Compsci 590.3: Introduction to Parallel Computing

Alvin R. Lebeck

Slides based on those from the University of Oregon and Intel
Admin

Logistics
• Homework 2
  – Due next Tuesday

Outline
• Vectorization
• Offload programming
• AoS vs. SoA
• Performance scalability
• Analytical performance measures
• Amdahl’s law and Gustafson-Barsis’ law
Vectorization

- Vectors/SIMD/data parallel instructions
  - Perform same operation on multiple elements
- For any optimization option –O2 or higher icc/icpc automatically vectorizes the code
- You may need to transform code to help compiler
- Use –vec-report=<n> to guide you.
Matrix Multiplication Example

- Compile with \texttt{--O2 --vec-report=5}

Line: code
72: for (i=0; i<N; i++){
73: for (j=0; j<M; j++){
74: for(k=0; k<P; k++){
75: C[i][j] += A[i][k] * B[k][j];
76: }
77: }
78: }

- \texttt{~50} seconds w/ vectorization
- \texttt{~56} seconds w/o vectorization
  - icpc \texttt{--O2 --no-vec...}

LOOP BEGIN at matmul_serial.c(74,4)
  remark #15542: loop was not vectorized: inner loop was already vectorized

LOOP BEGIN at matmul_serial.c(73,3)
  remark #15388: vectorization support: reference C has aligned access [matmul_serial.c(75,5)]
  remark #15388: vectorization support: reference C has aligned access [matmul_serial.c(75,5)]
  remark #15388: vectorization support: reference B has aligned access [matmul_serial.c(75,5)]
  remark #15399: vectorization support: unroll factor set to 4
  remark #15301: PERMUTED LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 2
  remark #15449: unmasked aligned unit stride stores: 1
  remark #15475: --- begin vector loop cost summary ---
  remark #15476: scalar loop cost: 9
  remark #15477: vector loop cost: 4.000
  remark #15478: estimated potential speedup: 2.240
  remark #15479: lightweight vector operations: 6
  remark #15480: medium-overhead vector operations: 1
  remark #15488: --- end vector loop cost summary ---
LOOP END
Requirements for Vectorization

- Straight line code (no branches)
  - Can be masked assignments
    - If \( b[i] \) \( a[i] = c[i] \times d[i] \);
- Countable loop
  - No while loops (or equivalent)
- No loop-carried dependencies (flow dependence across iterations)
  - for \( (i=0; i<N; i++) \)
    - \( d[i] = e[i] - a[i-1] \);
    - \( a[i-1] = b[i] + c[i] \);
- No special operators, function calls, etc. unless inlined and vectorized
- Inner loops not outer loops
- Helping the compiler
  - Transform code manually (permute loops)
  - Use pragmas (simd, ivdep, vector always)
  - You need to ensure correctness…
Parallel programming is the same on Intel® MIC and CPU
Heterogeneous Programming Model

- Programmer designates code sections to run on Intel® MIC target
  - No further programming / API usage is needed
  - Setup/teardown, data transfer, synchronization, are managed automatically by compiler and runtime
- Offload is optional
  - If Intel MIC device is missing, program may run entirely on CPU
    - If Intel MIC-specific code, may need alternate code path
    - There is an option to enforce failure if Intel MIC device is unavailable
Heterogeneous Memory Model (Non-shared Memory)

- CPU and Intel® MIC device do not share a common memory

- Two techniques are used to maintain program semantics with/without offload
  - Emulate shared data by copying back and forth at point of offload
    - Good for large blocks of contiguous data
  - Maintain coherence in a range of virtual addresses on CPU and Intel MIC device, automatically in software
    - Good for complex data structures
## Offload Compile/Run Overview

**CPU Program**

```c
f()
{
    #pragma offload
    target(mic)
    a = b + g();
}
```

```c
__attribute__((target(mic))) g()
{
}
```

```c
h()
{
}
```

**Contents of MIC Program**

```c
f_part1()
{
    a = b + g();
}
```

```c
__attribute__((target(mic))) g()
{
}
```

**Execution**

- At first offload, if Intel® MIC device is installed and available, MIC program is loaded
- At each offload, if Intel MIC device is present, statement is run on device, else statement runs on CPU
- At program termination, Intel MIC program is unloaded
Language Extensions for Offload (C/C++ pragmas)

