

C++ 태스크 기반 병렬 프로그래밍

김규래

서강대학교 전자공학과

April 6 2019

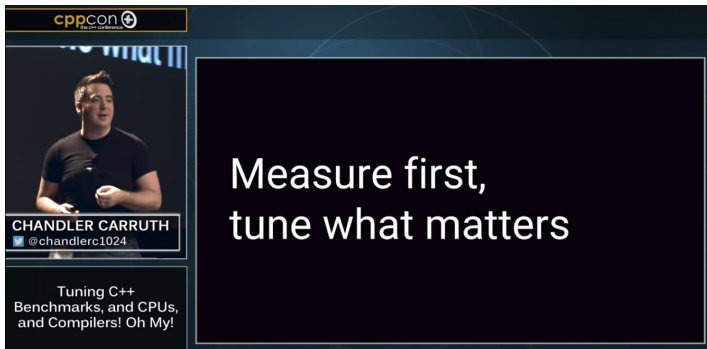



세션에서 다룰 내용

- 1 성능 최적화의 기본 원칙들
- 2 병렬 컴퓨팅 개요
- 3 병렬 컴퓨팅 방법 비교
- 4 Task Parallelism
- 5 Task Parallelism 프레임워크
- 6 Example



무조건 측정, 또 측정한다!



cppcon 

CHANDLER CARRUTH
@chandlerc1024

Tuning C++
Benchmarks, and CPUs,
and Compilers! Oh My!

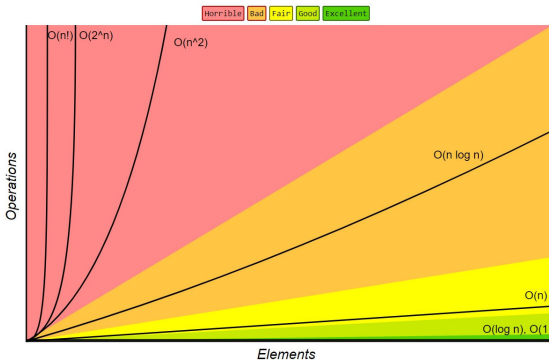
Measure first,
tune what matters

“성능을 실제로 벤치마킹 해보기 전까지는 절대 성능을 논하지
말아야 한다.”

Chandler Carruth, CppCon 2015



알고리즘적인 최적화가 최우선



복잡도의 벽은 많은 경우 넘어서기가 힘듭니다. ¹

¹Devopedia. 2018. "Algorithmic complexity." Version 6, December 7. Accessed 2019-03-23. <https://devopedia.org/algorithmic-complexity>



라이브러리, 프레임워크 교체를 먼저

Version	Speedup
Python	1
C	47
C parallelized	366
C parallelized & memory opt	6727
SIMD instructions	62806

프레임워크/시스템/언어에 따른 성능의 장벽은 생각보다 높습니다.²

²Leiserson et al., *There's Plenty of Room at The Top*, Under Review



성능 최적화의 두가지 관점

Latency 최소화

- 각 연산을 수행하는데 걸리는 시간을 최소화
- CPU Clock 조정, 캐시 최적화, strength reduction

Throughput 최대화

- 동일 시간에 처리하는 데이터의 양을 최대화
- 파이프라이닝, 병렬화
- 데이터의 양이 동일할 경우 전체적인 Latency 감소 효과



병렬화를 통한 Throughput 최대화

$$\text{Throughput} = \text{sockets} \times \frac{\text{cores}}{\text{socket}} \times \frac{\text{cycles}}{\text{second}} \times \frac{\text{operations}}{\text{cycle}}$$

병렬화를 하는 다양한 수단들

- CPU 에서 제공하는 SIMD instruction 을 사용
- Multithreading (Shared-memory parallelism)
- Distributed Computing (Distributed-memory parallelism)
- GPU, Xeon Phi 같은 병렬화에 특화된 하드웨어 사용



