CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 28: Process-Level Programming

Instructors: Dan Garcia, Justin Yokota

Agenda

- Coordination Game Debrief
- Multi-process programming

Have the number of votes per choice be four consecutive numbers, with the smallest one an even number

- N (must be even)
- N+1
- N+2
- N+3

Have the number of votes per choice be four consecutive numbers, with the smallest one an even number

N (must be even)
N+1
N+2
N+3

Have the number of votes per choice be four consecutive numbers, with the smallest one an even number

N (must be even)
N+1
N+2
N+3

Coordination Game: Student suggestion (Credit Tristan)

I suggest the following way of allocating votes:

Most people in person and everyone on Zoom: Take the SHA-256 cryptographic hash of your last name here (https://emn178.github.io/online-tools/sha256.html), copy the hash of your name and convert it to decimal here (https://www.rapidtables.com/convert/number/hex-to-decimal.html), then look at the last digit of your decimal hash, this will be known as your hash number below.

Your hash number should be a digit between 0-9.

Then we can apportion the votes according to the ratio on Friday (TBA on the day of).

10 people in person will refrain from voting until almost the end of the voting period to balance out random fluctuations in the votes.

Why this works: One of the requirements for a cryptographic hash is that it distributes all elements evenly (aka the hash is basically random), thus in the limit case as n (the number of students voting) goes to infinity, the ratios should work out exactly. Because we don't have an infinite amount of students voting there are going to be fluctuations in the data that prevent it from being perfect, thus we reserve some votes till the very end to balance out any inaccuracies due to randomness. I think the number of people that should refrain from voting till the end should be something like sqrt(num people voting), but I'm using ten as a manageable guesstimate.

BTW if someone finds a better hash than SHA-256 or do the stats to figure out a better estimate for the number of votes that we should reserve comment below.

Coordination Game: Staff Solution

- Largely similar to the idea suggested, actually
- The main difference was to take into consideration the systems we were running the algorithm on
- If run on a computer, this approach would work well
  - Minor issue: Last names aren't well-distributed, so the last name "Lee" would skew results a bit
- When dealing with students:
  - Con: Hard to run SHA; non-leading students aren't likely to have prepped this ahead of time
  - Con: Potential for mistake/bad timing of response
  - Con: No Thread IDs
  - Pro: Able to adjust the plan on the fly
  - Pro: Enough people are in-person that the actual lecture hall can be used as an asset

Coordination Game: Staff Solution

- Have everyone online go to random.org and select a random number from 1 to 4. At the same time, have in-person students log in to PollEv. Have the first 20 students (sqrt(N)+gap between possible win states is optimal) move to sit in the back row of the auditorium
- At the 1 minute mark, have all remaining students in the auditorium (except those in the back row) to select votes they personally think will lead to success. Because we have enough students in the back that can make adjustments, students are less worried about overshooting
- At 3 minutes, stop all non-back-row students voting. Use the remaining time to fine-tune the solution.

Coordination Game: Main Points

- In any parallel computing problem, it's very useful to have a manager, even if that manager ends up not doing any work directly. A single coordinator can make everything else much faster
- For speed, ideally you want to minimize the amount of communication that needs to be done; producing a single instruction for all to follow is better than trying to get the first N responses
- Know the system, and change your tactics accordingly

Coordination Game: Main Points

- Not directly related to parallelism, but: the engineering solution isn't always the best solution. Every project with an end user needs to account for how a non-expert may interact with your output, and if it's unintuitive, it'll likely perform worse than a less efficient but intuitive approach.
  - Keeping track of the human element is stereotypically a weak point of scientists/engineers, but is critical in many aspects of SWE and research (conveying specs to customers, inferring what the customer actually wants instead of what they say they want, properly onboarding a new project member).
  - Expert bias is real, and often means that your idea that you think is intuitive is entirely alien to other people
- I would personally recommend taking a psychology/intuitive design class if possible, just to be more aware of this side of project management.

Multithreading vs Multiprocess Code

- Threads: Different instruction sequences on the same process
  - Threads on the same process share memory
  - "Easy" to communicate
  - Limited to a set of cores wired to the same memory block (1 node)
  - Analogy: a "hive mind"; you can do multiple things at the same time, but it's still largely the same entity
- Processes: Largely independent from each other
  - Different processes can't share memory
  - "Difficult" and time consuming to communicate
  - Can expand to as many cores as you have available, over as many nodes as you want
  - Analogy: A group of people; each person does their own thing independently

Multiprocess Framework Overview

- While a multithreaded program is fundamentally one program, individual processes are essentially distinct program instances entirely
- Different processes don't share memory, but generally can share the same file system
- In a multithreaded program, the entire thing crashes if any single thread crashes. In a multiprocess program, each process runs independently, so if one process crashes/terminates, the others still keep going.
  - Can create "zombie processes" that stay alive past the main process, and just eat resources until a system restart happens.
- Because you don't share memory, you can't use locks or concurrency primitives
  - Effectively restricts multiprocess programs to problems that can be split into entirely independent tasks