- Offload **pragma** for data marshalling
  - `#pragma offload` in C/C++
- Offloads the following OpenMP block or Intel® Cilk™ Plus construct or function call or compound statement
- All functions that will execute on the card must be declared as targeted for offload (so compiled for host and MIC device)
  - In both the callee (required) and the caller (recommended)
    - `__attribute__((target(MIC)))` or `__declspec(target(mic))`
    - For many such declarations, use `#pragma offload_attribute (push,target(mic)) | (pop)`
- Offloaded data must be scalars, arrays, bit-wise copyable structs
  - Excludes all but simplest C++ classes
  - All data types can be used within the target code
  - Data copy is explicit
# Offload Directives (contd.)

Variables restricted to scalars, bwc structs, arrays and pointers to scalars/structs/arrays

<table>
<thead>
<tr>
<th>Clauses / Modifiers</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target specification</td>
<td>target ( name [:num] )</td>
<td>Where to run construct</td>
</tr>
<tr>
<td>Inputs</td>
<td>in (var-list modifiers_opt)</td>
<td>Copy CPU to target</td>
</tr>
<tr>
<td>Outputs</td>
<td>out (var-list modifiers_opt)</td>
<td>Copy target to CPU</td>
</tr>
<tr>
<td>Inputs &amp; outputs</td>
<td>inout (var-list modifiers_opt)</td>
<td>Copy both ways</td>
</tr>
<tr>
<td>Non-copied data</td>
<td>nocopy (var-list modifiers_opt)</td>
<td>Data is local to target</td>
</tr>
<tr>
<td><strong>Modifiers</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specify pointer length</td>
<td>length (element-count-expr)</td>
<td>Copy that many pointer elements</td>
</tr>
<tr>
<td>Control pointer memory allocation</td>
<td>alloc_if ( condition )</td>
<td>Allocate/free new block of memory for pointer if condition is TRUE</td>
</tr>
<tr>
<td></td>
<td>free_if ( condition )</td>
<td></td>
</tr>
<tr>
<td>Alignment for pointer memory allocation</td>
<td>align ( expression )</td>
<td>Specify minimum data alignment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
#pragma offload_attribute (push, target(mic))
double A[N] __attribute__((aligned(64)));
double B[N] __attribute__((aligned(64)));
double C[N] __attribute__((aligned(64)));
double sum = 0.0;
#pragma offload_attribute (pop)

void my_simple() {
#pragma offload target(mic) in(A,B,C,sum)
    {}
...
#pragma offload target(mic)
#pragma omp parallel for reduction(+:sum)
    for (i=0; i<N; i++){
        A[i] = (A[i] + B[i]) * C[i];
        sum += A[i];
    }
...
#pragma offload target(mic) out(A,sum)
    {}
}
Outline

- Vectorization
- Offload programming
- AoS vs. SoA
- Performance scalability
- Amdahl’s law and Gustafson-Barsis’ law
Array of Structures (AoS)

- May lead to better cache utilization if data is accessed randomly

```c
struct item {
    double a, b, c;
};

struct item myarray[N];

for (int i=0; i< N; i++)
    myarray[i].a = f(myarray[i].b, myarray[i].c);
```
Typically better for vectorization and avoiding false sharing

```c
struct item {
    double a[N], b[N], c[N];
};

struct item myitems;

for (int i=0; i< N; i++)
    myitems.a[i] = f(myitems.b[i], myitems.c[i]);
```
Data Layout Options

Array of Structures (AoS), padding at end

Array of Structures (AoS), padding after each structure

Structure of Arrays (SoA), padding at end

Structure of Arrays (SoA), padding after each component

FIGURE 6.20

Array of structures (AoS) versus structure of arrays (SoA). SoA form is typically better for vectorization and avoidance of false sharing. However, if the data is accessed randomly, AoS may lead to better cache utilization.

FIGURE 6.21

Data layout options for arrays of structures and structures of arrays. Data can be laid out structure-by-structure, structure-by-structure with padding per structure, or, for structure of array, array-by-array or array-by-array with padding. The structure of array form, either with or without padding, makes vectorization much easier.
Outline

- Vectorization
- Offload programming
- AoS vs. SoA
- Performance scalability
- Amdahl’s law and Gustafson-Barsis’ law
What is Performance?