병렬화를 통한 Throughput 최대화

$$\text{Throughput} = \text{sockets} \times \frac{\text{cores}}{\text{socket}} \times \frac{\text{cycles}}{\text{second}} \times \frac{\text{operations}}{\text{cycle}}$$

병렬화를 하는 다양한 수단들

- CPU 에서 제공하는 SIMD instruction 을 사용
- Multithreading (Shared-memory parallelism)
- Distributed Computing (Distributed-memory parallelism)
- GPU, Xeon Phi 같은 병렬화에 특화된 하드웨어 사용



병렬화를 통한 Throughput 최대화

$$\text{Throughput} = \text{sockets} \times \frac{\text{cores}}{\text{socket}} \times \frac{\text{cycles}}{\text{second}} \times \frac{\text{operations}}{\text{cycle}}$$

병렬화를 하는 다양한 수단들

- CPU 에서 제공하는 SIMD instruction 을 사용
- Multithreading (Shared-memory parallelism)
- Distributed Computing (Distributed-memory parallelism)
- GPU, Xeon Phi 같은 병렬화에 특화된 하드웨어 사용



병렬화를 통한 Throughput 최대화

$$\text{Throughput} = \text{sockets} \times \frac{\text{cores}}{\text{socket}} \times \frac{\text{cycles}}{\text{second}} \times \frac{\text{operations}}{\text{cycle}}$$

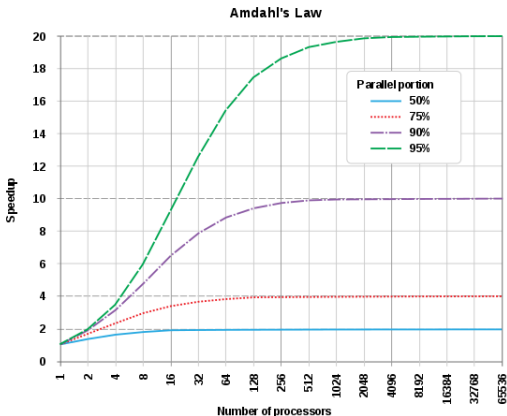
병렬화를 하는 다양한 수단들

- CPU 에서 제공하는 SIMD instruction 을 사용
- Multithreading (Shared-memory parallelism)
- Distributed Computing (Distributed-memory parallelism)
- GPU, Xeon Phi 같은 병렬화에 특화된 하드웨어 사용

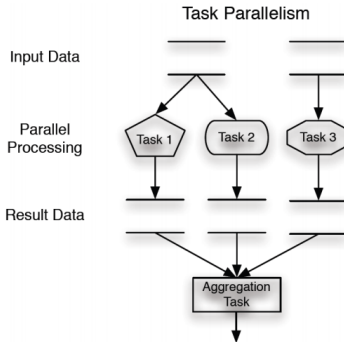
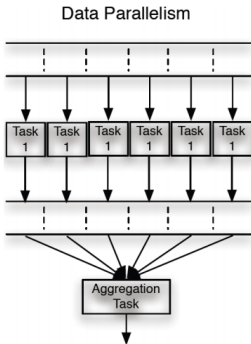


병렬의 근본적 한계

- 병렬화를 통해서 얻어낼 수 있는 성능은 매우 제한적입니다.
- 병렬화는 최후의 수단입니다.



Loop Parallelism vs Task Parallelism



3

³사진출처: Peter Tröger, *Parallel Programming Concepts: Programming Models*



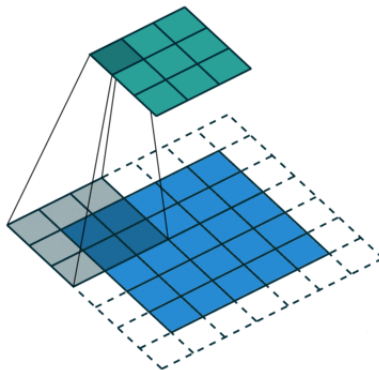
Loop Parallelism

- 데이터에 일괄적으로 수행되는 연산들을 병렬로 실행
- 기존 코드의 큰 수정 없이 쉽게 병렬화 적용 가능
- Fork-join Parallelism, Data Parallelism

```
#pragma omp parallel for  
for(int i = 0; i < N; ++i)  
{  
    c[i] = a[i] + b[i];  
}
```