Multiprocess Mayhem

- In a multiprocess program, each process runs independently, so if one process crashes/terminates, the others still keep going.
- Because each process is considered independent, it bypasses many of the restrictions the OS uses to police programs
  - Use tons of memory? The OS will kill your program before it gets too big
  - Try to access someone else's memory? The OS will force a segfault
  - Run in an infinite loop? The OS will prioritize other programs to ensure everyone gets their share of computation time.
- Can be used to create what's known as a fork bomb: You write a program that creates two copies of itself to run.
  - Ex. In Windows, %0|%0 (in a bash script) is a fork bomb
- Causes exponentially many copies of the same program to run, eventually crowding out all the "real" processes and causing the system to crash.
- Please don't try this at home. This is malware. I don't think it will permanently damage anything, but…

Multiprocess Framework Overview

- Inter-process communication is done by sending messages between nodes
- Generally, messages take a lot of time to transmit/communicate:
  - Time to initialize a message packet >> time to send one byte of a message >> time to perform memory operations
- Main engineering hurdle of a multiprocess program is to find a way to split a large problem into smaller independent problems while minimizing the number of message packets and total amount of data passed back and forth

Open MPI

- Open MPI is the system that we'll be using to showcase multiprocess computation
  - Used in many supercomputers for distributed programs
  - Relatively simple
- Primarily built for Linux systems, since all the major supercomputers nowadays are x86 Intel systems with a Linux OS.
- Unfortunately, it does NOT work on Windows.
  - Sorry, no code demos today!

Open MPI

- Open MPI is the system that we'll be using to showcase multiprocess computation
- Compiled with mpicc, which is a wrapper for gcc that enables multiprocess code
  - #include <mpi.h>
- Runs with mpirun command
  - Syntax: mpirun -n <number of processes>
- When run, this command copies the program to all available nodes and loads the program repeatedly
  - Compare OpenMP, which loads the program once, and does a fork/join during computation

Aside: Naming of OpenMP vs Open MPI

- What do you think OpenMP stands for?
  - Open Multi-Processing… for a multithreading library
- What do you think Open MPI stands for?
  - Open Message Passing Interface
- As far as I can tell these two groups are entirely independent organizations that decided to name their systems really similarly to each other
  - Open is used to indicate that the system is open-source, I think?
- They're about as different as Java vs Javascript.
- The moral of the story: Engineers are bad at names

Open MPI: Setup

- `int MPI_Init(int* argc, char*** argv)`
  - Initializes the MPI framework, connects everything together, etc.
  - Should be done at the start of an MPI program
    - Technically a few things are allowed before MPI_Init, but best practice is to do this first.
  - Send in the addresses to argc and argv, though for Open MPI, it doesn't actually use them; this is for compatibility with other MPI systems
- `int MPI_Finalize()`
  - Finalizes the program, cleaning up the MPI framework
  - Should be the last thing done in an MPI program. Must be done by all processes before termination

Open MPI: Process Identification

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Returns the number of MPI nodes in the group and process ID, respectively
  - The first argument can be used if you split up your processes into groups, but we won't go into this.
  - For this class, you can always use the constant `MPI_COMM_WORLD` to get the size of the entire program/process ID relative to all processes
- Note that all of these functions receive as input a pointer which will be used to store the return value. The actual return value is used to specify if the operation worked (0 if success, an error code if failure), so don't conflate the two!

Open MPI: Example


```
int main(int argc, char** argv) {
    if (argc != 2) {
        printf("Usage: %s <foldername>\n", argv[0]);
        return 1;
    }
    MPI_Init(&argc, &argv);
    int processID, clusterSize;
    MPI_Comm_size(MPI_COMM_WORLD, &clusterSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &processID);
    ... //Actual Code
    MPI_Finalize();
}
```

Open MPI: Process Identification

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Returns the number of MPI nodes in the group and process ID, respectively
  - The first argument can be used if you split up your processes into groups, but we won't go into this.
  - For this class, you can always use the constant `MPI_COMM_WORLD` to get the size of the entire program/process ID relative to all processes
- Note that all of these functions receive as input a pointer which will be used to store the return value. The actual return value is used to specify if the operation worked (0 if success, an error code if failure), so don't conflate the two!

Open MPI: Communication

- In any communication in real life, two things need to be true:
  - The sender should be ready to send a message
  - The receiver should be ready to receive a message
- Analogy: Playing catch; if I throw a ball and you're not ready to catch it, the ball will be lost
- The same is true in MPI
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- `int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- In order for a message to be sent, the receiver must call Recv, and the sender must call Send. Once both functions run, the message is sent.

