MODULE ONE: INTRODUCTION

Speaker, Date
MODULE OVERVIEW

Topics to be covered

- Introduction to parallel programming
- Common difficulties in parallel programming
- Introduction to OpenACC
- Parallel programming in OpenACC
INTRODUCTION TO PARALLEL PROGRAMMING
WHAT IS PARALLEL PROGRAMMING?

- “Performance Programming”
- Parallel programming involves exposing an algorithm’s ability to execute in parallel
- This may involve breaking a large operation into smaller tasks (task parallelism)
- Or doing the same operation on multiple data elements (data parallelism)
- Parallel execution enables better performance on modern hardware
A REAL WORLD CASE STUDY
Modern cancer research

- The Russian Academy of Science created a program to simulate light propagation through human tissue.
- This program was used to be able to more accurately detect cancerous cells by simulating billions of random paths that the light could take through human tissue.
- With parallel programming, they were able to run thousands of these paths simultaneously.
- The sequential program took 2.5 hours to run.
- The parallel version took less than 2 minutes.
WHAT IS PARALLEL PROGRAMMING?
A real world example

- A professor and his 3 teaching assistants (TA) are grading 1,000 student exams
- This exam has 8 questions on it
- Let’s assume it takes 1 minute to grade 1 question on 1 exam
- To maintain fairness, if someone grades a question (for example, question #1) then they must grade that question on all other exams
- The following is a sequential version of exam grading
SEQUENTIAL SOLUTION

Grade Exams 1-1000: Questions #1, 2, 3, 4, 5, 6, 7, 8: 8000m
SEQUENTIAL SOLUTION

Prof

Exams 1-1000 : Q #1 : 1000m

Exams 1-1000 : Q #2 : 1000m

Exams 1-1000 : Q #3 : 1000m

Exams 1-1000 : Q #4 : 1000m

Exams 1-1000 : Q #5 : 1000m

Exams 1-1000 : Q #6 : 1000m

Exams 1-1000 : Q #7 : 1000m

Exams 1-1000 : Q #8 : 1000m

8000+ m
PARALLEL SOLUTION

- Exams 1-250: Q #1, 2: 500m
- Exams 251-500: Q #3, 4: 500m
- Exams 501-750: Q #1, 2: 500m
- Exams 751-1000: Q #3, 4: 500m
- Exams 1-250: Q #3, 4: 500m
- Exams 251-500: Q #5, 6: 500m
- Exams 501-750: Q #5, 6: 500m
- Exams 751-1000: Q #5, 6: 500m
- Exams 1-250: Q #5, 6: 500m
- Exams 251-500: Q #7, 8: 500m
- Exams 501-750: Q #7, 8: 500m
- Exams 751-1000: Q #7, 8: 500m
- Exams 1-250: Q #7, 8: 500m
- Exams 251-500: Q #7, 8: 500m
- Exams 501-750: Q #7, 8: 500m
- Exams 751-1000: Q #7, 8: 500m
- Exams 1-250: Q #7, 8: 500m
- Exams 251-500: Q #7, 8: 500m
- Exams 501-750: Q #7, 8: 500m
- Exams 751-1000: Q #7, 8: 500m
- Exams 1-250: Q #7, 8: 500m
- Exams 251-500: Q #7, 8: 500m
- Exams 501-750: Q #7, 8: 500m
- Exams 751-1000: Q #7, 8: 500m
- Exams 251-500: Q #7, 8: 500m
- Exams 501-750: Q #7, 8: 500m
- Exams 751-1000: Q #7, 8: 500m
- Exams 1-250: Q #7, 8: 500m
- Exams 251-500: Q #7, 8: 500m
- Exams 501-750: Q #7, 8: 500m
- Exams 751-1000: Q #7, 8: 500m
PIPELINE STALL

