Reactive Task-parallel Programming

Christian Terboven <terboven@itc.rwth-aachen.de>

November 1st 2018, Tsukuba
**The future is dynamic**

- Dynamic Variability in HPC systems continues to increase
  - Feature processors (example: Intel Turbo)
  - Energy Management (example: Power Capping)
  - Detection and Correction of Errors

---

### Mitigating Processor Variations through Dynamic Load Balancing

Bilge Acan, Lasmikantz V. Kale
University of Illinois at Urbana-Champaign, Department of Computer Science
{acan2, kale}@illinois.edu

—There can be performance variation among same-sensor in large scale clusters, and supercomputers used by power, and temperature variations among cores. These variations manifest itself as frequency diff- the processors under dynamic overclocking, such as, different, multi-node processors also create an inherent who used in the system. For some tightly coupled systems even one slow processor in the critical path can slow the whole application therefore this variation is a problem. To mitigate the performance variation sensors, we propose a speed-aware dynamic load balancer which works on both homogeneous and hetero-heterogeneous hardware. Our main idea is to provide an of the task completion time based when moving a one processor to another on the processor speed. We a JVP performance improvement using our speed-aware balancer compared to the no load balancing case, how that our speed-aware balancer performs 3% a non-speed aware counterpart.

---

### Variation Among Processors Under Turbo Boost in HPC Systems

Bilge Acan, Phil Miller, Lasmikantz V. Kale
Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801
{acan2, miller121, kale}@illinois.edu

---

### Motivation

Dynamic Variability in HPC systems continues to increase due to dynamic characteristics such as: processor features (example: Intel Turbo), Energy Management (example: Power Capping), and Detection and Correction of Errors. These characteristics manifest themselves in the form of frequency differences among processors under Turbo Boost dynamic overclocking. This variation can lead to suboptimal and suboptimal parallel performance which is tightly coupled HPC applications. In this study, we explore compute-intensive benchmarks and application to analyze the variation among processors in four top supercomputers: Edison, Oak Ridge, and Blue Waters. We observe that there is an execution time difference of up to 16% among processors in the TurboBoost enabled supercomputers: Edison, Oak Ridge, and Blue Waters. This is due to the dynamic overclocking nature. We measure microbenchmarks from temperature and power to understand and find that the differences in the chips' power efficiency is the subject behind the frequency variation. Moreover, we analyze potential solutions such as disabling Turbo Boost, leaving idle cores and replacing slow chips to mitigate the variations. We also propose a speed-aware dynamic task redistribution (load balancing) algorithm to mitigate the negative effects of performance variation. Our speed-aware load balancing algorithm improves the performance up to 15% compared to no load balancing performance and 6% better than the non-speed aware counterpart.

---

### Algorithmic Efficiency

For years, CPUs were designed to automate parallelism. The voltage and clock frequency are increased (dynamic power management). The reduction in the high-performance computer was frequently done by increasing speed without slowing down or even drive up to a stop. This reported that, of our own future, Dynamic power management is everywhere, from cars and mobile phones to home heating and cooling. One of the key technologies changing dynamic power management. This is the shift to fly-by-wire control. The drivers of this trend are: the increase in power and temperature of the chips. The voltage and clock frequency increases among processors under Turbo Boost dynamic overclocking. This variation can lead to suboptimal and suboptimal parallel performance which is tightly coupled HPC applications. In this study, we explore compute-intensive benchmarks and application to analyze the variation among processors in four top supercomputers: Edison, Oak Ridge, and Blue Waters. We observe that there is an execution time difference of up to 16% among processors in the TurboBoost enabled supercomputers: Edison, Oak Ridge, and Blue Waters. This is due to the dynamic overclocking nature. We measure microbenchmarks from temperature and power to understand and find that the differences in the chips' power efficiency is the subject behind the frequency variation. Moreover, we analyze potential solutions such as disabling Turbo Boost, leaving idle cores and replacing slow chips to mitigate the variations. We also propose a speed-aware dynamic task redistribution (load balancing) algorithm to mitigate the negative effects of performance variation. Our speed-aware load balancing algorithm improves the performance up to 15% compared to no load balancing performance and 6% better than the non-speed aware counterpart.

