Parallelism in C++

Higher-level Parallelization in C++ for Asynchronous Task-Based Programming

Hartmut Kaiser (hartmut.kaiser@gmail.com)
State of the Art

- Modern architectures impose massive challenges on programmability in the context of performance portability
  - Massive increase in on-node parallelism
  - Deep memory hierarchies
- Only portable parallelization solution for C++ programmers: OpenMP and MPI
  - Hugely successful for years
  - Widely used and supported
  - Simple use for simple use cases
  - Very portable
  - Highly optimized
Parallelism in C++

- C++11 introduced lower level abstractions
  - `std::thread`, `std::mutex`, `std::future`, etc.
  - Fairly limited, more is needed
  - C++ needs stronger support for higher-level parallelism

- Several proposals to the Standardization Committee are accepted or under consideration
  - Technical Specification: Concurrency (note: misnomer)
  - Technical Specification: Parallelism
  - Other smaller proposals: resumable functions, task regions, executors

- Currently there is no overarching vision related to higher-level parallelism
  - Goal is to standardize a ‘big story’ by 2020
  - No need for OpenMP, OpenACC, OpenCL, etc.
  - This talk tries to show results of our take on this
Concepts of Parallelism
Parallel Execution Properties

• The *execution restrictions* applicable for the work items
  • Restrictions imposed from thread-safety perspective
  • i.e. 'can be run concurrently', or 'has to be run sequentially', etc.

• In what *sequence* the work items have to be executed
  • Sometimes we know what needs to go first
  • i.e. 'this work item depends on the availability of a result', 'no restrictions apply', etc.

• *Where* the work items should be executed
  • i.e. 'on this core', 'on that node', 'on this NUMA domain', or 'wherever this data item is located', etc.

• The *parameters* of the execution environment
  • Controlling number of items directly or through execution time which should run together on the same thread of execution
  • i.e. grain size control
Concepts and Types of Parallelism

- **Parallel Algorithms...**
- **Fork-Join, etc....**
- **Executors...**
- **Executor Parameters...**
- **Futures, Async, Dataflow**

- Restrictions
- Sequence, Where
- Grainsize

Application

*Concepts*

*Execution Policies*

Parallelism in C++, Hartmut Kaiser
Execution Policies (std)

• Specify execution guarantees (in terms of thread-safety) for executed parallel tasks:

  - `sequential_execution_policy`: `seq`
  - `parallel_execution_policy`: `par`
  - `parallel_vector_execution_policy`: `par_vec`

• Special rules related to exception handling

• In parallelism TS used for parallel algorithms only
Execution Policies (Extensions)

- Extensions: asynchronous execution policies

  - `parallel_task_execution_policy` (asynchronous version of `parallel_execution_policy`), generated with `par(task)`
  - `sequential_task_execution_policy` (asynchronous version of `sequential_execution_policy`), generated with `seq(task)`

- In both cases the formerly synchronous functions return a future<>
- Instruct the parallel construct to be executed asynchronously
- Allows integration with asynchronous control flow
Executors

• Executor are objects responsible for
  • Creating execution agents on which work is performed (N4466)
  • In N4466 this is limited to parallel algorithms, here much broader use

• Thus they
  • Abstract the (potentially platform-specific) mechanisms for launching work

• Responsible for defining the Where and How of the execution of tasks
The simplest Executor possible

- Creating executors is trivial:

```cpp
struct simplest_parallel_executor
{
  template <typename F>
  future<result_of_t<F()>>    // requires(is_callable<F>())
  async_execute(F && f)
  {
    return async(std::forward<F>(f));
  }
};
```
Execution Parameters

• Allows to control the grain size of work
  • i.e. amount of iterations of a parallel `for_each` run on the same thread
  • Similar to OpenMP scheduling policies: static, guided, dynamic
  • Much more fine control
The simplest Executor Parameters

