Two Topics: IO & Control Replication

CS315B
Lecture 10
I/O in Parallel Programming

• I/O tends to be an afterthought in parallel programming systems

• Many papers ignore I/O time in reported results!

• But in real life, I/O time is ... time
Regent I/O

• The situation is better with Regent

• Already have the notion
  • There are distinct collections of data
    • regions
  • That can be in different places, have different layouts, etc.
  • And the details are kept abstract
    • Programmer doesn’t need to know how data is accessed
Regent I/O Outline

• Interpret files as regions
  • Integrate I/O into the programming model

• Why?
  • Want to overlap I/O with computation
  • Need to define consistency semantics

• Bottom line
  • I/O is (almost) like any other data movement
Attach Operation

- Attach external resource to a region
  - Normal files, formatted files (HDF5), ...

<table>
<thead>
<tr>
<th>IndexSpace ⇔ HDF DataSpace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fields ⇔ HDF Datasets</td>
</tr>
<tr>
<td>Fields ⇔ HDF Datasets</td>
</tr>
<tr>
<td>Fields ⇔ HDF Datasets</td>
</tr>
<tr>
<td>Fields ⇔ HDF Datasets</td>
</tr>
<tr>
<td>Fields ⇔ HDF Datasets</td>
</tr>
<tr>
<td>Fields ⇔ HDF Datasets</td>
</tr>
</tbody>
</table>
Attach Operation

Semantics

- Invalidate existing physical instance of \( lr \)
- Maps \( lr \) to a new physical instance that represents external data (no external I/O)

![Diagram showing Region \( lr \) and instances on Node 1 and Node 2 with connections to Legion Runtime and Application](image)
Attach Operation

Semantics

- Invalidate existing physical instance of $lr$
- Maps $lr$ to a new physical instance that represents external data (no external I/O)

```
Region $lr$
```

```
Node 1
  Instance 1
  Instance 2

Node 2
  Instance 3
   Attach
     Instance 4

External Resource

Application

Legion Runtime
```
Digression: Task Coherence

**Privileges**
- Reads
- Reads/Writes
- Reduces (with operator)

**Coherence**
- Exclusive
- Atomic
- Simultaneous
- Relaxed

- Coherence declarations are wrt *sibling* tasks
Attach Operation

• Attached region accessed using *simultaneous coherence*
  • Different tasks access the region simultaneously
  • Requires that all tasks must use the *only valid* physical instance

• *Copy restriction*
  • Simultaneous coherence implies tasks cannot create local copies
  • May result in inefficient memory accesses
Acquire/Release

• For regions with simultaneous coherence

• Acquire removes the copy restriction
  • Can create copies in any memory
  • Up to application to know this is OK!

• Release restores the copy restriction
  • Invalidates all existing local copies
  • Flushes dirty data back to the file
Acquire/Release Example
Opaque Data Sources

• Can also attach to sources that are other programs
  • E.g., read/write in-memory data structures from another process

• Done through a serialization/deserialization interface
  • Attach specifies the ser/des routines
S3D I/O Example

- A production combustion simulation
- Checkpoint after fixed # of time steps
I/O Summary

• Definitely a useful feature!

• And less mature than other features
  • But simple cases will work fine

• Let us know if you need/want to use I/O
Control Replication
Implicit Parallel Programming Template

while (...) do
  for R in Parts do
    task1(R)
  end
  for R in Parts do
    task2(R)
  end
end
How Do We Scale This Program?

while (...) do
    for R in Parts do
        task1(R)
    end
    for R in Parts do
        task2(R)
    end
end

- Make more Parts
- Make each subregion $R$ smaller
Amdahl Strikes Back

• Recall Amdahl’s law
  • Parallel speedup is limited by the sequential portion left un-parallelized
  • There is some sequential overhead to launching tasks on a single processor

• If we double the # of subregions
  • Each subregion is ½ the size, so <= ½ of the work
  • Launch overhead doubles
  • Useful compute/overhead ratio decreases by >= 4X
task T()
    while (...) do
        for R in Parts do
            task1(R)
        end
        for R in Parts do
            task2(R)
        end
    end
}
Analysis

task T(){
    while (...) do
        for R in Parts do
            task1(R)
        end
        for R in Parts do
            task2(R)
        end
    end
}
Analysis

task T(){
while (...) do
  for R in Parts do
    task1(R)
  end
  for R in Parts do
    task2(R)
  end
end
}

|Parts| = 2
|Task launch| = 100 usecs
|task1| = .5 seconds
|task2| = .5 seconds

400 usecs

Total: 1 second
Analysis

task T(){
  while (...) do
    for R in Parts do
      task1(R)
    end
    for R in Parts do
      task2(R)
    end
  end
}

|Parts| = 4
|Task launch| = 100 usecs
|task1| = 250 msecs
|task2| = 250 msecs

800 usecs

Total: 500 msecs
Analysis

task T()
while (...) do
  for R in Parts do
    task1(R)
  end
  for R in Parts do
    task2(R)
  end
end
task T() {
    while (...) do
        for R in Parts do
            task1(R)
        end
        for R in Parts do
            task2(R)
        end
    end
}
task \( T() \) {
  while (...) do
    for \( R \) in Parts do
      task1(R)
    end
    for \( R \) in Parts do
      task2(R)
    end
  end
}