---

### Abstract

The design and manufacture of present-day CPUs causes fluctuations in supercomputer architectures such as variation in power and temperature of the chips. The voltage and clock frequency increases among processors under Turbo Boost dynamic overclocking. This variation can lead to suboptimal and suboptimal parallel performance which is tightly coupled HPC applications. In this study, we explore compute-intensive benchmarks and application to analyze the variation among processors in four top supercomputers: Edison, Oak Ridge, and Blue Waters. We observe that there is an execution time difference of up to 16% among processors in the TurboBoost enabled supercomputers: Edison, Oak Ridge, and Blue Waters. This is due to the dynamic overclocking nature. We measure microbenchmarks from temperature and power to understand and find that the differences in the chips' power efficiency is the subject behind the frequency variation. Moreover, we analyze potential solutions such as disabling Turbo Boost, leaving idle cores and replacing slow chips to mitigate the variations. We also propose a speed-aware dynamic task redistribution (load balancing) algorithm to mitigate the negative effects of performance variation. Our speed-aware load balancing algorithm improves the performance up to 15% compared to no load balancing performance and 6% better than the non-speed aware counterpart.

---

### Our Dynamic Future

Pete Beckman | Argonne National Laboratory and Northwestern University

Last month, as I tossed my bags in a rental car at the airport, I noticed that the car was particularly new. I was quite surprised, however, when I drove up to the first stop sign, and the car suddenly died. It was as if I had run out of gas or turned off the ignition. However, as soon as I took my foot off the brake pedal, the engine started itself back up. I pushed on the accelerator, and the car jumped forward. Over the next couple of days, I explored this advanced fuel-saving feature, trying to understand under what circumstances the car’s algorithms would decide it could save gas by temporarily shutting off and how quickly I could jump forward after moving my foot from brake to accelerator as the car automatically started itself and slowly adjusted the throttle.

Dynamic power management is everywhere, from cars and mobile phones to home heating and cooling. One of the key technologies changing dynamic power management. This is the shift to fly-by-wire control. The drivers of this trend are: the increase in power and temperature of the chips. The voltage and clock frequency increases among processors under Turbo Boost dynamic overclocking. This variation can lead to suboptimal and suboptimal parallel performance which is tightly coupled HPC applications. In this study, we explore compute-intensive benchmarks and application to analyze the variation among processors in four top supercomputers: Edison, Oak Ridge, and Blue Waters. We observe that there is an execution time difference of up to 16% among processors in the TurboBoost enabled supercomputers: Edison, Oak Ridge, and Blue Waters. This is due to the dynamic overclocking nature. We measure microbenchmarks from temperature and power to understand and find that the differences in the chips' power efficiency is the subject behind the frequency variation. Moreover, we analyze potential solutions such as disabling Turbo Boost, leaving idle cores and replacing slow chips to mitigate the variations. We also propose a speed-aware dynamic task redistribution (load balancing) algorithm to mitigate the negative effects of performance variation. Our speed-aware load balancing algorithm improves the performance up to 15% compared to no load balancing performance and 6% better than the non-speed aware counterpart.
Taking to the rescue

- Tasking is well-positioned to react to dynamic system behavior
  - Less global synchronization
  - More p2p synchronization

Image Source: John Shalf
Image Source: Jack Dongarra
Agenda

• Task Affinity

• Inter-node Tasking

• Conclusions
Task Affinity

OpenMP 5.0
Feature design

Fundamental questions

- Task affinity in the context of task stealing
  - Should our proposal be prescriptive or descriptive?
    - The latter
    - User may assume that task affinity hints bias task stealing

- Task affinity targets
  - Task-to-thread affinity resembles data-to-thread on NUMA
  - Task-to-data affinity resembles dataflow model
    - When in doubt, do both

- Task-to-thread affinity
  - Does the user want to deal with thread ids? No.
  - Are places better? Maybe, but the place list is static.
  - The thread affinity policies worked well
    - Adoption for task affinity with taskgroup
    - Also applicable to taskloop

- Task-to-data affinity
  - Affinity to allocation place or most recently used place?
    - Choice is application-dependent
    - When in doubt, do both

- New constructs, new clause, or extension of existing clause?
  - New construct not necessary.
  - depend clause could have been used, but that would mix concepts
    - We propose a new clause
Reactive Task-parallel Programming

Support for task affinity is part of OpenMP 5.0, to be released on November 8th, 2019

```
#pragma omp task [clause...] affinity(list)
```

```c
int a[N]; // N is large
...
#pragma omp task affinity(a[x-y])
{
    // task that makes use of a[x], ...
}
```

- Programmer specifies data used by task
- Recommended to execute task closely to data location
  - Do not prohibit task stealing & load balancing
- Runtime identifies the location of the data and schedules task to a close thread
- Clear separation between dependencies and affinity
Selected implementation details

- A map is introduced to store location information of data that was previously used.

