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

Vivek Kumar
IIIT New Delhi, India
Task Parallelism on Multicore Processors

```c
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.     Sort(/*Chunk C1*/);
6.     Sort(/*Chunk C2*/);
7.     Sort(/*Chunk C3*/);
8.     Sort(/*Chunk C4*/);
9.     Merge(/*Chunk C1*//*Chunk C2*/);
10.    Merge(/*Chunk C3*//*Chunk C4*/);
11. }
12. Merge(/*Chunk C1*//*Chunk C2*/);
13. Merge(/*Chunk C3*//*Chunk C4*/);
14. Merge(/*Chunk C1*//*Chunk C2*/);
15. }
16. ```
Introduction

Task Parallelism on Multicore Processors

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.}

Serial elision
High productivity
Task Parallelism on Multicore Processors

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. }

Serial elision

High productivity
Task Parallelism on Multicore Processors

```java
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. }
```

Serial elision
High productivity

Irregular execution DAG
Task Parallelism on Multicore Processors

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 C1*/2,*/Chunk C3*/);
16.}

Serial elision
High productivity

Multicore Processor

Irregular execution DAG
Task Parallelism on Multicore Processors

```c
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.}
```

Serial elision
High productivity

Irregular execution DAG

Multicore Processor
Task Parallelism on Multicore Processors

```
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. }
```

Serial elision
High productivity

Multicore Processor

Irregular execution DAG
Task Parallelism on Multicore Processors

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.}

Serial elision
High productivity

Irregular execution DAG

Multicore Processor
Task Parallelism on Multicore Processors

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.}

Serial elision
High productivity

Random work-stealing
High performance

Multicore Processor

Irregular execution DAG
Task Parallelism on **UMA** Multicore Processors

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.}

Motivation

Serial elision  
High productivity  

Uniform Memory Access (UMA)

Irregular execution DAG
Task Parallelism on **NUMA** Multicore Processors

Motivation

Serial elision

High productivity

Irregular execution DAG

Non Uniform Memory Access (NUMA)
Motivation

Task Parallelism on **NUMA** Multicore Processors

```c
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.}
```

- **Serial elision**
  - **High productivity**

- **Irregular execution DAG**

- **Non Uniform Memory Access (NUMA)**
Motivation

Task Parallelism on **NUMA** Multicore Processors

```c
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_at(0) Sort/*Chunk C1*/;
7.     async_at(1) Sort/*Chunk C2*/;
8.     async_at(2) Sort/*Chunk C3*/;
9.     async_at(3) Sort/*Chunk C4*/;
10. }
11. finish {
12.   async_at(0) Merge/*Chunk C12*/, /*Chunk C2*/;
13.   async_at(1) Merge/*Chunk C3*/ /*Chunk C4*/;
14. }
15. } Merge/*Chunk C12*/, /*Chunk C34*/;
16. }```

Serial elision
High productivity

Irregular execution DAG

Non Uniform Memory Access (NUMA)
Task Parallelism on **NUMA** Multicore Processors

```c
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_at(0) Sort(*Chunk C1*/);
7.     async_at(1) Sort(*Chunk C2*/);
8.     async_at(2) Sort(*Chunk C3*/);
9.     async_at(3) Sort(*Chunk C4*/);
10. }
11. finish {
12.   async_at(0) Merge(*Chunk C1*/, *Chunk C2*/);
13.   async_at(1) Merge(*Chunk C3*/, *Chunk C4*/);
14. }
15. Merge(*Chunk C12*/, *Chunk C34*/);
16.}
```

Serial elision

**High productivity**

**Motivation**

Irregular execution DAG

**Non Uniform Memory Access (NUMA)**

Node-0

Node-1

Node-2

Node-3

Shared Cache

DRAM
Task Parallelism on **NUMA** Multicore Processors

Motivation

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_at(0) Sort(/*Chunk C1*/);
7.     async_at(1) Sort(/*Chunk C2*/);
8.     async_at(2) Sort(/*Chunk C3*/);
9.     async_at(3) Sort(/*Chunk C4*/);
10. }
11. finish {
12.     async_at(0) Merge(/*Chunk C1*/, /*Chunk C2*/);
13.     async_at(1) Merge(/*Chunk C3*/, /*Chunk C4*/);
14. }
15. Merge(/*Chunk C12*/, /*Chunk C34*/);
16.}
Motivation

Task Parallelism on NUMA Multicore Processors

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_at(0) Sort(*Chunk C1*/);
7.     async_at(1) Sort(*Chunk C2*/);
8.     async_at(2) Sort(*Chunk C3*/);
9.     async_at(3) Sort(*Chunk C4*/);
10. }
11. finish {
12.     async_at(0) Merge(*Chunk C1*/, *Chunk C2*/);
13.     async_at(1) Merge(*Chunk C3*/, *Chunk C4*/);
14. }
15. Merge(*Chunk C12*/ , *Chunk C34*/);
16.}

Serial elision
High productivity

Irregular execution DAG

Non Uniform Memory Access (NUMA)
Motivation

Task Parallelism on **NUMA** Multicore Processors

```
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_at(0) Sort(*Chunk C1*/);
7.     async_at(1) Sort(*Chunk C2*/);
8.     async_at(2) Sort(*Chunk C3*/);
9.     async_at(3) Sort(*Chunk C4*/);
10. }
11. finish {
12.   async_at(0) Merge(*Chunk C1*/, *Chunk C2*/);
13.   async_at(1) Merge(*Chunk C3*/, *Chunk C4*/);
14. }
15. Merge(*Chunk C12*/*, Chunk C34*/);
16.}
```

