# 15618 Project Report: Cache Hierarchy-Aware Task Scheduler on Multicore Architectures

**Team members: Calvin Lin, Yen-Shuo Su**

## URL

https://github.com/calvinfornialin/CMU-15618-Final-Project.git

## Summary


- This project aims to design and implement a cache hierarchy-aware task scheduling system for multicore architectures. The goal is to optimize task execution on parallel systems by leveraging knowledge of the cache hierarchy to reduce cache misses and improve cache utilization and overall system performance.
- The scheduling mechanism is aimed to have the ability to adapt to various different kinds of cache hierarchies within systems and achieve a boost in performance compared to the scheduling approaches implemented within OpenMP, mainly static, dynamic scheduling schemes. Various types of cache hierarchies are shown in the figure below.

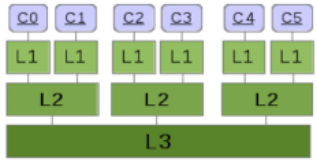




Fig. 1. Intel Dunnington

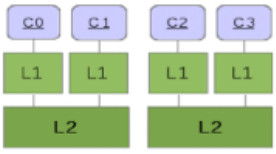



Fig. 2. Intel "Hapertown"

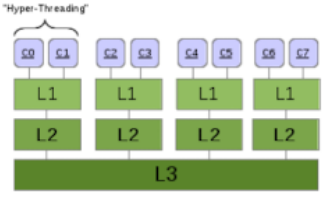



Fig. 3. Intel "Nehalem"

## Background

- Many modern applications, particularly those in scientific computing, data analytics, and machine learning, are compute-intensive and can benefit significantly from parallel execution. As learned in lectures, optimal performance on multicore systems requires careful consideration of how tasks are scheduled among multiple cores or processors, several crucial factors that we should consider are data locality, workload balance, and synchronization/communication time. However, the static or dynamic scheduling approaches introduced in lectures have neglected the cache hierarchy factor. So in this project we will propose a novel scheduling approach that takes not only data locality and workload balance but also cache hierarchy into account and focus on a compute-intensive component that involves frequent data accesses where parallelism can be exploited through task partitioning and scheduling based on cache topology and behavior.
- In OpenMP, the type of scheduling scheme that is utilized is basically one of three kinds, static, dynamic, and guided, which is pre-specified in the program during compile time, or is determined by the system/compiler through the auto scheduling type or is deferred until runtime through the runtime scheduling type. No matter which scheduling type is chosen, the scheduler follows the same scheme throughout the parallelization, which may pose a restriction on more complicated tasks that can benefit from the switching of scheduling schemes instead of following a particular scheduling scheme from the beginning all the way until the end.

# Approach

## Basics

- We're implementing our scheduler in OpenMP API and evaluating it on our own local machine (MacBook w/ M1 Pro chip), GHC machine, and PSC machine.
- Understanding the count of cache levels and the degrees of cache sharing at each level is crucial for informing our scheduling policy. Variations of sharing degree at different levels force programmer to make explicit and architecture-specific program optimization in order to get efficient execution. To achieve efficient performance across diverse architectures with varying cache topologies, a cache-aware scheduling algorithm must dynamically adapt to the target architecture. Therefore, such an algorithm requires a thorough understanding of the cache topology of the underlying machine, which can be acquired through dynamic exploration of the target platform during runtime initialization. Modern operating systems offer mechanisms to access cache hierarchy details at a high level, either through system files like in Linux OS or through native APIs like Windows. However, in our approach, we simply look up the spec of the corresponding running machine and make it an input argument to the program.
- Variation of the cache level count and cache sharing degrees raises the need to unify them under a higher abstract description. In our approach, we extract out the important metrics of cache hierarchy including cache levels, connectivity which can be described as the table below and gives an example of the three different platforms we are targeting. The two first columns gives the cache-levels and cores count, the following columns gives the count of cores sharing Li caches or Mi memory.