1. Encounter task region ...
2. Task with data affinity?
   - Yes
     - Location for data reference in map?
       - Yes
         - Identify NUMA domain where data is stored
         - Select thread pinned to NUMA domain
         - Save {reference, location} in map
         - Push task into other threads queue
         - Push to local queue
         - end
       - No
         - Push to local queue
   - No
     - Yes
       - Push to local queue
     - No
       - Location for data reference in map?
         - Yes
           - Identify NUMA domain where data is stored
           - Select thread pinned to NUMA domain
           - Save {reference, location} in map
           - Push task into other threads queue
           - end
         - No
           - Push to local queue
Selected implementation details


Encounter task region …

Task with data affinity?

No

Push to local queue

Yes

Location for data reference in map?

Interesting parts

No

Identify NUMA domain where data is stored

Save \{reference, location\} in map

Push task into other threads queue

Select thread pinned to NUMA domain

end
Evaluation with STREAM

How close can tasking come to the optimal execution?

- Single creator: one thread creates all tasks
- Parallel creator: tasks are created in parallel by all threads

**Intel® Xeon® E7-8860v4 (codename Broadwell)**
- 8 sockets, 18 cores per socket = 144 cores
- 2.2 GHz base frequency
- 1 TB memory
Evaluation with Applications

How much can this improve applications?

- Little improvements on standard 2-socket systems, more improvement on larger systems

Works well working with a lot of data & single task creator scenarios & tasks created in parallel but not all close to data

Not much room for improvement when: parallel task creator scenarios & tasks are already created where data is located

Intel® Xeon® E7-8860v4 (codename Broadwell)
8 sockets, 18 cores per socket = 144 cores
2.2 GHz base frequency
1 TB memory
Inter-node Tasking
Early results from our CHAMELEON project
Strategy to tackle load imbalance

- Shared Memory: over-decomposition
- Distributed Memory: over-decomposition & re-distribution of data

**Tasked-based Execution Environment**
- Allows task migration for load balancing between nodes

**Performance-) Introspection**
- Continuous monitoring the current process
- Determine runtime conditions and performance metrics

**Analysis and Consolidation**
- Consolidates information from processes
- Decision making
  - Whether to migrate tasks
  - Maybe also victim selection
Feature design / 2

General idea

1) Detect imbalance dynamically at runtime based on available information collected by introspection

Possible result:
Rank 0 detects that it has more work or is significantly slower

2) Rank 0 migrates task and data to Rank 1

3) Rank 1 executes + sends back results / output data (just if necessary)

- Desired to detect these situations as soon as possible to overlap communication and computation
- It is more like speculative migration instead of task stealing
Based on OpenMP concepts

- OpenMP target: offloading

```c
#pragma omp target
{
    // target region

    #pragma omp task
    {
        // task region
    }
}
```

Diagram:

```
Host thread ──> Target

Device thread(s)
```
Dynamic variability caused by application

- Showcase application: $\text{sam(oa)}^2$
  - Finite-Element and Finite-Volume simulations of dynamic adaptive meshes
  - Space Filling Curves (SFC) and Adaptive Meshes for
    Oceanic And Other Applications (Tohoku Tsunami 2011)
  - Developed at TU Munich

- Depending on situation either refinement or coarsening of cell / section

- Refinement leads to load imbalances
  - within a node
  - between nodes
Solution sketch

```c
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < N; i++)
    {
        // create task + mark as "offloadable"
        #pragma omp target map(tofrom: ...) nowait
        {
            // specify work here
        }
    }

    // where the magic happens
    int result = chameleon_distributed_taskwait();
}
```

Chameleon Lib
- Register task and data used by task and queue it
- Spawn communication threads
- Start executing queued tasks
- Finished when all tasks and outstanding communication done

Working Solution
Invitation for collaboration and exchange of ideas

• Our current (research) questions regarding this approach:
  – In which situation should we offload?
  – Which information do we need as base for decision making?
  – What should be the target rank?
  – When do we stop offloading?

• Jannis is currently doing an internship at RIKEN
Conclusions
Reactive Task-parallel Programming

Tasking model is becoming more attractive

• Many still consider Tasking to be complex

• Tasking brings advantages for dynamic systems

• Recent extension of features
  – Reductions, complex dependencies, more schedule control, …

• Affinity brings performance improvements
  – Including support for complex memory hierarchies

Invitation to collaborate

• Future research direction: runtime work for intra-node and inter-node tasking
• While he is around in Japan: talk to Jannis 😊
Vielen Dank
für Ihre Aufmerksamkeit