PufferFish: NUMA-Aware Work-stealing Library using Elastic Tasks

Vivek Kumar
IIIT New Delhi, India
Outline

- Introduction
- Contributions
- Motivating analysis
- Insights and approach
- Implementation
- Experimental Evaluation
- Summary
Task Parallelism on Multicore Processors

```c
1. void foo() {
2.     finish {
3.         async S1; // Parallel Task-1
4.         async S2; // Parallel Task-2
5.     } // Synchronization point
6.     S3; // Starts after termination of Task-1 & Task-2
7. }
```

- Dynamic task parallelism using async-finish
  - async `fork` a new task that can run in parallel to other tasks inside `finish`
  - `finish` `joins` all async tasks created inside its scope
- High **productivity** due to serial-elision
  - Removing all async and `finish` constructs results in a valid sequential program
- High **performance** from work-stealing runtime
  - Each worker (victim) **push** and **pop** async on its **deque**
  - Idle worker (thief) **randomly** chooses a victim to **steal** a task
Merge Sort on **UMA** Multicore Processor

```c
int *A;
/* Parallel recursive MergeSort */
void Sort(int low, int high) {
    fork
    Chunks = 4
    Sort C1
    Sort C2
    Sort C3
    Sort C4
    join
    Irregular execution DAG
    Sort and Merge recursively creating four and two tasks, respectively
    fork
    Merge C1,C2
    Merge C3,C4
    join
    Merge C12,C34
}
/* End of code */
```

- Multicore processor with Uniform Memory Access (UMA)
  - High performance
  - Same latency to access a memory location by all cores
Merge Sort on **NUMA Multicore Processor**

```c
int *A;
/* Parallel recursive MergeSort */
void Sort(int low, int high) {
    fork
    int Chunks = 4
    Sort C1, Sort C2, Sort C3, Sort C4
    join
    Merge C1,C2, Merge C3,C4
    fork
    Chunks = 4
    Sort C1, Sort C2
    join
    Merge C1,C2, Merge C3,C4
    join
    Merge C12,C34
    } /* End of code */
```

- **Multicore processor with Non-UMA (NUMA)**
  - Low performance
  - Random work-stealing disrupts the locality
    - Task and its data may not be on the same NUMA node
    - Thief doesn’t prioritize local steal over remote steal
The Problem

Work-Stealing in a Recursive Application with Irregular Execution DAG

- How to schedule a task on a NUMA node that has the task’s data
  - Programmer based task mapping
    - Program modification
    - Breaks serial elision
- How to prioritize local steal over remote steals
  - Hierarchical work-stealing
    - Remote steal breaks locality
    - Not stealing from remote node can starve workers within a node

int *A;
/* Parallel recursive MergeSort */
void Sort(int low, int high) {
  Sort C1
  Sort C2
  Sort C3
  Sort C4
  join
  Merge C1 C2
  Merge C3 C4
  join
  Merge C1 C2 C3 C4
} /* End of code */
Contributions

PufferFish programming model

For NUMA-aware task parallelism that uses data-affinity hints and *almost* supports serial elision

Lightweight work-stealing implementation

That integrates data-affinity hints with a hierarchical work-stealing library without causing starvation

Locality preserving hierarchical elastic tasks

That improves locality by reducing context switches at task creation by increasing or decreasing its parallelism

Detailed performance study

Using both micro and real-world benchmarks on a 32-core NUMA processor
Merge Sort using Hierarchical Place Trees (HPT [1])

1. int *A;
2. void Sort(int low, int high) {
3. if((high-low)<LIMIT) return SeqSort(low, high);
4. int Chunks=(high-low)/4;
5. finish {
6. async Sort/*Chunk C1*/();
7. async Sort/*Chunk C2*/();
8. async Sort/*Chunk C3*/();
9. async Sort/*Chunk C4*/();
10. }
11. finish {
12. async Merge/*Chunk C1*/, /*Chunk C2*/();
13. async Merge/*Chunk C3*/, /*Chunk C4*/();
14. }
15. Merge/*Chunk C12*/, /*Chunk C34*/();
16. }
17. void kernel() {
18. A = new int[N];
19. initialize();
20. Sort(0, N);
21. delete(A)
22. }