2D Convolution

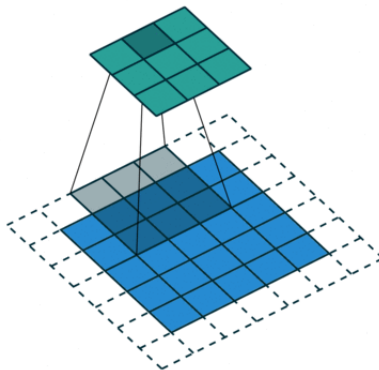


4

⁴사진출처: https://github.com/vdumoulin/conv_arithmetic



2D Convolution

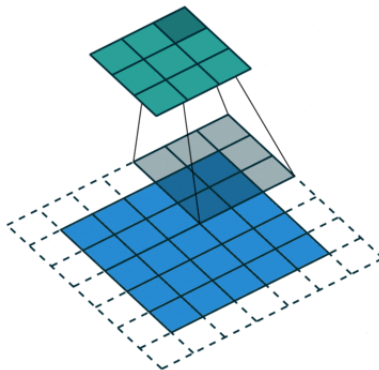


5

⁵사진출처: https://github.com/vdumoulin/conv_arithmetic



2D Convolution

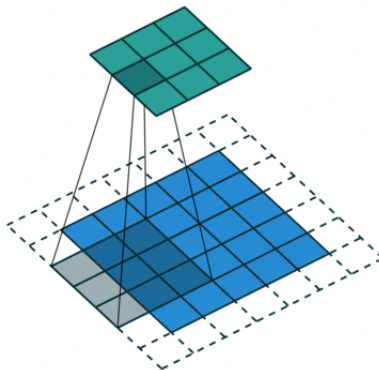


6

⁶사진출처: https://github.com/vdumoulin/conv_arithmetic



2D Convolution



7

⁷사진출처: https://github.com/vdumoulin/conv_arithmetic



2D Convolution

```

inline cv::Mat
filter(cv::Mat const& src,
      std::vector<float> const& kernel,
      size_t m, size_t n)
{
    /* ... */
    auto dst = cv::Mat(dst_m, dst_n, CV_32FC1);

    #pragma omp parallel for schedule(static) collapse(2)
    for(size_t i = 0; i < dst_m; ++i)
    {
        for(size_t j = 0; j < dst_n; ++j)
        {
            /* load image patch */

            float value = dot(buffer, kernel);
            dst.at<float>(i, j) = value;
        }
    }
    return dst;
}

```



2D Convolution

```
inline cv::Mat
filter(cv::Mat const& src,
       std::vector<float> const& kernel,
       size_t m, size_t n)
{
    /* ... */
    auto dst = cv::Mat(dst_m, dst_n, CV_32FC1);

    #pragma omp parallel for schedule(static) collapse(2)
    for(size_t i = 0; i < dst_m; ++i)
    {
        for(size_t j = 0; j < dst_n; ++j)
        {
            /* load image patch */

            float value = dot(buffer, kernel);
            dst.at<float>(i, j) = value;
        }
    }
    return dst;
}
```



Processing Pipeline

```
auto smooth    = filter(image, gaussian_kernel, 9, 9);  
auto x_edge    = filter(smooth, sobel_x_kernel, 3, 3);  
auto y_edge    = filter(smooth, sobel_x_kernel, 3, 3);  
auto gradient  = sobel_norm(x_edge, y_edge);  
auto output    = thresholding(x_edge, 0.7, 0.12);
```



