CMSC416: Introduction to Parallel Computing

Topic: OS Noise and Charm++
Date: April 09, 2024

Measuring OS Noise

- One of the talked about causes was “jitter” which happens because of the OS, especially if it is full weight
    - This is because they have a lot of services or daemons that have to wake up and do something
    - This can sometimes interrupt the user program or kernel
    - Can quantify the amount of noise happening, which can be done through Fixed Work Quanta or Fixed Time Quanta
- Fixed Work Quanta
    - If you run the same benchmark amount of work across a supercomputer, we can see how much variability there is in the execution time
- Case of the Missing Supercomputer Performance
    - Looked at the problem of performance variability, and what they noticed was that on some cores, performance was extremely slow because of OS noise
        - There are several OS daemons that are woken up and they are of different granularities
        - For example, one daemon is woken up every 30 ms and is awake for 10 ms
        - Every daemon has a different frequency at which they are woken up and also how long they stay awake for
- Assume we have a bulk synchronous program (stencil computation, Game of Life, etc.) where each process is sending and receiving messages
    - Assume that at some point, some OS daemons wake up, which ends up in one process being delayed more than everyone else
    - This also slows down everyone else because there is some process that is waiting for communication, which causes a ripple effect
    - This can especially happen if you have things like collectives or MPI_Barrier
    - Process compute variability can cascade into variability into communication
- How to mitigate OS noise?
    - Running a light-weight OS – usually involves turning off unnecessary daemons
    - Turn off unnecessary daemons
    - Reduce the frequency of daemons
    - Dedicated cores for OS daemons – waking them all up on core 0 is common
        - This allows the app to use cores from 1-31, 33-63 (if we have 64 cores)
    - User programs can avoid using certain cores
    - Some of these solutions can be combined with one another
- Second cause is network congestion
    - Although each job has access to the network, the networks are shared
    - Recall the old topologies – dragonfly, etc.

Charm++ –

- Distributed memory model, but in some ways you can call it a hybrid one
    - Can run it in distributed memory as well as shared
- Task-based programming models
    - We have usually thought of tasks or processes
    - When we looked at OpenMP, although we didn't have to think about threads, we still have to specify the number of threads through OMP_NUM_THREADS
    - In MPI, you have 1 process mapped to 1 physical core
    - In OpenMP, you have 1 thread mapped to 1 physical core
        - You can technically assign more threads to a core if you'd like
    - In task-based programming model, we delineate the concept of how a programmer codes and how the units are actually mapped to the physical machine
        - Programmer breaks down their program into tasks
        - Run-time is the one that decides the mapping of tasks to physical cores
        - Ex. even if you have 16 physical cores, you can have 16, 32, 64, etc. tasks which has many benefits
    - What is a task? – It is a code region that is executed concurrently by multiple processes or threads
    - Examples include Charm++, StarPU, HPX, Legion
    - Charm++ is an older example which was developed at the University of Illinois - Urbana-Champaign
    - All of the examples are on various levels of abstraction – some are very low-level and some let the run-time decide more of what is going on
    - Enable exposing a high degree of parallelism – reference the example of having less cores than tasks
        - Example of why it is useful to decouple: LULESH – could only run in cubic numbers (1, 8, 27) due to some physical restrictions because of 3D computation
    - Number of tasks independent of the number of processors – can make programming more flexible as you don't have to program according to some physical restrictions
    - Tasks might be short-lived or persistent throughout program execution
    - Runtime handles distribution and scheduling of tasks
- Key Principles of Charm++
    - A lot of these principles will apply to the other examples
    - Programmer decomposes data and work into objects (called chares)
        - Number of objects are arbitrary – can be a command-line argument
        - Decoupled from number of processes or cores
    - Each object can only access its own data, which is where it gets some similarities to MPI
        - Can only access their own address space (variables, memory, etc.)
        - Have to request data from other objects explicitly

        - However, don't need message passing, but can do remote method invocation
            - Example: if you have a function called foo, can call foo.get_data()
            - Technically this is a higher-level abstraction by Charm++
            - Since it is a remote method invocation, the two processes do not have to be on the same core or node
    - Asynchronous message-driven execution
        - Huge benefit of Charm++
        - When you call a function on another core, this is not blocking
            - It just means that you're asking the other process to look into the function call and it happens asynchronously
            - Not guaranteed as to when they are going to be executed

```
mainmodule hello {

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

};
```

- This is the CI file
- To tell the runtime that you can call remotely, add the keyword "entry"
- Other functions that you don't want to call remotely, just use normal C++ static