- Creating executor parameter policies is trivial:

```cpp
struct static_executor_parameter {
    template <typename Executor, typename F>
    std::size_t get_chunk_size(Executor& exec, F &&, std::size_t num_tasks) {
        std::size_t const cores = num_processing_units(exec);
        return (num_tasks + cores - 1) / cores;
    }
};
```
Rebind Execution Policies

- Execution policies have associated default executor and default executor parameters
  - par → parallel executor, static chunk size
  - seq → sequential executor, no chunking
- Rebind executor and executor parameters:

```cpp
numa_executor exec;
auto policy1 = par.on(exec); // rebind only executor

static_chunk_size param;
auto policy2 = par.with(param); // rebind only executor parameter

auto policy3 = par.on(exec).with(param); // rebind both
```
Stepping Aside
HPX – A General Purpose Runtime System for Applications of Any Scale
HPX – A General Purpose Runtime System

• Solidly based on a theoretical foundation – a well defined, new execution model (ParalleX)

• Exposes a coherent and uniform, standards-oriented API for ease of programming parallel and distributed applications.
  • Enables to write fully asynchronous code using hundreds of millions of threads.
  • Provides unified syntax and semantics for local and remote operations.

• HPX represents an innovative mixture of
  • A global system-wide address space (AGAS - Active Global Address Space)
  • Fine grain parallelism and lightweight synchronization
  • Combined with implicit, work queue based, message driven computation
  • Full semantic equivalence of local and remote execution, and
  • Explicit support for hardware accelerators (through percolation)
HPX – A General Purpose Runtime System

• Enables writing applications which out-perform and out-scale existing ones
  • A general purpose parallel C++ runtime system for applications of any scale
    • http://stellar-group.org/libraries/hpx
    • https://github.com/STEllAR-GROUP/hpx/

• Is published under Boost license and has an open, active, and thriving developer community.

• Can be used as a platform for research and experimentation
HPX – The API

- As close as possible to C++11/14 standard library, where appropriate, for instance:
  - `std::thread`  
  - `std::mutex`  
  - `std::future`  
  - `std::async`  
  - `std::bind`  
  - `std::function`  
  - `std::tuple`  
  - `std::any`  
  - `std::cout`  
  - `std::parallel::for_each, etc.`  
  - `std::parallel::task_region`  
  - `hpx::thread`  
  - `hpx::mutex`  
  - `hpx::future` (including N4107, ‘Concurrency TS’)  
  - `hpx::async` (including N3632)  
  - `hpx::bind`  
  - `hpx::function`  
  - `hpx::tuple`  
  - `hpx::any` (N3508)  
  - `hpx::cout`  
  - `hpx::parallel::for_each` (N4105, ‘Parallelism TS’)  
  - `hpx::parallel::task_region` (N4088)
Futures, Async, Dataflow

Task-based Parallelism
What is a (the) future

- A future is an object representing a result which has not been calculated yet

- Enables transparent synchronization with producer
- Hides notion of dealing with threads
- Makes asynchrony manageable
- Allows for composition of several asynchronous operations
- (Turns concurrency into parallelism)
What is a (the) Future?

- Many ways to get hold of a future, simplest way is to use (std) async:

```cpp
int universal_answer() { return 42; }

void deep_thought()
{
    future<int> promised_answer = async(&universal_answer);

    // do other things for 7.5 million years

    cout << promised_answer.get() << endl;  // prints 42, eventually
}
```
Compositional facilities

- Sequential composition of futures

```cpp
future<string> make_string()
{
    future<int> f1 = async([]() -> int { return 123; });

    future<string> f2 = f1.then(
        [](future<int> f) -> string
        {
            return to_string(f.get());  // here .get() won’t block
        });

    return f2;
}
```
Compositional facilities

- Parallel composition of futures

```cpp
future<int> test_when_all()
{
    future<int> future1 = async([]() -> int { return 125; });
    future<string> future2 = async([]() -> string { return string("hi"); });

    // future<tuple<future<int>, future<string>>>
    auto all_f = when_all(future1, future2); // also: when_any, etc.

    future<int> result = all_f.then(
        [](auto f) -> int {
            return do_work(f.get());
        });

    return result;
}
```
Dataflow – The New ‘async’ (HPX)

• What if one or more arguments to ‘async’ are futures themselves?