Processing Pipeline

```
auto smooth    = filter(image, gaussian_kernel, 9, 9);  
auto x_edge    = filter(smooth, sobel_x_kernel, 3, 3);  
auto y_edge    = filter(smooth, sobel_x_kernel, 3, 3);  
auto gradient  = sobel_norm(x_edge, y_edge);  
auto output    = thresholding(x_edge, 0.7, 0.12);
```



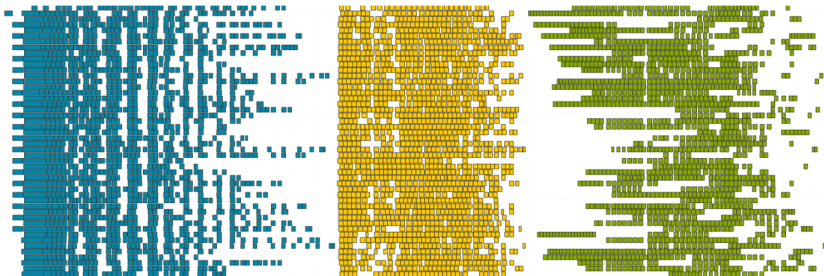
Example Image



Example Output



Loop Parallelism Example

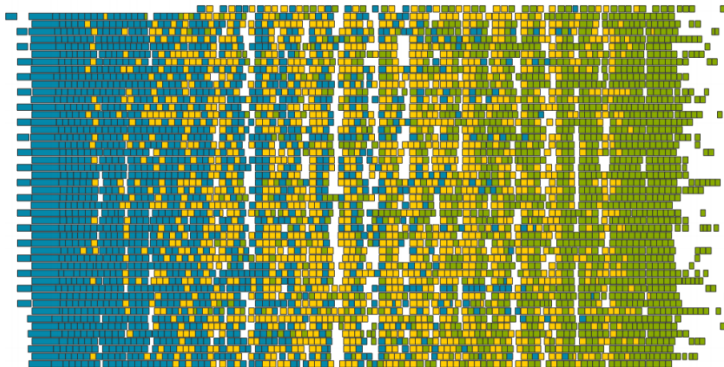


8

⁸사진출처: Abalenkovs, et al. Parallel Programming Models for Dense Linear Algebra on Heterogeneous Systems (2015).



Loop Parallelism Example



9

⁹사진출처: Abalenkovs, et al. Parallel Programming Models for Dense Linear Algebra on Heterogeneous Systems (2015).



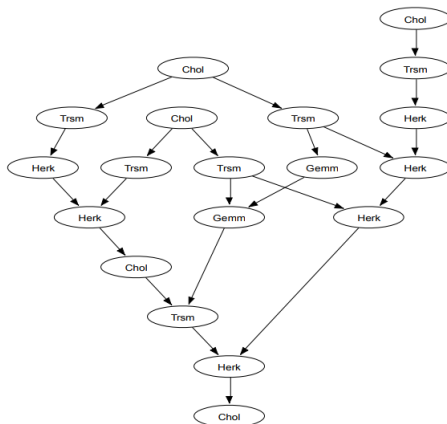
Task Parallelism

- 각 태스크의 종속관계를 파악하고 종속성이 없는 태스크들을 동적으로 병렬화
- 특수한 언어, 프레임워크를 통해 데이터 종속관계를 모델링해야 함
- Directed Acyclic Graph (DAG) 또는 Computation Tree 를 통해 종속관계 표현
- Cilk, HPX, TBB, cpp-taskflow

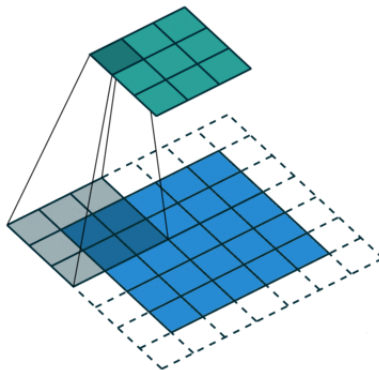


Directed Acyclic Graph (DAG)

- 사이클이 존재하지 않는 지향성 그래프.
- '연산' 들은 대부분 DAG 로 표현할 수 있습니다.