- In computing, performance is defined by 2 factors
  - Computational requirements (what needs to be done)
  - Computing resources (what it costs to do it)
- Computational problems translate to requirements
- Computing resources interplay and tradeoff

\[
\text{Performance} \sim \frac{1}{\text{Resources for solution}}
\]

- Hardware
- Time
- Energy
- ... and ultimately
- Money
Why do we care about Performance?

- We evaluate performance to understand the relationships between requirements and resources
  - Decide how to change “solutions” to target objectives
  - Generally care about execution time (seconds)
  - Sometimes also care about Power/Energy (performance/Watt)

- Performance measures reflect decisions about how and how well “solutions” are able to satisfy the computational requirements

- “The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible.”
  Charles Babbage, 1791 – 1871

- If Parallel performance <= Sequential then don’t bother
Performance Expectation (Loss)

- If each processor is rated at $k$ MFLOPS and there are $p$ processors, should we see $k \times p$ MFLOPS performance?
- If it takes 100 seconds on 1 processor, shouldn’t it take 10 seconds on 10 processors?
- Several causes affect performance
  - Each must be understood separately
  - But they interact with each other in complex ways
    - Solution to one problem may create another
    - One problem may mask another
- Scaling (system, problem size) can change conditions
- Need to understand performance space
Performance and Scalability

- Evaluation
  - Sequential runtime ($T_{seq}$) is a function of
    - problem size and architecture
  - Parallel runtime ($T_{par}$) is a function of
    - problem size and parallel architecture
    - # processors used in the execution
  - Parallel performance affected by
    - algorithm + architecture

- Scalability
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem
  - Actually many ways to define scalability, so beware
    - Constant problem
    - Constant time
    - Constant error
Performance Metrics and Formulas

- $T_1$ is the execution time on a single processor
- $T_p$ is the execution time on a $p$ processor system
- $S(p)$ (Sp) is the speedup
  \[ S(p) = \frac{T_1}{T_p} \]
- $E(p)$ (Ep) is the efficiency
  \[ Efficiency = \frac{S_p}{p} \]
- $Cost(p)$ ($C_p$) is the cost
  \[ Cost = p \times T_p \]
- Parallel algorithm is cost-optimal
  - *Parallel time = sequential time* ($C_p = T_1$, $E_p = 100\%$)
Amdahl’s Law (Fixed Size Speedup)

- Let $f$ be the fraction of a program that is sequential
  - $1-f$ is the fraction that can be parallelized
- Let $T_1$ be the execution time on 1 processor
- Let $T_p$ be the execution time on $p$ processors
- $S_p$ is the speedup
  
  $S_p = \frac{T_1}{T_p}
  = \frac{T_1}{(fT_1 + (1-f)T_1 / p)}
  = \frac{1}{(f + (1-f)/p)}$

- As $p \rightarrow \infty$
  
  $S_p = \frac{1}{f}$
Amdahl’s Law and Scalability

- **Scalability**
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem

- **When does Amdahl’s Law apply?**
  - When the problem size is fixed
  - *Strong scaling* ($p \to \infty$, $S_p = S_\infty \to 1/f$)
  - Speedup bound is determined by the degree of sequential execution time in the computation, not # processors!!!
  - Uhh, this is not good … Why?
  - Perfect efficiency is hard to achieve
    - Also means we still need very good sequential performance!
Gustafson-Barsis’ Law and Scalability

- **Scalability**
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem

- **When does Gustafson’s Law apply?**
  - When the problem size can increase as the number of processors increases
  - Weak scaling \((S_p = 1 + (p-1)f_{par})\)
  - Speedup function includes the number of processors!!!
  - Can maintain or increase parallel efficiency as the problem scales
Amdahl versus Gustafson-Baris

Amdahl

serial work

parallelizable work

Time

P=1

P=2

P=4

P=8
Amdahl versus Gustafson-Baris

Gustafson-Baris

serial work

parallelizable work

Time

P=1
P=2
P=4
P=8
Recall: DAG Model of Computation

- Think of a program as a directed acyclic graph (DAG) of tasks
  - A task can not execute until all the inputs to the tasks are available
  - These come from outputs of earlier executing tasks
  - DAG shows explicitly the task dependencies

