CMSC416: Introduction to Parallel Computing

Topic: **Task-based Programming Models and Charm++**
Date: April 09, 2024

[redacted]

Announcements:
- Extra credit will be discussed w/ TA's
- Project 4 preview is out

Measuring OS Noise:
- Operating system noise also referred to as a "jitter" leads to Performance variability and potential causes of it. This is because when the program is run the OS runs the executable, the user program runs until some OS daemon wants to run. During this your program is either put to sleep or interrupted.
    - Different daemons take different amounts of time and interrupt your program randomly.
- We can measure this noise by doing a simple loop and checking for variability when run several times.
- Quantifying how much noise is happening on the system
    - Can do a fixed amount of work and see if there is a difference in performance by running it a lot of times.
    - Fixed amount of tasks that are run to see what the difference in performance is
- Thread binding:
    - Since threads that are not binded can float around on different cores we won't be able to avoid the core that has more bottlenecks (more daemons on one core)
    - Instead you pin you threads on other nodes of the core

Case of the Missing Supercomputer Performance
- Some cores have really slow performance, on some of the cores there are several OS daemons that are at different granularities.
    - One particular demon wakes up every 30 seconds and runs for 10 ms
    - Different frequency that they wake up and run for a different amount of time

Impact on communication
- If one process gets delayed more than the others, the other process that was waiting for the stalling process gets delayed and this leads to a cascading effect.
- The collectives start to get messed up because of the delays of multiple processes
    - This leads to OS daemons leading to a larger impact than "expected"
        - We cannot "predict" which daemon is being run and leading to an interruption in the program because they are often random and decided by the OS. Some daemons might take 8 ms whereas others take up to 200 ms which once again leads to variability in performance of program.

Mitigating OS Noise

- Running a light-weight OS
    - One potential solution is to turn off unnecessary daemons when working on a full weight linux kernel
- We can also Reduce the frequency of daemons
    - By identifying and understanding the impacting of different daemons we can temporarily disable them for our programs if they are of no use. These changes would have to be reverted back after the computation is complete, especially if we are working in a shred HPC environment.
- Dedicated cores for OS daemons
    - Lot of them wake up on core 0, hence, we can use dedicated cores to reduce the interferences between background processes and high-priority computational tasks.
    - The isolation also leads to more predictable performance, as the computational tasks have consistent access to CPU time and resources.
    - The inverse of this is identifying where the largest concentration of Daemons wake up and ensure that User programs avoid using those cores.

* Network congestion and dedicated file systems are other reasons for performance variability. This will be reviewed in the future

Task-based programming Models & Charm++

- Distributed memory programming model (that can be hybrid)
    - Charm++ can be run in distributed and hybrid mode
    - When running in a distributed environment, Charm++ programs distribute chares across different nodes in the cluster. Each node operates independently with its own memory space, and chares communicate through asynchronous message-passing.
    - In a hybrid model, Charm++ can be used alongside shared memory models like OpenMP. This can exploit multi-threading within each node (using shared memory parallelism) while also taking advantage of distributed computing across nodes. This hybrid approach is useful for leveraging the full potential of HPC systems, which combine multi-core CPUs within nodes (suitable for shared memory models) with a large number of such nodes connected in a cluster (suitable for distributed memory models).

- Belongs to the general category of **task based programming models**
    - MPI → 1 process mapped to a physical core
    - OpenMP → Any # of threads (1 thread) on a single core

    - Describe program/computation in terms of tasks
        - Tasks can be small or large (fine grained coarse grained)
        - The primary work unit in task-based models is the 'task'. A task is a self-contained unit of work, which includes the code to be executed and its associated data.

        - Applications are broken down into multiple such tasks, which can be executed in parallel, taking advantage of the multiple processing units available in HPC systems.
        - A task can be defined as a code region that can be executed concurrently by multiple processes or threads, and alongside other tasks.
        - Tasks are usually designed to be as independent as possible because this allows for concurrent execution without having to synchronize at each step.
    - The advantage of this is that even if you have 16 Physical cores → it still allows you to make 32, 64. 1024 tasks
    - Notable examples (of task-based programming): Charm++, StarPU, HPX. Legion
        - High level idea all of them are very similar
        - higher level or lower level in terms of programming models
    - This model of programming enables exposing a high degree of parallelism → 16 cores does not limit you with only 16 processes
    - *Decoupled* → Number of tasks independent of the number of processors
        - Makes programming more flexible
    - Tasks might be short-lived or persistent throughout program execution
        - Tasks can be killed after working on a code region
    - Runtime handles distribution and scheduling of tasks
        - This concept it common across all the models