2D Convolution



10

¹⁰사진출처: https://github.com/vdumoulin/conv_arithmetic

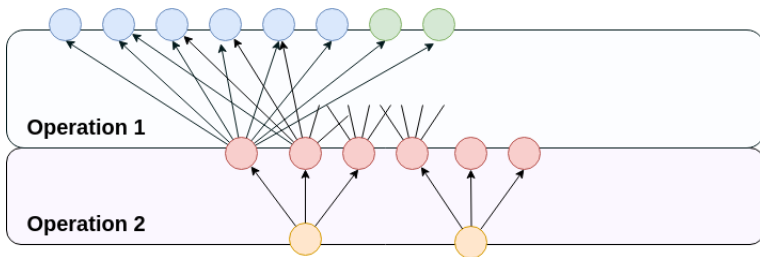


Processing Pipeline

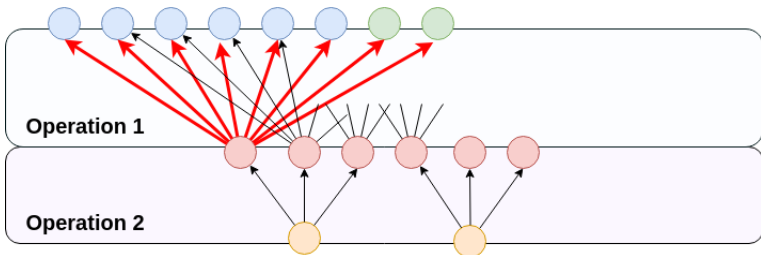
```
auto smooth    = filter(image, gaussian_kernel, 9, 9);  
auto x_edge    = filter(smooth, sobel_x_kernel, 3, 3);  
auto y_edge    = filter(smooth, sobel_x_kernel, 3, 3);  
auto gradient  = sobel_norm(x_edge, y_edge);  
auto output    = thresholding(x_edge, 0.7, 0.12);
```



Pixelwise Computation DAG



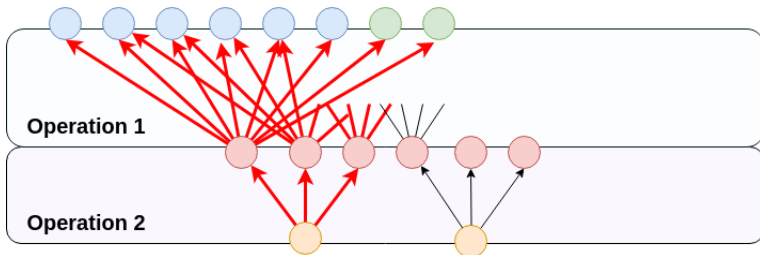
Pixelwise Computation DAG



- Loop 병렬의 경우 픽셀간의 병렬 관계를 한 단계에서만 활용합니다.
- 1단계에서 모든 픽셀들의 연산이 끝날 때까지 barrier 에서 blocking.



Pixelwise Computation DAG



- 실제로는 병렬 관계가 깊게까지 이어집니다.
- 이러한 병렬 가능한 구간을 자동으로 감지하기 위해서 DAG 형태로 연산을 표현.



Task Parallelism 기본 원리

프로그램 내에 존재하는 '깊은' 병렬화 가능 구간, 미세하게 병렬화가 가능한 task들을 종속관계 분석을 통해 병렬화하는 것이
Task parallelism.

- 1 DAG 생성
- 2 병렬화 가능한 task 들을 병렬로 실행
- 3 코어 간의 로드 임밸런스 해소



Task Parallelism 기본 원리

프로그램 내에 존재하는 '깊은' 병렬화 가능 구간, 미세하게 병렬화가 가능한 task들을 종속관계 분석을 통해 병렬화하는 것이
Task parallelism.

