Center for Computation & Technology

STE||AR GROUP

Distributed Object Abstraction in HPX

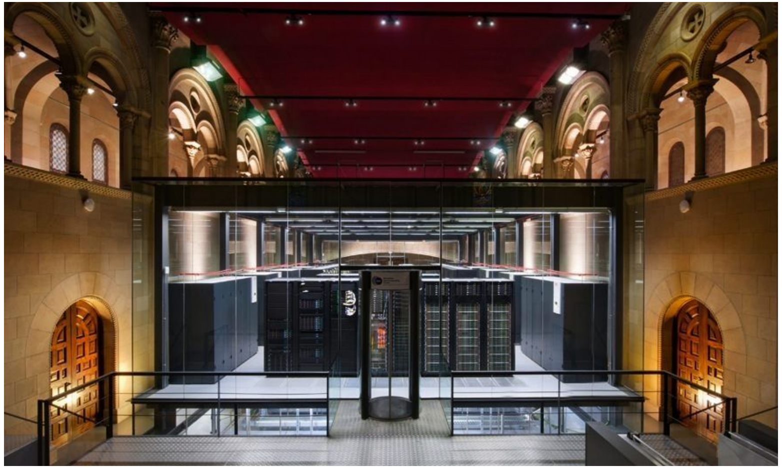
Weile Wei

May 6, 2019
CppNow
Aspen, CO

Background

Distributed Computing

- Important for solving large problems
  - Weather simulations
  - Astronomical simulations
  - Otherwise processing large amounts of data
- Requires communication
  - MPI (Message passing interface)
- Parallel Runtimes
  - HPX, Apache Hadoop, UPC/++



Barcelona Supercomputing Center
https://www.bsc.es/

Motivation

Provide high-level API mimicking STL containers for data access with minimal required awareness of distribution details

C++ Code

```
std::vector<double> vec(3, 42.0);
if( val[0] > threshold) {// Do something}
```

HPX code
in distributed setting

```
distributed_object<std::vector<double>> dist_vec
                    ("a_unique_str", std::vector<double> vec(3, 42.0));

if(dist_vec.fetch(from_remote).get()[0] > threshold)
      {// Do something }
```

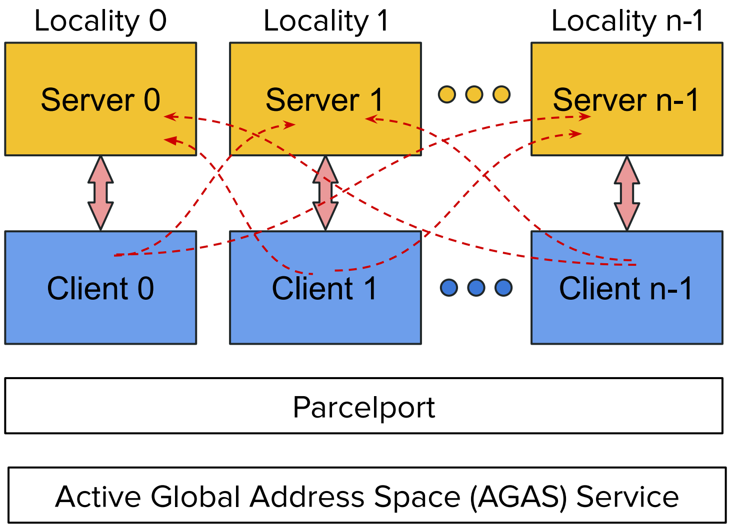
Background of HPX

- HPX
  - HPX is a C++ Standard Library for Concurrency and Parallelism.
- AGAS(Active Global Address Space):
  - AGAS exposes a single uniform address space spanning all localities an application runs on.
- Component:
  - A component is a C++ object which can be accessed remotely.
- Action:
  - An action is a function that can be invoked remotely.
- Distributed_object:
  - Each participated locality of the distributed_object has a local component (server) which has its own data. Each local component can invoke action on remote component through AGAS, however it requires each component to be registered in AGAS.

Implementation: distributed_object

- A single logical object partitioned over a set of localities/nodes/machines
- Every participating locality shares the same global name for the distributed object but owns its local data
- Local instance can obtain remote instances using fetch function within provided locality index
- Any C++ type can be made into a distributed object
- Inspired by UPC++'s dist_object API

distributed_object: Registration Methods All_to_All



distributed_object: Registration Methods All_to_All

- Allows distributed_objects to directly obtain references to instances on another locality.
  - Each distributed_object registers itself with AGAS using the basename given and the current locality id
  - Look-ups happen on an as-needed basis
  - Worst case N^2 lookups
  - Currently the template's default registration method

Distributed_object fetch() function

```
void add(distributed_object<int>& local, int& remote) {
        (*local) += remote;
}
//main function
distributed_object<int> dist_int("unique_name", cur_locality);
if (cur_locality == 0)
{
    std::vector<future<void>> results;
    auto range = irange(1, num_localities);
    for_each(seq, begin(range), end(range),
    [&](std::size_t remote_loc)
    {
        future<int> remote_val = dist_int.fetch(remote_loc);
        results.push_back(hpx::dataflow(unwrapped(add), dist_int, f1));
    });
    wait_all(results);
}
```

fetch() is an asynchronous function which returns a future of a copy of the instance of this distributed_object associated with the given locality index.

distributed_object for Subset of Localities

Allows for a disttributed_object to only be constructed on a specified subset of available localities. This may be useful when:

- Splitting workloads into constituent parts so relevant distributed_object is only used on a subset of localities
- Creating temporary structures which are only needed on a subset of localities for a given algorithm

```
// More than 2 localities
std::vector<size_t> participants{0, 1};

distributed_object<int> dist_int("dist_int"
                                , hpx::get_locality_id()
                                , participants);
```

Q&A

Distributed Object Abstraction in HPX

Weile Wei

May 6, 2019