- Think of the hardware as consisting of workers (processors)

- Consider a greedy scheduler of the DAG tasks to workers
  - No worker is idle while there are tasks still to execute
Work-Span Model

- TP = time to run with P workers
- $T_1 = \text{work}$
  - Time for serial execution
    - execution of all tasks by 1 worker
  - Sum of all work
- $T_\infty = \text{span}$
  - Time along the critical path
- Critical path: Sequence of task execution (path) through DAG that takes the longest time to execute
  - Assumes an infinite # workers available
Work-Span Example

- Let each task take 1 unit of time
- DAG at the right has 7 tasks
- $T_1 = 7$
  - All tasks have to be executed
  - Tasks are executed in a serial order
  - Can the execute in any order?
- $T_\infty = 5$
  - Time along the critical path
  - In this case, it is the longest pathlength of any task order that maintains necessary dependencies
Lower/Upper Bound on Greedy Scheduling

- Suppose we only have $P$ workers
- We can write a work-span formula to derive a lower bound on $T_P$
  - $\max(T_1 / P, T_\infty) \leq T_P$
- $T_\infty$ is the best possible execution time

- **Brent’s Lemma** derives an upper bound
  - Capture the additional cost executing the other tasks not on the critical path
  - $T_P \leq \text{time for non-critical tasks} + \text{time for critical path}$
  - Assume can do so without overhead
  - $T_P \leq (T_1 - T_\infty) / P + T_\infty$
Consider Brent’s Lemma for 2 Processors

- $T_1 = 7$
- $T_\infty = 5$
- $T_2 \leq (T_1 - T_\infty) / P + T_\infty$
  \hspace{1cm} \leq (7 - 5) / 2 + 5
  \hspace{1cm} \leq 6
Amdahl was an optimist!

![Graph showing speedup vs. P]

- **Amdahl's Law**
- **Work-Span Bound**
- **Brent's Lemma**

Alvin R. Lebeck (slides from UOregen)
Estimating Running Time

- Scalability requires that $T_\infty$ be dominated by $T_1$

$$T_P \approx \frac{T_1}{P} + T_\infty \text{ if } T_\infty << T_1$$

- Increasing work hurts parallel execution proportionately
- The span (critical path) impacts scalability, even for finite $P$
Parallel Slack

- Sufficient parallelism implies linear speedup

\[ T_P \approx \frac{T_1}{P} \quad \text{if} \quad \frac{T_1}{T_\infty} \gg P \]

Linear speedup  \hspace{1cm} Parallel slack
Asymptotic Complexity

- Time complexity of an algorithm summarizes how the execution time grows with input size.
- Space complexity summarizes how memory requirements grow with input size.
- Standard work-span model considers only computation, not communication or memory.
- Asymptotic complexity is a strong indicator of performance on large-enough problem sizes and reveals an algorithm’s fundamental limits.
Scalable Parallel Computing

- Scalability in parallel architecture
  - Processor numbers
  - Memory architecture
  - Interconnection network
  - Avoid critical architecture bottlenecks

- Scalability in computational problem
  - Problem size
  - Computational algorithms
    - Computation to memory access ratio
    - Computation to communication ratio

- Parallel programming models and tools

- Performance scalability
Why Aren’t Parallel Applications Scalable?

- Sequential performance
- Critical Paths
  - Dependencies between computations spread across processors
- Bottlenecks
  - One processor holds things up
- Algorithmic overhead
  - Some things just take more effort to do in parallel
- Communication overhead
  - Spending increasing proportion of time on communication
- Load Imbalance
  - Makes all processor wait for the “slowest” one
  - Dynamic behavior
- Speculative loss
  - Do A and B in parallel, but B is ultimately not needed
Critical Paths

- Long chain of dependence
  - Main limitation on performance
  - Resistance to performance improvement

- Diagnostic
  - Performance stagnates to a (relatively) fixed value
  - Critical path analysis

- Solution
  - Eliminate long chains if possible
  - Shorten chains by removing work from critical path
Bottlenecks

- How to detect?
  - One processor A is busy while others wait
  - Data dependency on the result produced by A