| Microarchitecture | #Cache levels | #Cores | L1 | L2 | L3 | M4 |
|---|---|---|---|---|---|---|
| Apple M1 Pro | 3 | 8 | 1 | 4 | 8 | - |
| Intel i7-9700 | 3 | 8 | 1 | 1 | 8 | - |
| AMD EPYC-7742 | 4 | 128 | 1 | 1 | 4 | 128 |

## Cache Hierarchy Aware Task Scheduling

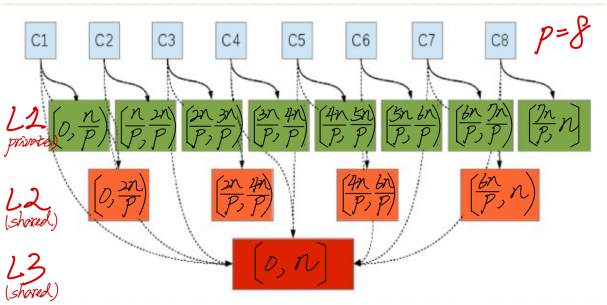
### Parallelize the For Loop based on Cache Topology

- Let's consider a for loop that has a range from i = 0 to i = n:
  - F corresponds to a "Range" which can be partitioned into N "Range" and M "Shared Range", which are both determined directly by the underlying architecture:
    - N = Cores count
    - M = Number of shared caches at all levels (consecutive cache levels shared by the same cores are considered as one)
    - P = N+M, total partition count
- Take the below cache topology as an example, it has 8 cores and a total of 5 shared caches at all levels. So P equals 13 in this example.
  - Green block indicates the "**private range (cache)**" or non-sharing work queue (static scheduling) that is exclusive to a single core. So in the green block (corresponds to a private cache), working set within the range will be stored in that cache.
  - Orange block indicates the "**shared range (cache)"** or sharing work queue that allows task stealing (dynamic scheduling) that is shared by several cores. So in the orange block (corresponds to a shared cache), working set within the range will be stored in that cache.
  - Red block also indicates the "**shared range (cache)"** or sharing work queue that allows task stealing (dynamic scheduling) that is shared by all of the cores. So in the red block (corresponds to a shared LLC), working set within the range will be

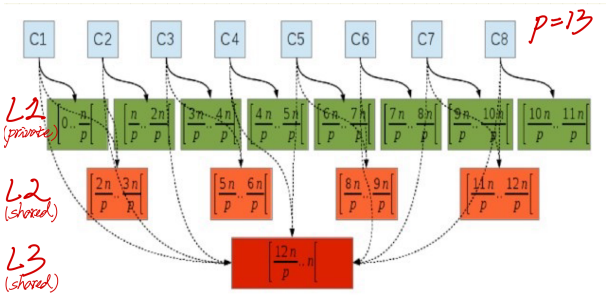
stored in that cache.

## Comparison between static and our scheduler

- Static




- Our scheduler




- We can observe from the above graphs that in the static scheduling scheme,
- There are several reasons why cache hierarchy-aware scheduler can improve performance:
  - Comparing to static scheduling, where the threads have to wait for all other threads to finish in order to finish the task, cache hierarchy-aware scheduler provides the flexibility for threads that are done with their statically assigned tasks to work on other tasks that are reserved for dynamic scheduling. This can better utilize the workers, especially in scenarios where the task workloads are very arbitrary and of large variance between iterations.
  - Comparing to dynamic scheduling which utilize a common work queue for every threads, cache hierarchy-aware scheduler ensures that the working sets (iterations) of a core it is responsible for will always be in its own private cache or the shared cache below it. This can restrict the threads to always work on tasks that are closer to each other and can prevent arbitrary access of the tasks as may possibly happen in a dynamic scheduling protocol.

## Runtime System

- Our initial approach is to read through the OpenMP runtime library and modify the runtime code to enable the scheduler to allocate task based on the retrieved cache topology data. However, it's a bit too complicated to read through the whole OpenMP package and grasp a whole concept, then modify the necessary part and integrate it with the rest of the system.
- So, we impose a layer above the OpenMP directives to leverage the existing static and dynamic scheduler. We schedule the specific parallel loop range to corresponding core conditionally to achieve the goal of above.