Q #1, 2
2m
Q #3, 4
2m
Q #5, 6
2m
Q #7, 8
2m
Q #1, 2
2m
Q #3, 4
2m
Q #5, 6
2m
Q #7, 8
2m
Q #1, 2
2m
Q #3, 4
2m
Q #5, 6
2m
Q #7, 8
2m
Q #1, 2
2m
Q #3, 4
2m
Q #5, 6
2m
Q #7, 8
2m
Q #1, 2
2m
Q #3, 4
2m
Q #5, 6
2m
Q #7, 8
2m
Q #1, 2
2m
Q #3, 4
2m
Q #5, 6
2m
Q #7, 8
2m
Q #1, 2
2m
Q #3, 4
2m
Q #5, 6
2m
Q #7, 8
2m
Q #1, 2
2m
Q #3, 4
2m
Q #5, 6
2m
Q #7, 8
2m
Q #1, 2
2m
Q #3, 4
2m
Q #5, 6
2m
Q #7, 8
2m
Q #1, 2
2m
Q #3, 4
2m
Q #5, 6
2m
Q #7, 2006+ m

OpenACC
GRADING EXAMPLE SUMMARY

It’s critical to understand the problem before trying to parallelize it

- Can the work be done in an arbitrary order, or must it be done in sequential order?
- Does each task take the same amount of time to complete? If not, it may be necessary to "load balance."

In our example, the only restriction is that a single question be graded by a single grader, so we could divide the work easily, but had to communicate periodically.

- This case study is an example of task-based parallelism. Each grader is assigned a task like “Grade questions 1 & 2 on the first 500 tests”

- If instead each question could be graded by different graders, then we could have data parallelism: all graders work on Q1 of the following tests, then Q2, etc.
AMDAHL’S LAW
Amdahl’s law is an observation that how much speed-up you get from parallelizing the code is limited by the remaining serial part.

Any remaining serial code will reduce the possible speed-up.

This is why it’s important to focus on parallelizing the most time consuming parts, not just the easiest.
APPLYING AMDAHL’S LAW

Estimating Potential Speed-up

- What’s the maximum speed-up that can be obtained by parallelizing 50% of the code?
  
  \[ \frac{1}{100\% - 50\%} = \frac{1}{1.0 - 0.50} = 2.0X \]

- What’s the maximum speed-up that can be obtained by parallelizing 25% of the code?
  
  \[ \frac{1}{100\% - 25\%} = \frac{1}{1.0 - 0.25} = 1.3X \]

- What’s the maximum speed-up that can be obtained by parallelizing 90% of the code?
  
  \[ \frac{1}{100\% - 90\%} = \frac{1}{1.0 - 0.90} = 10.0X \]
INTRODUCTION TO OPENACC
OpenACC is a directives-based programming approach to parallel computing designed for performance and portability on CPUs and GPUs for HPC.

```c
main()
{
    <serial code>
    #pragma acc kernels
    {
        <parallel code>
    }
}
```
3 WAYS TO ACCELERATE APPLICATIONS

- **Libraries**: Easy to use, Most Performance
- **Compiler Directives**: Easy to use, Portable code, OpenACC
- **Programming Languages**: Most Performance, Most Flexibility

OpenACC
OPENACC PORTABILITY

Describing a generic parallel machine

- OpenACC is designed to be portable to many existing and future parallel platforms.
- The programmer need not think about specific hardware details, but rather express the parallelism in generic terms.
- An OpenACC program runs on a host (typically a CPU) that manages one or more parallel devices (GPUs, etc.). The host and device(s) are logically thought of as having separate memories.
OPENACC
Three major strengths

- Incremental
- Single Source
- Low Learning Curve
OPENACC

• Maintain existing sequential code
• Add annotations to expose parallelism
• After verifying correctness, annotate more of the code

Begin with a working sequential code.
Parallelize it with OpenACC.
Rerun the code to verify correct behavior, remove/alter OpenACC code as needed.

```
for( i = 0; i < N; i++ )
{
  < loop code >
}

#pragma acc parallel loop
for( i = 0; i < N; i++ )
{
  < loop code >
}
```

```
#pragma acc parallel loop
for( i = 0; i < N; i++ )
{
  < loop code >
}
```
OPENACC

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code
OPENACC

Supported Platforms
- POWER
- Sunway
- x86 CPU
- x86 Xeon Phi
- NVIDIA GPU
- PEZY-SC