\(|\text{Parts}| = 32\)
\(|\text{Task launch}| = 100 \text{ usecs}\)
\(|\text{task1}| = 30 \text{ msecs}\)
\(|\text{task2}| = 30 \text{ msecs}\)

Total: 60 msecs
Analysis

task T() {
    while (...) do
        for R in Parts do
            task1(R)
        end
        for R in Parts do
            task2(R)
        end
    end
}

|Parts| = 64
|Task launch| = 100 usecs
|task1| = 15 msecs
|task2| = 15 msecs

Total: 30 msecs
Analysis

task T(){
    while (...) do
        for R in Parts do
            task1(R)
        end
        for R in Parts do
            task2(R)
        end
    end
}

|Parts| = 128
|Task launch| = 100 usecs
|task1| = 7.5 msecs
|task2| = 7.5 msecs

Total: 26 msecs
What Does That Mean?

while (...) do
    for R in Parts do
        task1(R)
    end
    for R in Parts do
        task2(R)
    end
end

• Can scale this program to 8 or 16 nodes
  • Should be more, but...

• We want to run on 100’s or 1,000’s of nodes
SPMD Programming Revisited

• Recall that SPMD programs
  • Launch 1 task per processor at program start-up
  • These tasks run for the duration of the program
  • Tasks explicitly communicate to exchange data

• Notice
  • SPMD programs launch the minimum # of tasks to keep the machine busy
  • These tasks run for the maximum amount of time
  • Best possible launch overhead/work ratio!
How Do We Scale This Program?

while (...) do
  for R in Parts do
    task1(R)
  end
  for R in Parts do
    task2(R)
  end
end

must_epoch
  for i = 1,num_tasks do
    task(part[i],phaseb[i])
  end

where

tasks know which other tasks they have to communicate with
The Price

• SPMD programs minimize distributed overheads related to control

• The price is explicit parallel programming
  • Tasks must communicate with each other while they execute
  • Introduces synchronization, message passing ...
Implicit Parallelism

Traditional auto-parallelization
[Irigoin 91; Blume 95; Hall 96; ...]

for step = 0, nsteps:
  for i, j in grid:
    out[i,j] = F(in[i,j], in[i+1, j], ...)
  ...

Inspector/executor method
[Crowley 89; Ravishankar 12; ...]

for step = 0, nsteps:
  for c in mesh:
    out[c] = G(in[c], in[neighbor[c]])
  ...

- Requires static analysis of individual memory accesses
- Limited applicability

- Requires dynamic analysis of individual memory accesses
- Expensive runtime analysis
Task-Based Implicit Parallelism

```python
task tF(out, in):
    for i, j in out:
        out[i,j] = F(in[i,j], in[i+1, j], ...)
    for step = 0, nsteps:
        for sg in grid:
            tF(out[sg], in[sg])
    ...

task tG(out, in):
    for c in out:
        out[c] = G(in[c], in[neighbor[c]])
    for step = 0, nsteps:
        for sm in mesh:
            tG(out[sm], in[sm])
    ...
```

- User specifies coarse-grain tasks (and data)
- Analysis performed at level of tasks (instead of iterations)
- Dynamic analysis is better but still expensive
Task Execution (Not Replicated)

- Sequential execution: tasks form a stream in program order
- System discovers parallelism by analyzing dependencies
- Dataflow is scheduled and copies are inserted as needed
Control Replication

- Technique to generate scalable SPMD code from implicitly parallel (task-based) programs

- Asymptotic reduction in steady state analysis
  - $O(1)$ instead of $O(N)$ in number of nodes
Task Execution (Replicated)

- stream(s) of tasks
- dataflow
- execution schedule
- control replication

- stream 0
- stream 1
- stream 0
- stream 1
- node 0
- node 1
- node 0
- node 1

- implicitly parallel
- explicitly parallel

- control replication

- sync
- copy

- dependence analysis
- scheduling

- analysis is now parallel
Control Replication
Control Replication

• Regent can do this for you!

• `__demand(__replicable)`

• Takes a program in implicit parallel style, converts it to SPMD style

• Restrictions
  • Each “rank” must execute the same sequence of Legion API calls
    • i.e., the control code is replicated in each rank
Control Replication

• We recommend using control replication for your project
  • Write in implicit style

• Should scale to 256-512 nodes
  • At least