## Initial Cache Hierarchy-Aware Task Scheduler (CHATS) Implementation

- Through thorough understanding of the OpenMP API, a big challenge is that the dynamic and static scheduling provided by OpenMP does not provide the capability of controlling the set of thread IDs that are used to run a certain parallelizes block. The first few versions of the implementation included bugs of multiple threads running the same iterations or some iterations being skipped. One of the first trials included using an outer condition loop to control whether or not a thread ID with run the parallel region, but it is soon discovered that this does not yield the intended implementation since the scheduling mechanism still distributes the tasks to all of the workers.
- A work around to solve the previous problem encountered is to utilize the teams option offered by OpenMP. By defining teams based on the second layer of the shared cache, the tasks can be distributed to the each team, which represents the set of threads that share the same L2 cache. The L1 cache concept is then imitated through static scheduling within each team. Finally the tasks that reside in the L3 cache are conducted dynamically by all workers in an independent parallelized region that follows. Some of the problems with this implementation is that we have no control over which threads reside in which team, the partition of the teams is decided by OpenMP, and we can only be sure that the number of threads within each team are the same as our topology. Another problem is that the last dynamic scheduled region corresponding to data in L3 cache can only be done individually in a separate

parallelized region, which means that no threads can start working on the tasks until all tasks up to the L2 cache layer are finished, which is a bit different from our intention and can increase the idle time of some threads.

- Though a bit different from our topology, this is the first huge step that we made on our CHATS implementation. Thus, we provide experimental results of this CHATS implementation using the evaluation tasks and test the performance of this method against the static, dynamic, and our final implementation.

### Refined Cache Hierarchy-Aware Task Scheduler (CHATS-refined) Implementation

- The refined version of CHATS (CHATS-refined) aims to solve the problems addressed in the initial implementation. There is no problem with the static scheduling part of the tasks residing in the L1 cache. By using static scheduling along with the nowait option provided by OpenMP, threads that are finished with the tasks statically assigned to them can move on to latter parts of the code, having the flexibility to work on their responsible set of tasks that are dynamically assigned to them without having to wait for the other threads to finish on their statically assigned tasks.
- As for the dynamic parts, in order to achieve the control of the responsibility of the threads on different dynamically scheduled blocks, we utilize an array to keep track of which iterations have already been done and the atomic exchange operation. Each thread iterates through the set of tasks they may be dynamically scheduled. Through the atomic exchange operation, it can be made sure that no two threads will work on the same iteration. If the iteration has already been done by another thread, the current thread will just simply skip it. The critical region is only on the read and write of one certain element in the array, and threads do not have to wait for another thread to finish the task corresponding to a certain iteration before they can access the element of the array that is used to indicate whether that iteration has been done or not. Each thread will first mark an iteration as done before it starts working on it to minimize the time wasted waiting on the critical section.

# Performance Evaluation

- We compare our implementation to several common schedulers including static scheduling and dynamic scheduling.
    - Static scheduling: Static scheduling is the most straightforward scheduling technique: data is statically partitioned into N equal chunks, these chunks are then processed respectively by N parallel threads. This scheduling scheme avoid communication between threads, offer good data locality when the parallel loop is executed several time. However, this method may result in load unbalancing, especially in the case of heavy workload, since faster threads remains idle, waiting for other threads until finishing their work.
    - Dynamic scheduling: Dynamic scheduling provide better load balancing since threads does not remains idle as long as chunks are available in the common work queue. Unfortunately, while improving workload distribution, this technique may introduces a costly communication between threads accessing concurrently to the common work queue. This may results into ineffective uses of processor caches. Also, this technique provide poor data reuse since a same chunks may be processed by different cores when the parallel loop is executed multiple time. Bad data reuse may amplify consequently cache-miss rate.
- So our implementation kind of combines static and dynamic scheduling. We’re trying to leverage advantages of both scheduling scheme, aiming to maintain data locality while also incorporating the dynamic allocation benefits to prevent idle processor time and improve overall load balancing. By smartly assigning tasks based on dynamic and cache topology, our scheduler dynamically adapts to the workloads and cache topologies, making it more responsive to real-time changes and varying demands. This hybrid approach reduces cache misses significantly and enhances the execution efficiency of parallel loops, leading to better performance metrics across various benchmarks when compared to using purely static or dynamic methods.