- 1 **DAG 생성**
API 또는 Domain Specific Language
- 2 **병렬화 가능한 task 들을 병렬로 실행**
DAG 를 순회하면서 task 들을 consume
- 3 **코어 간의 로드 임밸런스 해소**
Task Scheduling



Task Parallelism 기본 원리

프로그램 내에 존재하는 '깊은' 병렬화 가능 구간, 미세하게 병렬화가 가능한 task들을 종속관계 분석을 통해 병렬화하는 것이
Task parallelism.

- 1 **DAG 생성**
API 또는 Domain Specific Language
- 2 병렬화 가능한 task 들을 병렬로 실행
DAG 를 순회하면서 task 들을 consume
- 3 코어 간의 로드 임밸런스 해소
Task Scheduling



OpenMP 4.0

```
int fib(int n)
{
    int i, j;
    if (n<2)
    {
        return n;
    }
    else
    {
#pragma omp task shared(i)
        i = fib(n - 1);
#pragma omp task shared(j)
        j = fib(n - 2);
#pragma omp taskwait
        return i+j;
    }
}
```



OpenMP 4.0

```

for(int i=0; i<num_blocks; i++) {
    #pragma omp task depend( inout: block_list[i][i] )
    diag_func( block_list[i][i] );
    for(int j=i+1; j<numBlocks; j++) {
        #pragma omp task depend( in: block_list[i][i]) \
            depend(inout: block_list[i][j])
        row_func(block_list[i][j], block_list[i][i] );
    }
    for(int j=i+1; j<num_blocks; j++) {
        #pragma omp task depend( in: block_list[i][i]) \
            depend(inout: block_list[j][i])
        col_func( block_list[j][i], block_list[i][i] );
        /* ... */
    }
}

```



OpenMP Task System

```

for(int i=0; i<num_blocks; i++) {
#pragma omp task depend( inout: block_list[i][i] )
    diag_func( block_list[i][i] );
    for(int j=i+1; j<numBlocks; j++) {
#pragma omp task depend( in: block_list[i][i]) \
                        depend(inout: block_list[i][j])
        row_func(block_list[i][j], block_list[i][i] );
    }
    for(int j=i+1; j<num_blocks; j++) {
#pragma omp task depend( in: block_list[i][i]) \
                        depend(inout: block_list[j][i])
        col_func( block_list[j][i], block_list[i][i] );
        /* ... */
    }
}

```



OpenMP Task System

- OpenMP 4.0 이상을 지원하는 모든 컴파일러에서 사용 가능.
(GCC)
- 기존 코드의 큰 변경 없이 사용 가능.
- Shared-memory 만 지원.
- 생성할 수 있는 DAG 의 형태가 제한적.
- 데이터 기반으로 하고 싶을 경우 모든 것을 C스타일 배열만 사용 가능.
- GCC 의 경우 아직 Work-stealing 스케줄링이 구현돼 있지 않음.



CppTaskflow

```
// create three regular tasks
tf::Task A = tf.emplace([](){}).name("A");
tf::Task C = tf.emplace([](){}).name("C");
tf::Task D = tf.emplace([](){}).name("D");

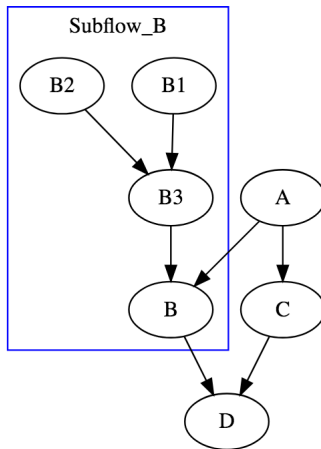
// create a subflow graph (dynamic tasking)
tf::Task B = tf.emplace([](tf::SubflowBuilder& subflow) {
    tf::Task B1 = subflow.emplace([](){}).name("B1");
    tf::Task B2 = subflow.emplace([](){}).name("B2");
    tf::Task B3 = subflow.emplace([](){}).name("B3");
    B1.precede(B3);
    B2.precede(B3);
}) .name("B");

A.precede(B); // B runs after A
A.precede(C); // C runs after A
B.precede(D); // D runs after B
C.precede(D); // D runs after C

tf.wait_for_all();
```