• Normal behavior: pass futures through to function

• Extended behavior: wait for futures to become ready before invoking the function:

```cpp
template <typename F, typename... Arg>
future<result_of_t<F(Ags...)>> // requires(is_callable<F(Ags...)>)
dataflow(F && f, Arg &&... arg);
```

• If ArgN is a future, then the invocation of F will be delayed

• Non-future arguments are passed through
Parallel Algorithms
# Parallel Algorithms

<table>
<thead>
<tr>
<th>adjacent_difference</th>
<th>adjacent_find</th>
<th>all_of</th>
<th>any_of</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>copy_if</td>
<td>copy_n</td>
<td>count</td>
</tr>
<tr>
<td>count_if</td>
<td>equal</td>
<td>exclusive_scan</td>
<td>fill</td>
</tr>
<tr>
<td>fill_n</td>
<td>find</td>
<td>find_end</td>
<td>find_first_of</td>
</tr>
<tr>
<td>find_if</td>
<td>find_not</td>
<td>for_each</td>
<td>for_each_n</td>
</tr>
<tr>
<td>generate</td>
<td>generate_n</td>
<td>includes</td>
<td>inclusive_scan</td>
</tr>
<tr>
<td>inner_product</td>
<td>inplace_merge</td>
<td>is_heap</td>
<td>is_heap_until</td>
</tr>
<tr>
<td>is_partitioned</td>
<td>is_sorted</td>
<td>is_sorted_until</td>
<td>lexicographical_compare</td>
</tr>
<tr>
<td>max_element</td>
<td>merge</td>
<td>min_element</td>
<td>minmax_element</td>
</tr>
<tr>
<td>mismatch</td>
<td>move</td>
<td>none_of</td>
<td>nth_element</td>
</tr>
<tr>
<td>partial_sort</td>
<td>partial_sort_copy</td>
<td>partition</td>
<td>partition_copy</td>
</tr>
<tr>
<td>reduce</td>
<td>remove</td>
<td>remove_copy</td>
<td>remove_copy_if</td>
</tr>
<tr>
<td>remove_if</td>
<td>replace</td>
<td>replace_copy</td>
<td>replace_copy_if</td>
</tr>
<tr>
<td>replace_if</td>
<td>reverse</td>
<td>reverse_copy</td>
<td>rotate</td>
</tr>
<tr>
<td>rotate_copy</td>
<td>search</td>
<td>search_n</td>
<td>set_difference</td>
</tr>
<tr>
<td>set_intersection</td>
<td>set_symmetric_difference</td>
<td>set_union</td>
<td>sort</td>
</tr>
<tr>
<td>stable_partition</td>
<td>stable_sort</td>
<td>swap_ranges</td>
<td>transform</td>
</tr>
<tr>
<td>uninitialized_copy</td>
<td>uninitialized_copy_n</td>
<td>uninitialized_fill</td>
<td>uninitialized_fill_n</td>
</tr>
<tr>
<td>unique</td>
<td>unique_copy</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
std::vector<int> v = { 1, 2, 3, 4, 5, 6 };  
parallel::transform(  
    parallel::par, begin(v), end(v),  
    [](int i) -> int  
    {  
        return i + 1;  
    });

// prints: 2,3,4,5,6,7,  
for (int i : v) std::cout << i << ",";
Parallel Algorithms

// uses default executor: par
std::vector<double> d = { ... }
parallel::fill(par, begin(d), end(d), 0.0);

// rebind par to user-defined executor
my_executor my_exec = ...;
parallel::fill(par.on(my_exec), begin(d), end(d), 0.0);