## Evaluation Tasks

- We use three main different kinds of tasks aimed at generating different variance of workloads between each iteration in order to test the capabilities and pros and cons of the various scheduling schemes. The three different kinds of tasks and their purposes are briefly described respectively below.

- LINPACK test bench: The LINPACK test bench solves a system of linear equations in which the problem size is the number of linear equations. This is a type of task in which the workloads between each iteration are more evenly distributed in which the static scheduling scheme should be able to benefit from.
- PrimeCheck test bench: The PrimeCheck test bench calculates the number of prime numbers less than the number corresponding to the current iteration index. This is a type of task in which the workloads between each iteration are not evenly distributed and the workloads of higher iterations would be larger that would benefit the dynamic scheduling scheme.
- RandomWorkload test bench: The RandomWorkload test bench simulates a random workload by having the thread to wait for a random amount of time before it is viewed that the thread has finished its task. This is a type of task in which the workloads between each iteration are not evenly distributed and there is no obvious pattern in the workload since the workloads are sampled to be completely random between each iteration.

## Performance Measurement

- The two main metrics that we concern is the processing time and #cache miss, which are measured by the perf tool in Linux OS.
- Cache misses for the Local Machine (M1 Pro Chip) are not measured due to the unavailability of a widely used performance measuring tool such as perf in Linux OS.

## Cache Miss Number under Various Problem Size and Full Hardware Utilization

- In this part of evaluation, the size of the problem set or the working set is varied while the underlying hardware are fully utilized, which means all the cores and the caches are utilized.
- The computation time of each experiment is averaged over 100 times in order to gain a more robust measure of the computation time of the tasks under different scheduling schemes.

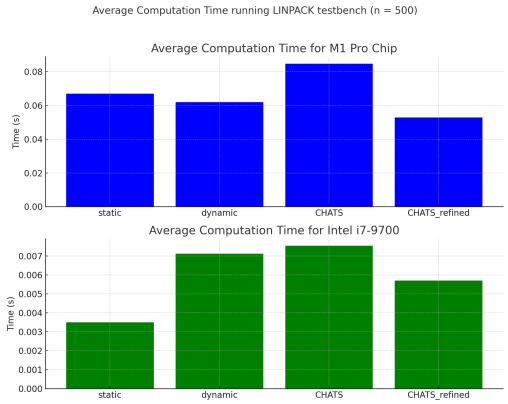
### LINPACK test bench (n = 500)

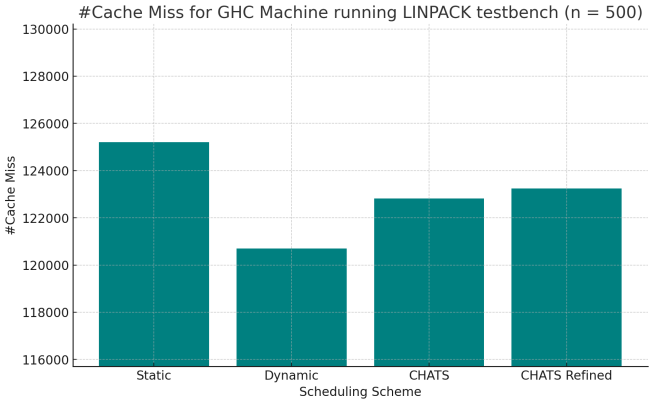
- Local Machine (M1 Pro Chip)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.0670 | 0.0619 | 0.0848 | 0.0528 |
| **Total Speedup** | 1.00x | 1.08x | 0.79x | 1.27x |
| **#Cache Miss** | N/A | N/A | N/A | N/A |

- GHC Machine (Intel i7-9700)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.00350 | 0.00712 | 0.00755 | 0.00571 |
| **Total Speedup** | 1.00x | 0.49x | 0.46x | 0.61x |
| **#Cache Miss** | 125210 | 120705 | 122825 | 123248 |