• HPT implementation in HClib [2]
  – Top-level task partitioning by programmer
    • Required at each finish scopes
      – Breaks serial elision property of async-finish
  – Hierarchical work-stealing
    • Worker W0 attempts to pop task from P5, P1, P0, and then attempts to steal also in same order if pop failed
      – Starvation at NUMA places P2 and P3 during Merge

Insights and Approach

• Preserve serial elision in async-finish programming over NUMA processor
  – PufferFish programming model for integrating data-affinity hints in an async

<table>
<thead>
<tr>
<th>async_hinted</th>
<th>numa_alloc_block_cyclic</th>
<th>numa_alloc_interleaved</th>
<th>numa_free</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign data-affinity hints with an async task</td>
<td>Block cyclic allocation of physical pages on NUMA nodes</td>
<td>Round-robin allocation of physical pages over NUMA nodes</td>
<td>Deallocate the physical pages</td>
</tr>
</tbody>
</table>

• Hierarchical work-stealing should neither break the task locality, nor it should induce starvation
  – Automatically calculate place to push async_hinted
  – If there is no load imbalance at a worker’s leaf place, let it directly execute the task
    • Avoids context switch at task creation and improves locality
Merge Sort using PufferFish Programming Model

1. int *A;
2. void Sort(int low, int high)
3. if((high-low)<LIMIT) return SeqSort(low, high);
4. int Chunks=(high-low)/4;
5. finish {
6. async Sort(*Chunk C1*/);
7. async Sort(*Chunk C2*/);
8. async Sort(*Chunk C3*/);
9. async Sort(*Chunk C4*/);
10. }
11. finish {
12. async Merge(*Chunk C1*, *Chunk C2*/);
13. async Merge(*Chunk C3*, *Chunk C4*/);
14. } Merge(*Chunk C12*, *Chunk C34*/);
15. void kernel() {
16. A = new int[N];
17. initialize();
18. Sort(0, N);
19. delete(A)
20. }

async-finish for UMA

async_hinted-finish for NUMA

- PufferFish programming model
  - Implemented over HPT implementation in HCLib
  - Assigns data-affinity hints to async tasks instead of place affinity
    - No program modification based on NUMA architecture
    - Supports serial elision
      - Except for two NUMA memory allocation/deallocation APIs
Hierarchical Elastic Tasks

1. void async_hinded (void* Array, int start_index, int end_index, task_t* task) {
2.     Place* DRAM_place = get_place_with_physicalPages(Array, start_index, end_index);
3.     if (/* Physical pages map to multiple DRAM places */) {
4.         async_at_hpt (PO, task);
5.     } else {
6.         if (/* Current worker is under same DRAM_place */) {
7.             if (/* No failed steals at current worker’s leaf place */) {
8.                 direct_execution (task); /* Avoids a context switch with push */
9.             } else {
10.                async_at_hpt (/* Current worker’s leaf place */, task);
11.             }
12.         } else {
13.             async_at_hpt (DRAM_place, task);
14.         }
15.     }
16.}
Hierarchical Work-Stealing

- Modifications to HPT in HCLib
  - Worker can **pop** only from its leaf place
  - Hierarchical **steals** within a NUMA domain and then from logical root
    - W0 at place P5 steal from all deques at places P5, P1, P6, and P0, respectively until successful
      - Strict locality without worker starvation
Performance Analysis on AMD EPYC 7551

Executing summary for seven recursive benchmarks with regular/irregular DAG on a 32-core processor with four NUMA nodes
Summary and Conclusion

• Mapping async-finish to NUMA node in recursive applications
  – Breaks serial elision
  – Create starvation

• PufferFish
  – async-finish programming model with data-affinity hints instead of NUMA place hints
    • Almost serial elision
    • No program modifications for different NUMA configurations
  – Hierarchical work-stealing with strict locality and hierarchical elastic tasks
    • Improves locality without starvation
Artifact

• Open sourced on Github
  – https://github.com/hipec/pufferFish/archive/v1.0.zip

• Author information
  – http://vivkumar.github.io/