- Typical situations:
  - N-to-1 reduction / computation / 1-to-N broadcast
  - One processor assigning job in response to requests

- Solution techniques:
  - More efficient communication
  - Hierarchical schemes for master slave

- Program may not show ill effects for a long time
- Shows up when scaling
Algorithmic Overhead

- Different sequential algorithms to solve the same problem
- All parallel algorithms are sequential when run on 1 processor
- All parallel algorithms introduce additional operations (Why?)
  - Parallel overhead
- Where should be the starting point for a parallel algorithm?
  - Best sequential algorithm might not parallelize at all
  - Or, it doesn’t parallelize well (e.g., not scalable)
- What to do?
  - Choose algorithmic variants that minimize overhead
  - Use two level algorithms
- Performance is the rub
  - Are you achieving better parallel performance?
  - Must compare with the best sequential algorithm
Factors which determine a program's performance are complex, interrelated, and sometimes hidden.

Application related factors
- Algorithms, dataset sizes, task granularity, memory usage patterns, load balancing, I/O communication patterns

Hardware related factors
- Processor architecture, memory hierarchy, I/O network

Software related factors
- Operating system, compiler/preprocessor, communication protocols, libraries
Utilization of Computational Resources

- Resources can be under-utilized or used inefficiently
  - Identifying these circumstances can give clues to where performance problems exist

- Resources may be “virtual”
  - Not actually a physical resource (e.g., thread, process)

- Performance analysis tools are essential to optimizing an application's performance
  - Can assist you in understanding what your program is "really doing"
  - May provide suggestions how program performance should be improved
Performance Analysis and Tuning: The Basics

- Most important goal of performance tuning is to reduce a program's wall clock execution time
  - Iterative process to optimize efficiency
  - Efficiency is a relationship of execution time
- So, where does the time go?
- Find your program's hot spots and eliminate the bottlenecks in them
  - **Hot spot**: an area of code within the program that uses a disproportionately high amount of processor time
  - **Bottleneck**: an area of code within the program that uses processor resources inefficiently and therefore causes unnecessary delays
- Understand *what, where, and how* time is being spent
Parallel Performance Engineering Process

- Implementation
  - Preparation
  - Performance Analysis
    - Program Tuning
      - Production
  - Refinement
    - Measurement
    - Analysis
    - Ranking

Alvin R. Lebeck (slides from UOregon)
Sequential Performance

- Sequential performance is all about:
  - How time is distributed
  - What resources are used where and when

- “Sequential” factors
  - Computation
    - choosing the right algorithm is important
    - compilers can help
  - Memory systems and cache and memory
    - more difficult to assess and determine effects
    - modeling can help
  - Input / output
Parallel Performance

- Parallel performance is about sequential performance AND parallel interactions
  - Sequential performance is the performance within each thread of execution
  - “Parallel” factors lead to overheads
    - concurrency (threading, processes)
    - communication (message passing or caching effects)
    - synchronization (both explicit and implicit)
  - Parallel interactions also lead to parallelism inefficiency
    - load imbalances
Sequential Performance Tuning

- Sequential performance tuning is a *time-driven* process
- Find the thing that takes the most time and make it take less time (i.e., make it more efficient)
- May lead to program restructuring
  - Changes in data storage and structure
  - Rearrangement of tasks and operations
- May look for opportunities for better resource utilization
  - Cache management is a big one
  - Locality, locality, locality!
  - Virtual memory management may also pay off
- May look for opportunities for better processor usage
Parallel Performance Tuning

- In contrast to sequential performance tuning, parallel performance tuning might be described as *conflict-driven* or *interaction-driven*.

- Find the points of parallel interactions and determine the overheads associated with them.

- Overheads can be the cost of performing the interactions:
  - Transfer of data
  - Extra operations to implement coordination

- Overheads also include time spent waiting:
  - Lack of work
  - Waiting for dependency to be satisfied
Interesting Performance Phenomena

- Superlinear speedup
  - Speedup in parallel execution is greater than linear
  - $S_p > p$
  - How can this happen?
- Need to keep in mind the relationship of performance and resource usage
- Computation time (i.e., real work) is not simply a linear distribution to parallel threads of execution
- Resource utilization thresholds can lead to performance inflections