Open MPI: Communication

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- `int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- buf: An array of data to be sent/a buffer to receive data
- datatype: constants used to specify the type of the input (ex. MPI_UINT64_T)
- count: How many elements of datatype to receive
- dest: For Send only, the process ID of the intended recipient of the message
- source: For Recv only, the process ID of the expected sender of the message
- tag: For when you want to further classify messages
  - A Send/Recv pair only "matches" if the tags are the same
- comm: The communication group; for this class, just set it to MPI_COMM_WORLD
- status: For Recv only, will be set to contain the source, tag, and error message of the communication

Open MPI: Communication

- Recv can specify a sender of MPI_ANY_SOURCE and tag of MPI_ANY_TAG to receive from any sender/tag
  - Often useful in manager-worker frameworks
  - Once this type of recv happens, the recipient can then check the status for more info. If the recipient doesn't need the status, you can set that argument to MPI_STATUS_IGNORE to save memory
- By default, Recv and Send are blocking; the process will wait until its partner is ready to communicate
  - This can lead to deadlock; ex. if two processes wait for each other to send a message
  - Can avoid this with IRecv and ISend, which are nonblocking versions of Recv and Send, respectively

Application to Matrix Multiplication

- Normally, a single matrix multiplication won't be easy to multi-process
  - Too much cross-communication, so trying to use MPI will likely just add a lot of file and message operations
- However, really useful if we have many matrix multiplications to do.
- Ex. Let's say we have ~100,000 independent matrix multiplications to do
  - You have 200k files "Task0a.mat, Task0b.mat, Task1a.mat, …" in a folder somewhere, and need to make "Task0ab.mat", "Task1ab.mat", etc.
- How would we parallelize this over 1000 processes?

Multiprocessing ManyMatMul: Naive Approach

Have process 0 do tasks 0-99
Have process 1 do tasks 100-199
…
Have process 999 do tasks 99900-99999

- Any problems with this approach?
- While this will work, it might not load balance well
  - What if the tasks were sorted by size/the last 100 were 1000 times larger than all the other tasks?
- Need some way to dynamically assign work, without tons of communication

MPI Example: The Manager-Worker framework

- A very common framework for MPI programs; fairly simple to implement, while being versatile enough that you can adapt it to new purposes
- Assumes that the problem you're solving can be reduced to a set of independent tasks, that can be done in any order, independent of each other.
- Main idea: Have two roles:
  - Manager, whose job is to assign work and inform the user of progress
  - Worker, who receives work from the manager, and does the work
- Involves writing two versions of the code: one for process 0, and one for all other processes

Manager Pseudocode


```
Set up
While there's work to do:
    Wait until a worker says "I'm ready for more work" (recv from all)
    Find the next task to do
    Send to the worker what task to do
Repeat #Worker times:
    Wait until a worker says "I'm ready for more work" (recv from all)
    Send to the worker "All work done"
Finalize
```

Worker Pseudocode


```
Set up
While True:
    Send to the manager "I'm ready for more work"
    Receive message from manager
    If message is "Here's more work":
        Do the work
    Else if message is "All work done":
        break
Finalize
```

How to send messages?

- Generally want to minimize the size of each message.
- Easiest option: Assign each task/task type a number, and send that number as your message. This is your communication protocol.
  - Ex. -1 means "No more work", 0 means "Do task 0", 1 means "Do task 1", etc.
  - Up to you how you encode this, but make sure you document this somewhere for your sanity
- If some task requires input parameters or returns an output, might run several more rounds of recvs/sends.
- Alternatively, each task's input is from a file, and each task's output is sent to another file. This means you just need to send one number to assign a task, and the worker thread can directly read/write input files.

Multiprocessing ManyMatMul: Manager-Worker Approach

- We do end up "wasting" one process as a manager, but it's generally a good idea to not have the manager do other work
  - If the manager gets stuck with a hard task, ends up stalling all the other workers
- By having only one manager in charge of the big picture, no need to worry about concurrency issues
- Make sure that all processes receive a kill command; otherwise, we get zombie processes
- What if the tasks had some dependencies? (ex. Matmul 100 needs to be done after Matmul 99 and 98)
  - Set up a queue of work that can be done right now, and keep track of how much work needs to be done total
  - If a worker finishes when there's no work to do right now, tell the worker to wait and come back in a few milliseconds.

Multiprocess+Multithreading?

- You can theoretically run a multiprocess program as a multithreaded one without communications
- Generally, lack of communications causes the multiprocess program to be slower/less applicable
- At the same time, multithreaded code is limited to one node, while multiprocess code can be extended indefinitely.
- Can get some improvement by making one process per node, and each process uses #cores/node threads, but this will specialize your code more towards a particular architecture.

Performance Programming Overview

Optimization	Max Speedup	Pros	Cons
Register/Function Inlining	<2x	Easy change, reduces memory accesses	Minimal effect, optimizing compiler might do this already
Loop Unrolling	<2x	Reduces Branching	Minimal effect, significant penalty to maintainability
Cache Optimizations	~10x	Surprisingly good	Often requires algorithmic changes
SIMD	~8x	Fairly applicable, minimal overhead	Limited by hardware, often hit hard by Amdahl's Law
Multithreading/OpenMP	#cores/node	More flexible than SIMD and MPI, generally	Concurrency issues, high overhead
Multiprocess/Open MPI	#cores	Can be extended arbitrarily large	Expensive communication, high overhead