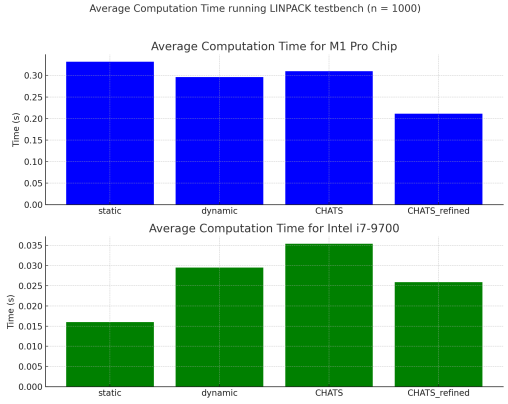

## LINPACK test bench (n = 1000)

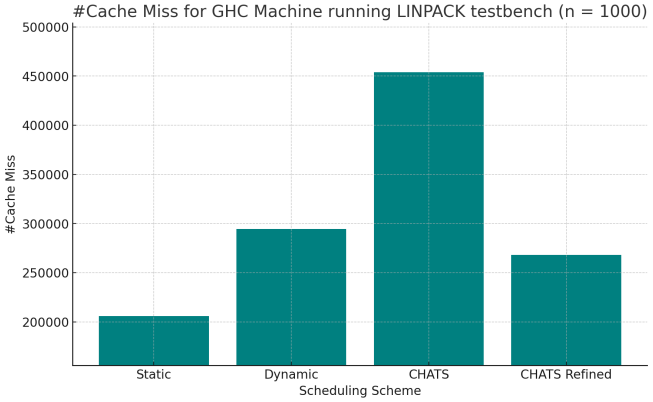
- Local Machine (M1 Pro Chip)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.332 | 0.296 | 0.310 | 0.211 |
| **Total Speedup** | 1.00x | 1.12x | 1.07x | 1.57x |
| **#Cache Miss** | N/A | N/A | N/A | N/A |

- GHC Machine (Intel i7-9700)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.0160 | 0.0295 | 0.02721 | 0.0259 |
| **Total Speedup** | 1.00x | 0.54x | 0.45x | 0.61x |
| **#Cache Miss** | 205710 | 294306 | 453914 | 268141 |




## LINPACK test bench (n = 1500)

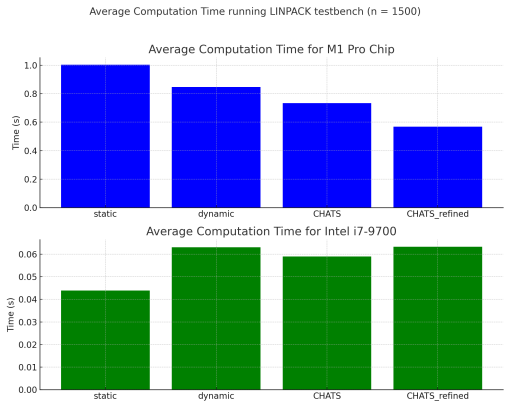
- Local Machine (M1 Pro Chip)

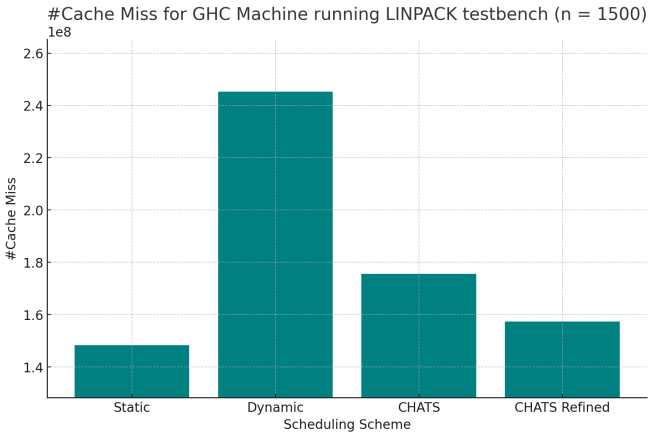
| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 1.004 | 0.847 | 0.733 | 0.569 |

| | | | | |
|---|---|---|---|---|
| **Total Speedup** | 1.00x | 1.19x | 1.37x | 1.76x |
| **#Cache Miss** | N/A | N/A | N/A | N/A |