// rebind par to user-defined executor and user defined executor parameters
my_params my_par = ...
parallel::fill(par.on(my_exec).with(my_par), begin(d), end(d), 0.0);
Extending Parallel Algorithms

Sean Parent: C++ Seasoning, Going Native 2013
Extending Parallel Algorithms

- New algorithm: gather

```cpp
template <typename BiIter, typename Pred>
pair<BiIter, BiIter> gather(BiIter f, BiIter l, BiIter p, Pred pred)
{
    BiIter it1 = stable_partition(f, p, not1(pred));
    BiIter it2 = stable_partition(p, l, pred);
    return make_pair(it1, it2);
}
```
Extending Parallel Algorithms

• New algorithm: gather_async

```cpp
template <typename BiIter, typename Pred>
future<pair<BiIter, BiIter>> gather_async(BiIter f, BiIter l, BiIter p, Pred pred)
{
    future<BiIter> f1 = parallel::stable_partition(par(task), f, p, not1(pred));
    future<BiIter> f2 = parallel::stable_partition(par(task), p, l, pred);
    return dataflow(
        unwrapped([](BiIter r1, BiIter r2) { return make_pair(r1, r2); }),
        f1, f2);
}
```
Extending Parallel Algorithms (await)

- New algorithm: gather_async

```cpp
template<typename BiIter, typename Pred>
future<pair<BiIter, BiIter>> gather_async(BiIter f, BiIter l, BiIter p, Pred pred)
{
    future<BiIter> f1 = parallel::stable_partition(par(task), f, p, not1(pred));
    future<BiIter> f2 = parallel::stable_partition(par(task), p, l, pred);
    return make_pair(await f1, await f2);
}
```
Fork-join Parallelism
Task blocks

- Canonic fork-join parallelism of independent and non-homogeneous code paths

```cpp
template <typename Func>
int traverse(node const& n, Func compute) {
    int left = 0, right = 0;

    define_task_block(
        policy,           // any (possibly rebound) execution policy
        [&](auto& tb) {
            if (n.left) tb.run([&] { left = traverse(*n.left, compute); });
            if (n.right) tb.run([&] { right = traverse(*n.right, compute); });
        });

    return compute(n) + left + right;
}
```
Two Examples
STREAM Benchmark

- Assess memory bandwidth
- Series of parallel for loops, 3 arrays (a, b, c)
  - copy step: \( c = a \)
  - scale step: \( b = k \times c \)
  - add two arrays: \( c = a + b \)
  - triad step: \( a = b + k \times c \)
- Best possible performance possible only if data is placed properly
  - Data has to be located in memory of NUMA-domain where thread runs
- OpenMP: implicitly by using ‘first touch’, i.e. run initialization and actual benchmark using same thread
  - #pragma omp parallel for schedule(static)
STREAM Benchmark: HPX

// create NUMA-aware executor, uses all cores of NUMA-domain zero
auto executor = numa_executor("numanode:0");

// create NUMA-aware allocator, uses executor for ‘first-touch’ initialization
auto allocator = numa_allocator(executor);

// create and initialize the three data arrays a, b, and c
std::vector<double, allocator> a(size, 0.0, allocator);
std::vector<double, allocator> b(size, 0.0, allocator);
std::vector<double, allocator> c(size, 0.0, allocator);
auto policy = par.on(executor);

parallel::copy(policy, begin(a), end(a), begin(c));

parallel::transform(policy, begin(c), end(c), begin(b),
    [k](double val) { return k * val; });

parallel::transform(policy, begin(a), end(a), begin(b), end(b), begin(c),
    [](double val1, double val2) { return val1 + val2; });

parallel::transform(policy, begin(b), end(b), begin(c), end(c), begin(a),
    [k](double val1, double val2) { return val1 + k * val2; });
STREAM Benchmark: HPX vs. OpenMP

TRIAD STREAM Results
(50 million data points)