CppTaskflow



CppTaskflow

- Illinois 대학에서 오픈소스로 개발¹¹.
- Shared-memory 만 지원.
- 그래프 라이브러리들과 유사한 형태의 모던 C++ API.
- Work-stealing 알고리즘 탑재.
- map, filter, reduce 등의 알고리즘들 제공.

¹¹Huang et al., “Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++” in IPDPS, 2007,



HPX

```

template <typename Type>
inline hpx::future<int>
traverse(node& t)
{
    if(!t.has_children())
        return t.compute_result();
    auto results = std::vector<hpx::future<Type>>(factor);

    for(size_t i = 0; i < factor; ++i)
        results[i] = hpx::async(traverse, t.children[i]);

    return hpx::when_all(results, t.compute_results()).then(
        [](auto f, auto r){
            return combine_results(f, r);
        }
    );
    return t.compute_result();
}

```



HPX

```
template <typename Type>
inline hpx::future<int>
traverse(node& t)
{
    if(!t.has_children())
        return t.compute_result();
    auto results = std::vector<hpx::future<Type>>(factor);

    for(size_t i = 0; i < factor; ++i)
        results[i] = hpx::async(traverse, t.children[i]);

    return hpx::when_all(results, t.compute_results()).then(
        [](auto f, auto r){
            return combine_results(f, r);
        }
    );
    return t.compute_result();
}
```



HPX

R f(p...)	Synchronous Execution (returns R)	Asynchronous Execution (returns future<R>)	Fire & Forget Execution (returns void)
Functions (direct invocation)	f(p...) C++	async(f, p...)	apply(f, p...)
Functions (lazy invocation)	bind(f, p...)(...)	async(bind(f, p...), ...) C++ Standard Library	apply(bind(f, p...), ...)
Actions (direct invocation)	HPX_ACTION(f, action) a(id, p...)	HPX_ACTION(f, action) async(a, id, p...)	HPX_ACTION(f, action) apply(a, id, p...)
Actions (lazy invocation)	HPX_ACTION(f, action) bind(a, id, p...) (...)	HPX_ACTION(f, action) async(bind(a, id, p...), ...)	HPX_ACTION(f, action) apply(bind(a, id, p...), ...)

HPX



HPX

- Louisiana 주립대학에서 오픈소스로 개발¹².
- Shared-memory 와 AGAS 형태의 분산 메모리 프로그래밍 지원.
- 표준 C++ 기반의 Task Parallelism 과 Loop Parallelism 프로그래밍 지원.
- C++ STL 알고리즘들의 병렬 버전 지원.
- 다양한 Work-stealing 스케줄링 제공.
- 빌드 시스템을 HPX 에 맞춰야된다는 것이 단점.

¹²Kaiser, Hartmut, et al. "Hpx: A task based programming model in a global address space." Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. ACM, 2014.



Example with HPX

```
auto smooth    = filter(image, gaussian_kernel, 9, 9);  
auto x_edge    = filter(smooth, sobel_x_kernel, 3, 3);  
auto y_edge    = filter(smooth, sobel_x_kernel, 3, 3);  
auto gradient  = sobel_norm(x_edge, y_edge);  
auto output    = thresholding(x_edge, 0.7, 0.12);
```



Example with HPX

```
inline cv::Mat
filter(cv::Mat const& src,
      std::vector<float> const& kernel,
      size_t m, size_t n)
{
    /* ... */
    auto dst = cv::Mat(dst_m, dst_n, CV_32FC1);

    #pragma omp parallel for schedule(static) collapse(2)
    for(size_t i = 0; i < dst_m; ++i)
    {
        for(size_t j = 0; j < dst_n; ++j)
        {
            /* load image patch */

            float value = dot(buffer, kernel);
            dst.at<float>(i, j) = value;
        }
    }
    return dst;
}
```