- GHC Machine (Intel i7-9700)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.0439 | 0.0630 | 0.0590 | 0.0633 |
| **Total Speedup** | 1.00x | 0.70x | 0.74x | 0.69x |
| **#Cache Miss** | 148301222 | 245246470 | 175582127 | 157304158 |




## PrimeCheck test bench (n = 500)

- Local Machine (M1 Pro Chip)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.00308 | 0.00159 | 0.00180 | 0.00150 |
| **Total Speedup** | 1.00x | 1.94x | 1.71x | 2.05x |
| **#Cache Miss** | N/A | N/A | N/A | N/A |

- GHC Machine (Intel i7-9700)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.00252 | 0.000931 | 0.00109 | 0.00140 |
| **Total Speedup** | 1.00x | 2.71x | 2.29x | 1.8x |
| **#Cache Miss** | 94103 | 90131 | 90860 | 86905 |

## PrimeCheck test bench (n = 1000)

- Local Machine (M1 Pro Chip)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.0204 | 0.00872 | 0.00923 | 0.00963 |
| **Total Speedup** | 1.00x | 2.34x | 2.21x | 2.12x |
| **#Cache Miss** | N/A | N/A | N/A | N/A |

- GHC Machine (Intel i7-9700)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.0176 | 0.00642 | 0.00879 | 0.00981 |
| **Total Speedup** | 1.00x | 2.75x | 2.02x | 1.8x |
| **#Cache Miss** | 104834 | 96960 | 103733 | 102168 |

### PrimeCheck test bench (n = 1500)

- Local Machine (M1 Pro Chip)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.0633 | 0.0268 | 0.0274 | 0.0303 |
| **Total Speedup** | 1.00x | 2.36x | 2.31x | 2.09x |
| **#Cache Miss** | N/A | N/A | N/A | N/A |

- GHC Machine (Intel i7-9700)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.0551 | 0.0201 | 0.0227 | 0.0308 |
| **Total Speedup** | 1.00x | 2.75x | 2.5x | 1.79x |
| **#Cache Miss** | 127178 | 109035 | 120631 | 111066 |

---

### RandomWorkload test bench (n = 50)

- Local Machine (M1 Pro Chip)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.456 | 0.374 | 0.496 | 0.384 |
| **Total Speedup** | 1.00x | 1.22x | 0.92x | 1.19x |
| **#Cache Miss** | N/A | N/A | N/A | N/A |

- GHC Machine (Intel i7-9700)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.435 | 0.359 | 0.795 | 0.408 |
| **Total Speedup** | 1.00x | 1.21x | 0.55x | 1.07x |
| **#Cache Miss** | 4054945 | 4271452 | 4748712 | 4158517 |

### RandomWorkload  test bench (n = 100)

- Local Machine (M1 Pro Chip)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.816 | 0.701 | 0.803 | 0.713 |
| **Total Speedup** | 1.00x | 1.16x | 1.02x | 1.14x |
| **#Cache Miss** | N/A | N/A | N/A | N/A |

- GHC Machine (Intel i7-9700)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 0.792 | 0.662 | 0.967 | 0.766 |

| | | | | |
|---|---|---|---|---|
| **Total Speedup** | 1.00x | 1.20x | 0.82x | 1.03x |
| **#Cache Miss** | 4544838 | 4719815 | 4629401 | 4643663 |

### RandomWorkload  test bench (n = 150)

- Local Machine (M1 Pro Chip)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 1.186 | 1.032 | 1.145 | 1.046 |
| **Total Speedup** | 1.00x | 1.15x | 1.04x | 1.13x |
| **#Cache Miss** | N/A | N/A | N/A | N/A |

