CMSC 416 Parallel Computing
April 9th, 2024

Topics: OS Noise, Task-based programming and Charm++

OS Noise

OS noise refers to fluctuations in the performance of an operating system (OS) that can impact the efficiency of parallel computing systems.

Measuring OS Noise

Measuring OS noise involves assessing factors like Fixed Work Quanta (FWQ), where tasks run for a fixed workload, and Fixed Time Quanta (FTQ), where tasks run for a specific time duration. In the case of missing supercomputer performance, various factors contribute to OS noise. Some processor cores may operate slower than others due to different daemon activities, which are background processes with varying wake-up frequencies. This can affect communication between processes, leading to significant delays for certain tasks. Compute variability issues can cascade into communication delays, which can delay program execution by huge amounts.

Mitigating OS Noise

Several strategies can mitigate OS noise in parallel computing environments. Implementing the following mitigation strategies can reduce the effects of OS noise when running parallel programs.

1. *Using Lightweight OS*: Employing a lightweight operating system reduces the overhead with parallel tasks.
2. *Turning off Unnecessary Daemons*: Turning off unnecessary background processes minimizes OS activity and reduces interference with compute tasks.
3. *Reducing Daemon Frequencies*: Lowering the frequency of daemon activities decreases their impact on parallel computations.
4. *Dedicating Cores to OS Daemons*: Allocating specific processor cores exclusively for OS tasks ensures smoother parallel processing. For example, in a system with 64 cores, dedicating cores 0 and 32 for only OS operations leaves the remaining cores (1-31 and 33-64) available for user programs.
5. *Core Allocation to OS*: User programs can optimize performance by avoiding specific cores affected by OS noise, which improves parallel processing efficiency.

Task-based Programming Models and Charm++

In various task-based programming models, such as Charm++, StarPU, HPX, and Legion, programs or computations are described in terms of tasks. A task can be defined as a piece of code that can be executed concurrently by multiple processes, potentially alongside other tasks. These tasks can vary in size, being either large or small, and range from fine-grained to coarse-grained.

Charm++

Charm++ (developed at UIUC) is an example of this approach. It enables programmers to define tasks and assign them to cores or nodes. Tasks can be short-lived or persistent throughout program execution, and the runtime system manages their scheduling and distribution efficiently.

Charm++ Key Principles

1. *Decomposition into Chares*: Programmers decompose both data and work into objects, known as chares. Charm++ extends C++ to facilitate this, allowing programmers to create C++ objects.
2. *Assigning Objects to Physical Resources*: The runtime system of Charm++ assigns objects to physical resources such as cores and nodes, handling distribution and scheduling of tasks. Note that blocking calls should not be assumed.
3. *Object has Access Only to its Own Data*: If data is needed from some other object, it must be requested explicitly via remote method invocation (foo.get_data()).
4. *Asynchronous Message-driven Execution*: With message-driven execution, Objects communicate with each other via remote method invocation.

Creating a Charm++ “Hello World” Program

To create a "Hello World" program in Charm++, you work with a Charm Interface (CI) file and a C++ file. The task of printing "hello world" is divided into tasks within the C++ file. When using Charm++, a wrapper is used. This involves a Charm compiler to compile the interface file, which is then converted into valid C++ code. The compilation process generates files named charm_hello.decl.h and charm_hello.def.h, which need to be included at the bottom of your C++ files. Following this, you compile your C++ code and link it to an executable.

Charm++ also supports the creation of chare arrays, allowing users to generate indexed collections of data-driven objects. These arrays can be structured in various dimensions, such as 1D, 2D, or 3D. The mapping of array elements (objects) to hardware resources is managed by the runtime system (RTS), which aims to load balance evenly across physical cores.

Over-decomposition in Charm++

Over-decomposition in Charm++ involves the creation of numerous small objects per physical core, often organized into arrays spanning one, two, or three dimensions. The runtime system handles the allocation of objects to processors and can communicate data between physical resources (cores), enabling automatic load balancing.

Message-driven execution

Message-driven execution in Charm++ operates on the principle that an object is scheduled for execution by the runtime scheduler only upon receiving a relevant message. This message-driven approach ensures that objects are processed first-come, first-served. Once an object completes its task, it is removed from the execution queue, allowing another object to commence processing.

Cost of creating additional objects

The cost of creating additional objects in Charm++ includes factors such as context switch overhead, impacts on cache performance, and memory overhead. Charm++ allows for experimentation with different grain sizes and determining the appropriate grain size for objects. For example, you can use Charm++ for testing performance on 16, 64, or 128 tasks on 16 processes. This distinguishes Charm++ from MPI, which lacks these testing options, as it is limited to running 16 tasks on 16 processes.