LSU | Center for Computation & Technology

STE||AR GROUP

Distributed Object in HPX

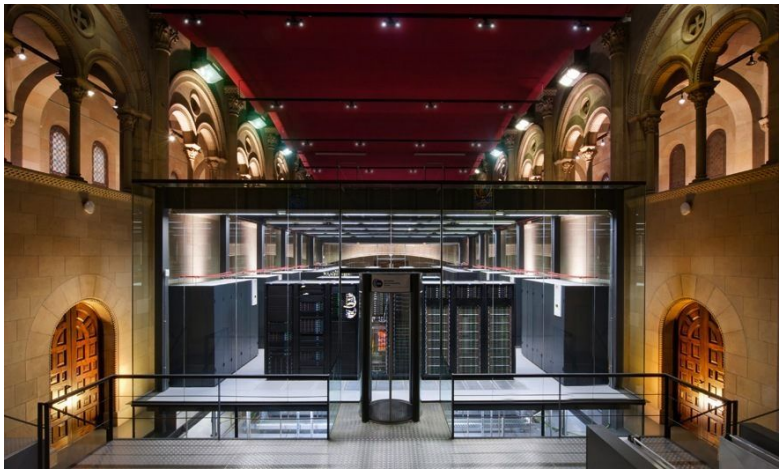
Weile Wei
Maxwell Reeser

April 25, 2019

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

Phylanx: An Asynchronous Distributed Array Computing Toolkit

- Distributed matrix operation (mul, transpose, etc.) requires
  - easy-to-use distributed container
  - user-friendly API
  - easy transfer of data between localities
  - reuse code as much as possible

Motivation

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

C++ Code

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

HPX code
in distributed setting

```
distributed_object<double> dist_val("a_unique_str", 42.0);

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

Motivation

MPI sample code for sending an integer

```
int number;
if (world_rank == 0) {
      number = -1;
      MPI_Send(&number, 1, MPI_INT,
           1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
      MPI_Recv(&number, 1, MPI_INT,
           0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Process 1 received number %d from
              process 0\n", number);
}
```

HPX code for fetching an integer

```
distributed_object<int> my_num(“example_int”,0);
if (hpx::find_here() == 0) {
     *my_num = -1;
} else if (hpx::find_here() == 1) {
     int fetched_val = my_num.fetch(0).get();
     printf("Process 1 received number %d from
             process 0\n", fetched_val);
}
```

http://mpitutorial.com/tutorials/mpi-send-and-receive/

Implementation: distributed_object

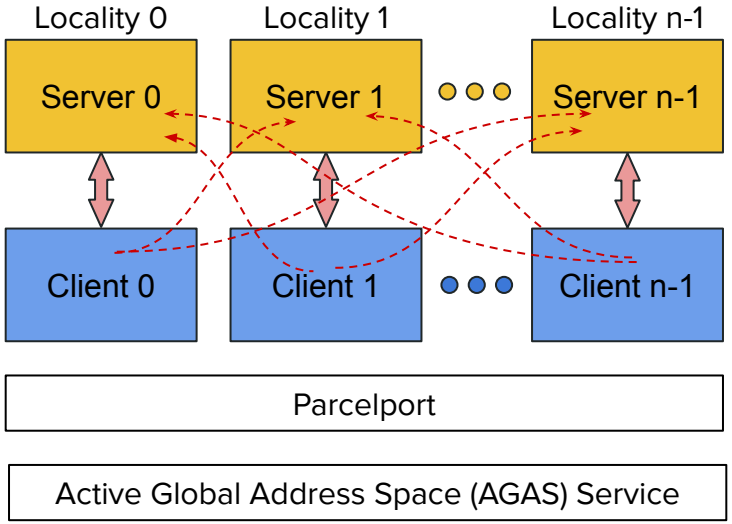
- A single logical object partitioned over a set of localities/nodes/machines
- Every participating locality shares the same global name for the distributed object but owns its local data
- Local instance can obtain remote instances using fetch function within given localities
- Any C++ type can be made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

Features

AGAS Registration

- 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.

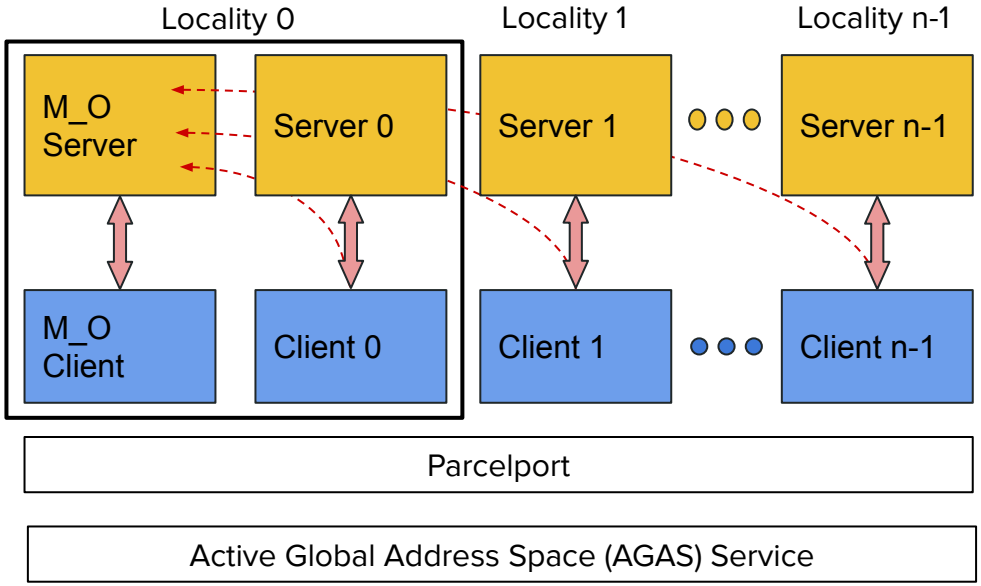
distributed_object: Registration Methods All_to_All



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: Registration Methods Meta_Object



distributed_object: Registration Methods Meta_Object

- Allows distributed_objects to obtain all references to other instances in one step
  - Each distributed_object registers itself with a Meta object using the basename given
  - Registration must complete before any operations on the distributed_object can take place
  - In every case, N lookups must be done, for each locality to find the Meta object
  - Slower on startup than All-to-All but a much lower upper-bound on messages sent
  - Specified as template parameter

```
distributed_object<int, construction_type::Meta_Object> my_num(“example/int”,0);
```

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);
barrier_wait_for_construction_all_localities();
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), f1, dist_int));
    });
    wait_all(results);
}
```

fetch() asynchronously 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 dist_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 (as in Phylanx)
- 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);
```

distributed_object: Template Specialization

```
class distributed_object<T&, C>
```

Template specialization allows wrapping an existing object

```
distributed_object(std::string base, data_type data,

        std::vector<size_t> sub_localities = all_localities())
```

distributed_object for Referencing an Object

```
std::vector<double> vec1(10, 42.0);

// wrap an existing object
distributed_object<std::vector<double> &> dist_vec("vec1", vec1);

vec1[0] = 41.0; // try to update original value


cout << dist_vec[0]; // 41 dist_vec will be updated, too!
```

Q&A

Distributed_object fetch() function

```
distributed_object<int> dist_int("unique_name", cur_locality);

barrier_wait_for_construction_all_localities();
// addition only computes in locality 0
if (cur_locality == 0)
{
    for (int i = 0; i < num_localities; i++)
    {
        (*dist_int) += dist_int.fetch(i).get();
    }
}
```

fetch() asynchronously returns a future of a copy of the instance of this distributed_object associated with the given locality index.