Single Source
- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

The compiler can **ignore** your OpenACC code additions, so the same code can be used for **parallel** or **sequential** execution.

```c
int main(){
...
    #pragma acc parallel loop
    for(int i = 0; i < N; i++)
        < loop code >
}
```
**OPENACC**

<table>
<thead>
<tr>
<th>Incremental</th>
<th>Single Source</th>
<th>Low Learning Curve</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ Maintain existing sequential code</td>
<td>▪ Rebuild the same code on multiple architectures</td>
<td></td>
</tr>
<tr>
<td>▪ Add annotations to expose parallelism</td>
<td>▪ Compiler determines how to parallelize for the desired machine</td>
<td></td>
</tr>
<tr>
<td>▪ After verifying correctness, annotate more of the code</td>
<td>▪ Sequential code is maintained</td>
<td></td>
</tr>
</tbody>
</table>
OpenACC is meant to be easy to use, and easy to learn. The programmer remains in familiar C, C++, or Fortran. No reason to learn low-level details of the hardware.

The compiler will get hints from the programmer about which parts of the code to parallelize. Then, the compiler will generate parallelism for the target parallel hardware.
OPENACC

Incremental
- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

Single Source
- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

Low Learning Curve
- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.
Massively Scaling Computational Electromagnetics Code Using OpenACC

NekCEM, or Nekton for Computational Electromagnetics, is a code designed for highly efficient, accurate predictive modeling of physical systems arising in electromagnetics, photonics, electronics, quantum mechanics, and accelerator physics. It’s used in the design of large particle accelerators for producing high-energy photons and in the design of photonic and semiconductor devices for solar energy production. “NekCEM enables researchers to prototype advanced numerical algorithms for solving the underlying partial differential equations at extreme scales of parallelism,” said Dr. Misun Min, a computational scientist at Argonne National Laboratory. “Simulation-based investigation with NekCEM will help research communities understand fundamental physics over range of length scales, with extreme-scale computing capability on the future-generation HPC platforms.” Such research, said Min, can significantly reduce the cost and risk of the design and analysis of physical systems.

Challenge
Dr. Min’s team needed to process enormous amounts of data with a high degree of accuracy. Frequent performance bottlenecks, excessive time required for data processing, and the need to scale to future GPU-based architectures were the team’s main challenges. “Our principal challenge is rapid time to solution,” said Min. “NekCEM strong-scales to a few hundred points per core on the Argonne Leadership Computing Facility (ALCF) Blue Gene/P and Blue Gene/Q. Our biggest challenge on next-generation architectures is to be able to strong-scale but keep low solution times for problems involving a few hundred million grid points.”

A second challenge is maintaining code portability. NekCEM is part of a larger code base that has been in development for decades and has hundreds of users. “During this time, many architectures have come and gone, and we cannot port to models for which there is little demand,” said Min. “That OpenACC is truly open source was an important factor in our decision to use it for our research.”

SUCCESS STORY

LSDalton
Quantum Chemistry
Aarhus University
12X speedup
1 week

PowerGrid
Medical Imaging
University of Illinois
40 days to
2 hours

COSMO
Weather and Climate
MeteoSwiss, CSCS
40X speedup
3X energy efficiency

INCOMP3D
CFD
NC State University
4X speedup

NekCEM
Comp Electromagnetics
Argonne National Lab
2.5X speedup
60% less energy

MAESTRO
Astrophysics
Stony Brook University
4.4X speedup
4 weeks effort

CloverLeaf
Comp Hydrodynamics
AWE
4X speedup
Single CPU/GPU code

FINE/Turbo
CFD
NUMECA International
10X faster routines
2X faster app
OPENACC RESOURCES

Resources
https://www.openacc.org/resources

Success Stories
https://www.openacc.org/success-stories

Events
https://www.openacc.org/events

FREE Compilers

Compilers and Tools
https://www.openacc.org/tools

Download & Tools
OpenACC compilers, profiling tools, and libraries are designed and available to download from multiple vendors and academic organizations.