Example with HPX

```
template<template<typename> class Future, size_t Block>
class future_image_block
{
private:
    size_t _image_m;
    size_t _image_n;
    size_t _block_m;
    size_t _block_n;
    std::vector<Future<std::vector<float>>> _futures;
public:
    inline future_image_block()
        : _futures(), _image_m(0), _image_n(0)
    {}

    inline future_image_block(size_t m, size_t n)
        : _image_m(m),
          _image_n(n),
          _block_m(ceil(m / static_cast<float>(Block))),
          _block_n(ceil(n / static_cast<float>(Block))),
          _futures(_block_m * _block_n)
    {}

    /* ... */
};
```



Example with HPX

```
template<template<typename> class Future, size_t Block>
inline decltype(auto)
filter(future_image_block<Future, Block>& src,
       std::vector<float> const& kernel,
       size_t m, size_t n)
{
    /* ... */
    auto dst = future_image_block<Future, Block>(dst_m, dst_n);
    for(size_t block_i = 0; block_i < dst.block_rows(); ++block_i)
    {
        for(size_t block_j = 0; block_j < dst.block_cols(); ++block_j)
        {
            dst.block(block_i, block_j) = hpx::async(
                [&src, &kernel, block_i, block_j, ker_m, ker_n, dst_m,
                 return block_filter(block_i, block_j,
                                     ker_m, ker_n,
                                     dst_m, dst_n,
                                     src, kernel);
            });
        }
    }
    return dst;
}
```

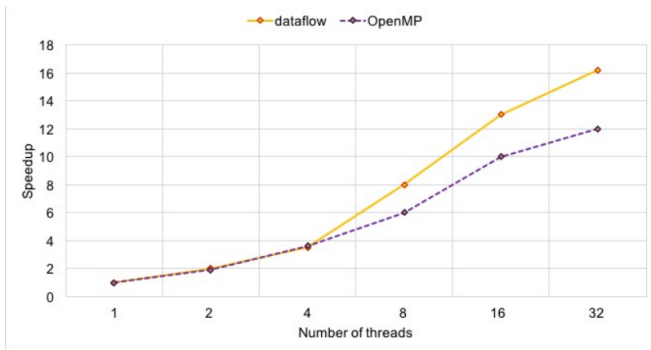


Example with HPX

```
template<template<typename> class Future, size_t Block>
inline decltype(auto)
filter(future_image_block<Future, Block>& src,
      std::vector<float> const& kernel,
      size_t m, size_t n)
{
    /* ... */
    auto dst = future_image_block<Future, Block>(dst_m, dst_n);
    for(size_t block_i = 0; block_i < dst.block_rows(); ++block_i)
    {
        for(size_t block_j = 0; block_j < dst.block_cols(); ++block_j)
        {
            dst.block(block_i, block_j) = hpx::async(
                [&src, &kernel, block_i, block_j, ker_m, ker_n, dst_m,
                 return block_filter(block_i, block_j, ker_m, ker_n,
                                     dst_m, dst_n, src, kernel);
            );
        }
    }
    return dst;
}
```



Example with HPX



13

¹³Khatami, Zahra, Hartmut Kaiser, and J. Ramanujam. "Redesigning op2 compiler to use hpx runtime asynchronous techniques." 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, 2017.



Results

- Task Parallelism 을 사용할 경우 Loop Parallelism 에 비해서 더 높은 성능을 얻을 수가 있습니다.
- Loop Parallelism 에 비해서 파라미터 튜닝을 조금 더 거쳐야 합니다.
- 비동기 프로그래밍이라서 머리가 좀 더 아픕니다...



Section

감사합니다

