), x,y: region(is, float), a: float )
where reads(x, y), writes(y)
```

<table>
<thead>
<tr>
<th>0</th>
<th>2.72</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.14</td>
</tr>
<tr>
<td>2</td>
<td>42.0</td>
</tr>
<tr>
<td>3</td>
<td>12.7</td>
</tr>
<tr>
<td>4</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Example Task

```c
task saxpy(is: ispace(int1d), x: region(is, float),
            y: region(is, float), a: float)

where
    reads(x, y), writes(y)

do
    for i in is do
        y[i] += a*x[i]
    end
end
```
Regions

- Regions can be partitioned into subregions
- Partitioning is a primitive operation
  - Supports describing arbitrary subsets of a region
Tasks

- Tasks can call subtasks
  - Sequential semantics, implicit parallelism
  - If tasks do not interfere, can be executed in parallel

```
task foo(x,y,z: region(...))
where reads writes(x,y,z) do
  bar(y,x)
  bar(x,y)
  bar(x,z)
  bar(z,y)
end

task bar(r,s: region(...)) where reads(r), writes(s)
```
Mapping Interface

- Mapper selects:
  - Where tasks run
  - Where regions are placed

- Mapping computed dynamically

- Decouple correctness from performance
Legion Architecture

- **Applications**
  - Realm
  - Legion (runtime)
  - Regent (compiler)
- **Mappers**
  - DSL compilers
  - Bishop (compiler)
- **Type System**
- **Data Model/Partitioning**
- **Func/Perf Verif Tools**
- **POSIX**
- **pthreads**
- **CUDA**
- **GASNet**
- **libnuma**
- **Isometry (DMA)**
Regent: A Legion Language

- Easy to use and significantly less code
- Type checker for Legion semantics
- Compiler matches performance of hand-written Legion (including kernels: vectorization, GPU, etc.)

```plaintext
task saxpy(is : ispace(int1d), x: region(is, float), 
           y: region(is, float), a: float)

where reads(x, y), writes(y) do
  for i in is do
    y[i] += a*x[i]

end
end
```
**S3D: Task Parallelism**

- One call to *Right-Hand-Side-Function (RHSF)* as seen by the Legion runtime
  - Called 6 times per time step by Runge-Kutta solver
  - Width == task parallelism
  - H2 mechanism (only 9 species)
  - Heptane (52 species) is significantly wider
**S3D: Performance Portability**

- Keeneland: “experimental” system
- S3D communication patterns made mapping decision hard
  - So-called “obvious” mapping was terrible
  - Best mapping was input-dependent
- Trivial to try all CPU/GPU combinations and dynamically choose best

![S3D Mapping Alternatives on Keeneland](image)
S3D: Scalability

- Titan
- Heptane
- 643 points per node

![Graph showing scalability of S3D on Titan and Heptane.](image)
S3D: Advancing Science

- Simulated “Primary Reference Fuel” mechanism
  - Too computationally intensive until now
- Switched machines halfway through
  - Performance-tuned for new machine in hours

![Graph showing performance comparison between Legion S3D and MPI Fortran S3D](image-url)
Legion Summary

- The programmer
  - Describes the structure of the program’s data (regions)
  - The tasks that operate on that data

- The Legion runtime
  - Guarantees tasks appear to execute in sequential order
  - Ensures tasks have the correct versions of their regions

- The Regent language
  - Type system checks correctness of programs
  - Significantly easier to use, less code
  - Compiler matches performance of hand-written Legion
ParSEC

- Parallel Runtime Scheduling and Execution Controller
  - Generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures

- Applications are Direct Acyclic Graph (DAG) of tasks
  - Labeled edges designating data dependencies
  - DAGs are compact problem-size independent format

- PaRSEC assigns computation threads to the cores, overlaps communications and computations and uses a dynamic, fully-distributed scheduler

- Framework includes libraries, a runtime system, and development tools for porting applications to heterogeneous environments

http://icl.cs.utk.edu/parsec
ParSEC Toolchain

- Serial representation can be automatically translated in the JDF (DAG) representation
- JDF is recompiled as C-code and with PaRSEC
Serial to DAG (Tiled QR Factorization)

- Serial code are loops of identifiable tasks
- Latent parallelism

PaRSEC compiler performs symbolic analysis of data flows, and transforms into internal representation of the DAG (JDF)

```plaintext
FOR k = 0 .. SIZE-1
    A[k][k], T[k][k] <- DGEQRT(A[k][k])
FOR m = k+1 .. SIZE-1
    A[k][k], A[m][k], T[m][k] <- DTSQRT(A[k][k], A[m][k], T[m][k])
FOR n = k+1 .. SIZE-1
    A[k][n] <- DORMQR(A[k][k], T[k][k], A[k][n])
    A[k][n], A[m][n] <- DSSMQR(A[m][k], T[m][k], A[k][n], A[m][n])
```
DAG Scheduling in ParSEC (1/2)

- Scheduler in PaRSEC is divided in two parts
  - Generic, handling the placement of ready tasks into the actual scheduling infrastructure
  - Algorithm dependent, using knowledge of algorithm
- Scheduler is partly in the library and partly in the opaque object encapsulating the algorithm
- Internally, PaRSEC creates a thread per core on the local machine and binds them on each core
  - Each thread runs its own version of the scheduler, alternating between times when it runs the body of the task, and times when it executes the local scheduler to find a new task to run
- Scheduling is not only distributed between the nodes, it is also distributed between the cores
To improve locality and data reuse on NUMA, the scheduler favors the queuing of the new enabled tasks in the local queue of the calling thread.

If the thread-local queue is full, the tasks are put on a single node-local waiting-queue.

When a task is completed, an epilogue, compiled from the JDF and found in the opaque object, is executed to compute which tasks are now enabled.

ParSEC uses the HWLOC library to discover machine architecture and define the work-stealing strategy.
Task Dependencies and Data Flow (1/2)

- PaRSEC needs to track both task dependencies and data flows between tasks.
- When a task is completed, the epilogue stores pointers to each of the data produced by the task in a structure shared between the threads.
- When a new task is to be scheduled, the pointers corresponding to all local variables are retrieved from this shared structure, and assigned to the local variables, before the task is executed.
- When all tasks depending on a particular data are completed, the engine stops tracking this data, and releases all internal resources associated with it.
Epilogue ends by calling the Asynchronous Communication Engine to trigger the emission of the output data
- Implemented as a separate thread (one per node)
- Task completions which enable remote tasks (by satisfying their dependencies) are signified from one communication engine to the others using small control messages
- When a communication engine learns that a remote task has completed, it computes which local tasks this enables, and sends a control message to receive the corresponding data, then pull

In PaRSEC, communications are implicitly inferred from the data dependencies between tasks
- Asynchrony and dynamic scheduling are key PaRSEC concepts
- Effectively achieve communication/computation overlap and asynchronous progress of tasks in a distributed environment
StarPU

- See slides