Commercial Compilers
PGI

cr"1

Open Source Compilers
PGI Academic Compilers with OpenACC Edition

Contact PGI for pricing information.

PGI Academic Licensing Contact for Non-commercial User

Contact OpenACC Licensing Contact for non-commercial users.

https://www.openacc.org/tools/

Events
The OpenACC community organizes a variety of events throughout the year. Events range from talks at conferences and workshops to webinars, online courses, and more. Join us at one of the upcoming events around the world as we learn OpenACC programming and to participate in activities with the OpenACC team.!
EXPRESSING PARALLELISM WITH OPENACC
Array pairing example

```c
void pairing(int *input, int *output, int N)
{
    for(int i = 0; i < N; i++)
        output[i] = input[i*2] + input[i*2+1];
}
```
Array pairing example

subroutine pairing(input, output, N)

    do i=1,N
        output(i) = input(i*2) + input(i*2+1);
    end do

end subroutine
C O D I N G  W I T H  O P E N A C C

Array pairing example - parallel

```c
void pairing(int *input, int *output, int N){
    #pragma acc parallel loop
    for(int i = 0; i < N; i++)
        output[i] = input[i*2] + input[i*2+1];
}
```
CODING WITH OPENACC
Array pairing example - parallel

```
subroutine pairing(input, output, N)
  !$acc parallel loop
  do i=1,N
    output(i) = input(i*2) + input(i*2+1);
  end do
end subroutine
```

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
DATA DEPENDENCIES

Not all loops are parallel

```c
void pairing(int *a, int N){
    for(int i = 1; i < N; i++)
        a[i] = a[i] + a[i-1];
}
```

```
1  2  6 10 15 21 28 36 45 55
```

```
i=1  i=2  i=3  i=4  i=5  i=6  i=7  i=8  i=9
```
DATA DEPENDENCIES

Not all loops are parallel

```c
void pairing(int *a, int N){
    #pragma acc parallel loop
    for(int i = 1; i < N; i++)
        a[i] = a[i] + a[i-1];
}
```

If we attempted to parallelize this loop we would get wrong answers due to a *forward dependency.*

Sequential

Parallel
DATA DEPENDENCIES

Not all loops are parallel

```c
void pairing(int *a, int N){
    #pragma acc parallel loop
    for(int i = 1; i < N; i++)
        a[i] = a[i] + a[i-1];
}
```

Even changing how the iterations are parallelized will not make this loop safe to parallelize.
DATA DEPENDENCIES

Not all loops are parallel

```fortran
subroutine pairing(a, N)

    do i = 1, N
        a(i) = a(i) + a(i-1)
    end do

end subroutine
```

![Diagram showing data dependencies]

- `i=1`: 1
- `i=2`: 2
- `i=3`: 6
- `i=4`: 14
- `i=5`: 15
- `i=6`: 261
- `i=7`: 28
- `i=8`: 36
- `i=9`: 45
- `i=10`: 56

The arrows indicate dependencies between the values of `a(i)` for different `i` values.
DATA DEPENDENCIES

Not all loops are parallel

```fortran
subroutine pairing(a, N)
  !$acc parallel loop
  do i = 1,N
    a(i) = a(i) + a(i-1)
  end do
end subroutine
```

If we attempted to parallelize this loop we would get wrong answers due to a forward dependency.

Sequential

Parallel
DATA DEPENDENCIES

Not all loops are parallel

subroutine pairing(a, N)
    !$acc parallel loop
    do i = 1,N
        a(i) = a(i) + a(i-1)
    end do
end subroutine

Even changing how the iterations are parallelized will not make this loop safe to parallelize.
MODULE 1 REVIEW
Parallel programming is the only way to fully utilize modern, parallel hardware.

The key idea parallel programming is split up tasks within a program in a way that they can be run in parallel (at the same time).

Parallel programming requires the programmer to understand their program and which operations can be performed in parallel and the communicate that to the system.

OpenACC is a high-level, easy to use, directive based programming standard designed to parallelize sequential programs.
THANK YOU