Charm++: key principles

- Programmer decomposes data and work into objects (called *chares*)
    - Decoupled from number of processes or cores
    - This encapsulation makes it easier to organize and manage complex computations.
- Runtime assigns objects to physical resources (cores and nodes)
- Each object can only access its own data
    - Request data from other objects via remote method invocation: foo.get_data() rather than sending messages
        - Take two objects A → B
            - A being in node 0
            - B being in node 1
            - They can still communicate
    - Higher level abstraction than MPI
- Asynchronous message-driven execution
    - Chares communicate with each other using asynchronous method invocations, these are non-blocking calls that allow a chare to continue its execution without waiting for the to call to complete. This feature allows for overlapping computation with communication leading to increase in parallel efficiency.

```
mainmodule hello {

  array [1D] Hello {
    entry Hello();
    entry void sayHi();
  };

};
```

- Need a separate *CI file* → tells Charm that there are tasks/objects that are special and need to communicate
  - This file serves as an interface definition for chares. It is a separate file, as previously mentioned, where the programmer declares chares and their *entry methods*.
- Create a 1D array of objects called Hello
  - Specify which functions of this class can be called remotely *entry*
  - Methods or functions that can be called remotely are called *entry methods* in Charm
  - Other objects that you do not want to call remotely can be in you C++ file

```
void Hello ::sayHi() {
  CkPrintf("Hello from chare %d on processor %d.\n", thisIndex,
CkMyPe());
}
```

- Standard C++ file
  - *CkPrintf*
  - *thisIndex*
  - *CkMyPe()*

```
Main::Main(CkArgMsg* msg) {
  numObjects = 5; // number of objects

  CProxy_Hello helloArray =
          CProxy_Hello::ckNew(numObjects);

  helloArray.sayHi();
}
```

- Now we can write main function → in main constructor we can specify the number of objects that we are looking to create
    - CProxy_Hello → how we instantiate a new charm array
    - Once it is instantiated we can start using the array
        - helloArray.sayHi();

Compiling a charm program

- Mpicc is a wrapper, similarly, the charm compiler (*charmc*) is used to translate the hello.ci file to a valid C++ file. This creates the decl.h file and def.h file
    - The .ci file contains teh interface definitions for Charm++ objects including entry methods
    - The *charm_hello.decl.h* is a header file containing declarations of the chare classes, including entry methods
    - The *charm_hello.def.h* is a header file containing definitions that are necessary for the runtime system to handle the objects and messages.
        - charmc hello.ci
        - charmc -c hello.C
        - charmc -o hello hello.o
            - The source code file for the Charm++ program is compiled using charmc with the -c flag to product an object file
                - -c tells charm to compile the source file but not link it
            - After compilation the resulting object file hello.o, needs to be linked to create the executable program.
                - This is done w/o the -c flag
                - The output will be an executable named hello, which can be run on a Charm++ supported parallel machine.

Chare arrays

- User can create indexed collection of data-driven objects
    - Once we have the helloArray handle, it is an easy way to call any task in that array
    - CProxy_Hello helloArray = CProxy_Hello::ckNew(numElements);
        - New chare array named helloArray of type Hello where numElements specifies the number of elements (chares) in the array.
    - numElements is an input via command line arguments
    - Can create different types of arrays 1D, 2D, 3D…
    - The Charm++ runtime system automatically handles the mapping of chare array elements to physical hardware resources such as CPUs or nodes in a distributed system.

Over-decomposition

- Create lots of “small” objects per physical core
    - A large number of “small” objects or tasks are created, more than the number of physical cores available.
    - These objects are usually lightweight in terms of their computation workload and memory footprint.

The objects created are organized into chare arrays, which can have various dimensions as previously mentioned.

- System assigns objects to processors and can migrate objects between physical resources. The Charm++ system dynamically assigns these objects to different processors. This is not a static one-to-one mapping; objects can be moved between processors as needed.
- Facilitates automatic load balancing
    - Over-decomposition inherently facilitates automatic load balancing. The runtime system can balance the workload across the available processors by scheduling the execution of these small objects as resources become available.

Message-driven execution

- An object is scheduled by the runtime scheduler only when a message for it is received
    - If you have multiple objects on a single process, only one process can run at a time, this is done at a first come first serve basis.
    - Charm++ objects and charm++ functions are non preemptible
        - Has to complete execution of the current function and for the scheduler decides what to schedule next
        - Scheduler constantly looks at queue to see what to process next

Cost of creating more objects

- Context switch overhead
    - When a system handles many small tasks, it needs to frequently switch context to another task. This involves saving the state of the task and loading the state of another, this can introduce delays and affect overall performance of the program.
- Cache performance
    - Get cache benefits w/o doing anything extra
    - Each small object can have its own set of data due to the encapsulation of chares, hence, this could lead to more frequent cache misses if the data does not remain in the cache between accesses.
- Memory overhead
    - Working with more objects means more overhead for the memory system.
    - The increased memory footprint can lead to higher memory usage which could cause problems in memory-constrained environments.
- Fine-grained messages
    - Smaller objects may communicate through fine-grained messages, which could lead to an increase in the overhead of communication.

MPI is more coarse grained
Task based programming model is more fine grained

- Might get better cache performance