Serial elision
**High productivity**

Irregular execution DAG

**Hierarchical work-stealing**

1. Worker Starvation

**Non Uniform Memory Access** *(NUMA)*

Node-0  Node-1  Node-2  Node-3
Motivation

Task Parallelism on **NUMA** Multicore Processors

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_at(0) Sort(/*Chunk C1*/);
7.     async_at(1) Sort(/*Chunk C2*/);
8.     async_at(2) Sort(/*Chunk C3*/);
9.     async_at(3) Sort(/*Chunk C4*/);
10. }
11. finish {
12.   async_at(0) Merge(/*Chunk C1*/, /*Chunk C2*/);
13.   async_at(1) Merge(/*Chunk C3*/, /*Chunk C4*/);
14. }
15. Merge(/*Chunk C12*/, /*Chunk C34*/);
16.}

- **Serial elision**
  - High productivity

- **Irregular execution DAG**

- **Hierarchical work-stealing**
  - 2. Bad Locality

- **Non Uniform Memory Access (NUMA)**
Task Parallelism on **NUMA** Multicore Processors

Motivation

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_at(0) Sort(*Chunk C1*/);
7.    async_at(1) Sort(*Chunk C2*/);
8.    async_at(2) Sort(*Chunk C3*/);
9.    async_at(3) Sort(*Chunk C4*/);
10. }
11. finish {
12.    async_at(0) Merge(*Chunk C1*/, *Chunk C2*/);
13.    async_at(1) Merge(*Chunk C3*/, *Chunk C4*/);
14. }
15. Merge(*Chunk C12*/, *Chunk C34*/);
16.}

3. No Serial elision
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_hinted (A, C1_start, C1_end) Sort(/*Chunk C1*/);
7.     async_hinted (A, C2_start, C2_end) Sort(/*Chunk C2*/);
8.     async_hinted (A, C3_start, C3_end) Sort(/*Chunk C3*/);
9.     async_hinted (A, C4_start, C4_end) Sort(/*Chunk C4*/);
10. } 
11. finish {
12.     async_hinted (A, C1_start, C2_end) Merge(/*Chunk C1*/, /*Chunk C2*/);
13.     async_hinted (A, C3_start, C4_end) Merge(/*Chunk C3*/, /*Chunk C4*/);
14. } 
15. Merge(/*Chunk C12*/, /*Chunk C34*/;
16.}

1. Supports Serial elision
(except for using NUMA aware malloc/free)
Library for Task Parallelism & Work-Stealing over NUMA Processors

1. Supports Serial elision (except for using NUMA aware malloc/free)

2. Supports Irregular execution DAG

```
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_hinted (A, C1_start, C1_end) Sort(/*Chunk C1*/);
7.        async_hinted (A, C2_start, C2_end) Sort(/*Chunk C2*/);
8.        async_hinted (A, C3_start, C3_end) Sort(/*Chunk C3*/);
9.        async_hinted (A, C4_start, C4_end) Sort(/*Chunk C4*/);
10.    }
11.    finish {
12.        async_hinted (A, C1_start, C2_end) Merge(/*Chunk C1*/, /*Chunk C2*/);
13.        async_hinted (A, C3_start, C4_end) Merge(/*Chunk C3*/, /*Chunk C4*/);
14.    }
15.    Merge(/*Chunk C12*/, /*Chunk C34*/);
16.}
```
Contribution: PufferFish

Library for Task Parallelism & Work-Stealing over NUMA Processors

1. Supports Serial elision (except for using NUMA aware malloc/free)
2. Supports Irregular execution DAG
3. Improves Locality
4. Removes Starvation

Hierarchical work-stealing
Contribution: PufferFish

Library for Task Parallelism & Work-Stealing over NUMA Processors

1. Supports Serial elision (except for using NUMA aware malloc/free)
2. Supports Irregular execution DAG

3. Improves Locality
4. Removes Starvation

5. Hierarchical Elastic Tasks (further improves the locality)

Hierarchical work-stealing

NUMA Memory Hierarchy

Logical Root
Contribution: PufferFish

Library for Task Parallelism & Work-Stealing over NUMA Processors

1. Supports Serial elision (except for using NUMA aware malloc/free)

2. Supports Irregular execution DAG

3. Improves Locality

4. Removes Starvation

5. Hierarchical Elastic Tasks (further improves the locality)

Hierarchical work-stealing

NUMA Memory Hierarchy
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
```c
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_hinded (A, C1_start, C1_end) Sort(*Chunk C1*);
7.     async_hinded (A, C2_start, C2_end) Sort(*Chunk C2*);
8.     async_hinded (A, C3_start, C3_end) Sort(*Chunk C3*);
9.     async_hinded (A, C4_start, C4_end) Sort(*Chunk C4*);
10. }
11. finish {
12.    async_hinded (A, C1_start, C2_end) Merge(*Chunk C1*/, *Chunk C2*/);
13.    async_hinded (A, C3_start, C4_end) Merge(*Chunk C3*/, *Chunk C4*/);
14. }
15. Merge(*Chunk C12*/, *Chunk C34*/);
16. }
```