- GHC Machine (Intel i7-9700)

| **scheduling scheme** | static | dynamic | CHATS | CHATS_refined |
|---|---|---|---|---|
| **Average Computation Time(s)** | 1.134 | 0.992 | 1.326 | 1.095 |
| **Total Speedup** | 1.00x | 1.14x | 0.86x | 1.04x |
| **#Cache Miss** | 6546717 | 6526243 | 7246848 | 6665742 |

## Performance Analysis

- On the Local Machine (M1 Pro Chip), CHATS-refined is able to achieve the best performance out of all the scheduling schemes on the LINPACK test bench, showing its effectiveness on surpassing the performance of static scheduling mechanism on tasks that are more evenly distributed due to the extra flexibility for threads to work on other tasks when it is done with its statically assigned tasks. On tasks in which the workloads between each iteration are not evenly distributed, CHATS-refined is able to surpass the performance of static scheduling in terms of computation time. It is also able to achieve a very similar performance compared to that of the dynamic approach on the experiments we conducted.
- On the GHC Machine (Intel i7-9700), the performance of CHATS-refined almost always lies somewhere between that of the static and dynamic scheduling scheme, both in terms of computation time and cache misses. This is expected since it is essentially a mixture of the two schemes. Although CHATS-refined is able to surpass the performance of static scheduling scheme on the LINPACK test bench on the local machine, this is not the case on the GHC machines. This is possibly due to the structure of the GHC machine, having no shared L2 caches, making CHATS-refined not able to benefit too much from the dynamically assigned tasks.

# Conclusion

- In this project, we have successfully implemented a cache hierarchy-aware task scheduler. We have proved its effectiveness on different cache hierarchies and different tasks, surpassing the performance of at least either the static or the dynamic scheme, and sometimes even both on specific cache hierarchies. CHATS-refined has proven its ability to serve as a task scheduling scheme that can take into consideration of the cache hierarchy and divide tasks into static and dynamic parts accordingly. Since it guarantees performance that is at least greater than one of the implemented scheduling schemes in OpenMP, we view CHATS-refined to serve as a general purpose scheduling scheme that can be used on the scheduling of arbitrary tasks and guarantee a good enough if not the best performance.

# Potential Future Works

- There are several points that can be done in the future to perfect the whole program:

    - Automate the process of constructing the underlying hardware topology
- There are several points that might potentially improve the performance:
    - Leverage also cache size and the task size: Since the working set size is the direct reason that affects the cache miss rate, we can implement a mechanism to estimate the tasks working set size in each iteration. Then improve the scheduling scheme additionally based on this metric.
    - Difference between E/P core: Since modern processor accommodates heterogeneous cores including performance core (P core) and efficiency core (E core), which is what we didn't include in our implementation. The difference in cache size and clock frequency is also a factor that can affects the scheduler's performance.
    - Dynamically switching between different scheduling schemes based on real-time cache performance: Extend the scheduler to make it capable of dynamically switching between task distribution strategies like static, dynamic, and cache-aware schedulers on-the-fly based on real-time cache usage and observed behavior.
    - The current implementation is making use of an array that is the same size as the number of iterations. This implementation is pretty straightforward and can work perfectly. However, as the problem size grows bigger, this array will also grow linearly. Future works will look into a better implementation of this concept and can potentially save both memory and time.

# Work of Each Teammates

| Task | Person in charge | Task Status |
|---|---|---|
| Build the initial version of the proposed scheduler. | Yen-Shuo | Done |
| Debug and customize the scheduler to make it easier to conduct experiments related to sensitivity analysis. | Calvin | Done |
| Use the scheduler and test its performance on LINPACK used for performance comparison between different types of schedulers. | Yen-Shuo | Done |
| Measure the performance gains of the proposed schedulers against OpenMP implemented scheduler types using PrimeCheck and RandomWorkload. Make adjustments if needed. | Calvin | Done |

- credit for each teammates: Calvin Lin: 40%, Yen-Shuo Su: 60%

# Reference


- Nader Khammassi, Jean-Christophe Le Lann, "Design and Implementation of a Cache Hierarchy-Aware Task Scheduling for Parallel Loops on Multicore Architectures"