Bandwidth [GB/s]
Number of cores per NUMA Domain
Matrix Transposition

An extended Example
Matrix Transposition

\[ B = A^T \Rightarrow \]

\[ B \quad = \quad A^T \]
Matrix Transposition

```cpp
void transpose(std::vector<double>& A, std::vector<double>& B)
{
    #pragma omp parallel for
    for (std::size_t i = 0; i != order; ++i)
        for (std::size_t j = 0; j != order; ++j)
            B[i + order * j] = A[j + order * i];
}

int main()
{
    std::vector<double> A(order * order);
    std::vector<double> B(order * order);

    transpose(A, B);
}
```
Matrix Transposition

```cpp
// parallel for
std::vector<double> A(order * order);
std::vector<double> B(order * order);
auto range = irange(0, order);
for_each(par, begin(range), end(range),
    [&](std::size_t i)
    {
        for (std::size_t j = 0; j != order; ++j)
        {
            B[i + order * j] = A[j + order * i];
        }
    });
```
Matrix Transposition (distributed)
Matrix Transposition (distributed)

```cpp
std::size_t my_id = hpx::get_locality_id();
std::size_t num_blocks = hpx::get_num_localities();
std::size_t block_order = order / num_blocks;

std::vector<block> A(num_blocks);
std::vector<block> B(num_blocks);
```
Matrix Transposition (distributed)

for (std::size_t b = 0; b != num_blocks; ++b) {
    if (b == my_id) {
        A[b] = block(block_order * order);
        B[b] = block(block_order * order);
        hpx::register_with_basename("A", A[b], b);
        hpx::register_with_basename("B", B[b], b);
    } else {
        A[b] = hpx::find_from_basename("A", b);
        B[b] = hpx::find_from_basename("B", b);
    }
}
Matrix Transposition (distributed)

B

phase

my_id

transpose

A

phase

my_id

Parallelism in C++, Hartmut Kaiser
Matrix Transposition (distributed)

```cpp
std::vector<future<void>> results;
auto range = irange(0, num_blocks);
for_each(seq, begin(range), end(range),
    [&](std::size_t phase) {
        future<block_data> f1 = A[phase].get_data(my_id, block_size);
        future<block_data> f2 = B[my_id].get_data(phase, block_size);
        results.push_back(hpx::dataflow(unwrapped(transpose), f1, f2));
    });
wait_all(results);```

Parallelism in C++, Hartmut Kaiser
Matrix Transposition (await)

```cpp
auto range = irange(0, num_blocks);
for_each(par, begin(range), end(range),
    [&](std::size_t phase)
    {
        future<block_data> f1 = A[phase].get_data(my_id, block_order);
        future<block_data> f2 = B[my_id].get_data(phase, block_order);
        transpose(await f1, await f2);
    });
```
Matrix Transpose: HPX vs. OpenMP

Matrix Transpose (SMP, 24kx24k Matrices)

- HPX (1 NUMA Domain)
- HPX (2 NUMA Domains)
- OMP (1 NUMA Domain)
- OMP (2 NUMA Domains)
Matrix Transpose: HPX vs. MPI (SMP)
Matrix Transpose: HPX vs. OpenMP (Xeon Phi)

Matrix Transpose (Xeon/Phi, 24kx24k matrices)

Data transfer rate [GB/s]

Number of cores
Real Application: Astrophysics, Hydrodynamics coupled with Gravity

![Graph showing time steps per second vs. number of cores for different cell counts and parallelisms.]
Conclusions

• Higher-level parallelization abstractions in C++:
  • uniform, versatile, and generic

• Not only possible, but necessary
  • Fork-join/loop-based parallelism: matching performance
  • New algorithms are not easily implementable using existing abstractions

• HPX code was identical for all benchmarks

• All of this is enabled by use of modern C++ facilities
  • On top of versatile runtime system (fine-grain, task-based schedulers)

• Shows great promise for distributed use cases
  • Parallel abstractions are not the cause for performance degradation
  • Insufficient quality of networking layer