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

- This is the normal C++ file – from each chare we want to print a number and the processor
- Similar to MPI, still going to have to create processes, but the number of chares is independent of the number of processes

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

  CProxy_Hello helloArray =
          CProxy_Hello::ckNew(numObjects);

  helloArray.sayHi();
}
```

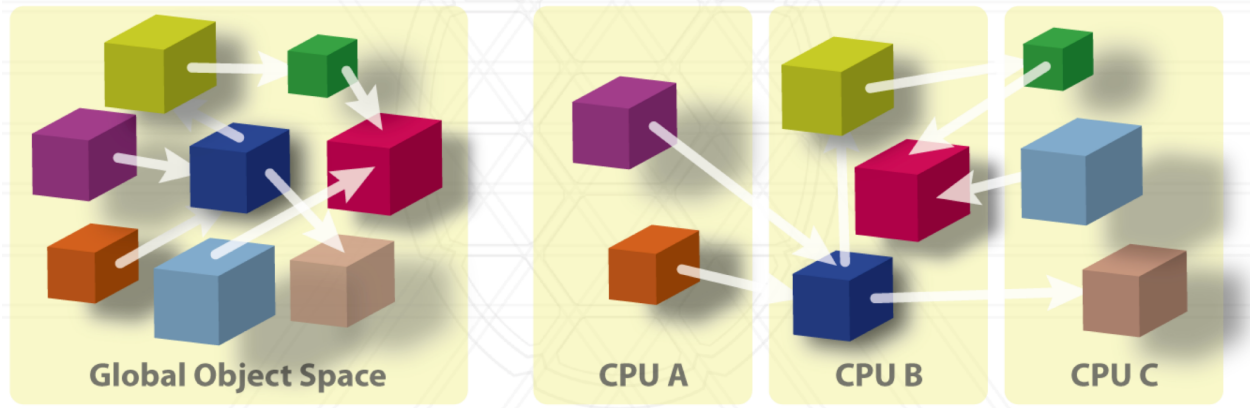
- Let's say we want to create an array of 5 chares
- CProxy_Hello is how you can instantiate a new array with the number of arrays
- The proxy is like a handle you can use to access different elements, such as how `helloArray.sayHi()` is being called.
    - When you're not giving a specific index, all of the objects are going to be called and they are going to print in a random order
- This is letting you run programs across processes and cores

Compiling a program:

- Use `charmc`: for example, use `charmc hello.ci`
- Charm translator for .ci files generates a .decl.h and .def.h file
- Then, it's like compiling a normal C++ file
- Have to include .decl.h fie at the top of the **.h file** and the .def.h file at the bottom of the **.C file**

Chare arrays:

- Can create independent tasks or an indexed collection of tasks
- There are two parts, which is in the CI file you have to mention that you're going to create it, and then the second is the ability to actually translate it
- `CProxy_Hello helloArray = CProxy_Hello::ckNew(numElements);` is an example, where numElements is the amount of elements you want
    - numElements isn't actually pointers to those objects
    - They are just handles that allow you to access the information, and we don't need to worry about the actual underlying implementation
- Can create up to a 6D array
- Mapping of array elements (objects) is handled by the runtime (sometimes abbreviated by RTS)
- Default scheme is a load balancing one – try to assign an equal amount of chares to each core



User View System View

- The arrows in the User View aren't messages, they are just the remote method invocation
- They are called asynchronous because you don't need a handshake for a 2-way data transaction – can just send it over and not have to worry about it
- Runtime will assign the various objects to CPU A, B, and C

Over-decomposition:

- Create lots of "small" objects per physical core
    - Objects grouped into arrays: 1D, 2D, …

Message-driven execution:

- An object is scheduled by the runtime scheduler only when a message for it is received
- Facilitates adaptive overlap of computation and communication
- There is a Charm++ RTS under each processor
- The scheduler is the one which determines which job will be scheduled, which happens on a first-come first-serve basis
- Once a job is done, then the next job is gone
- They are non-preemptable from the POV of the RTS – of course the OS can interrupt though
- If a chare sends a message to another, then it is added to the message queue, which is handled by the runtime
    - Looks at it in a FIFO order, and handles the messages by calling whatever needs to be called according to the chare's message
    - Sending the message from one process to another, adding to the queue, constantly looking at the queue, is all done by the runtime

Cost of creating more objects?

- MPI is more coarse-grained
- More objects means smaller blocks which means better cache performance
    - Automatically they might fit into the cache without doing anything extra, which is contrasting with MPI

- Memory overhead – might go up by a small fraction (maybe 10-15% based on how big your program is)