MPI RMA as a directory/cache interoperability layer

Nick Brown (EPCC) ; Tiberiu Rotaru and Bernd Lörwald (Fraunhofer ITWM) ; Vicenç Beltran and Xavier Teruel (BSC) ; Olivier Aumage (INRIA)

www.intertwine-project.eu
@intertwine_eu
INTERTWinE is funded by the EC
Grant agreement number 671602

INTERTWinE: Programming model interoperability

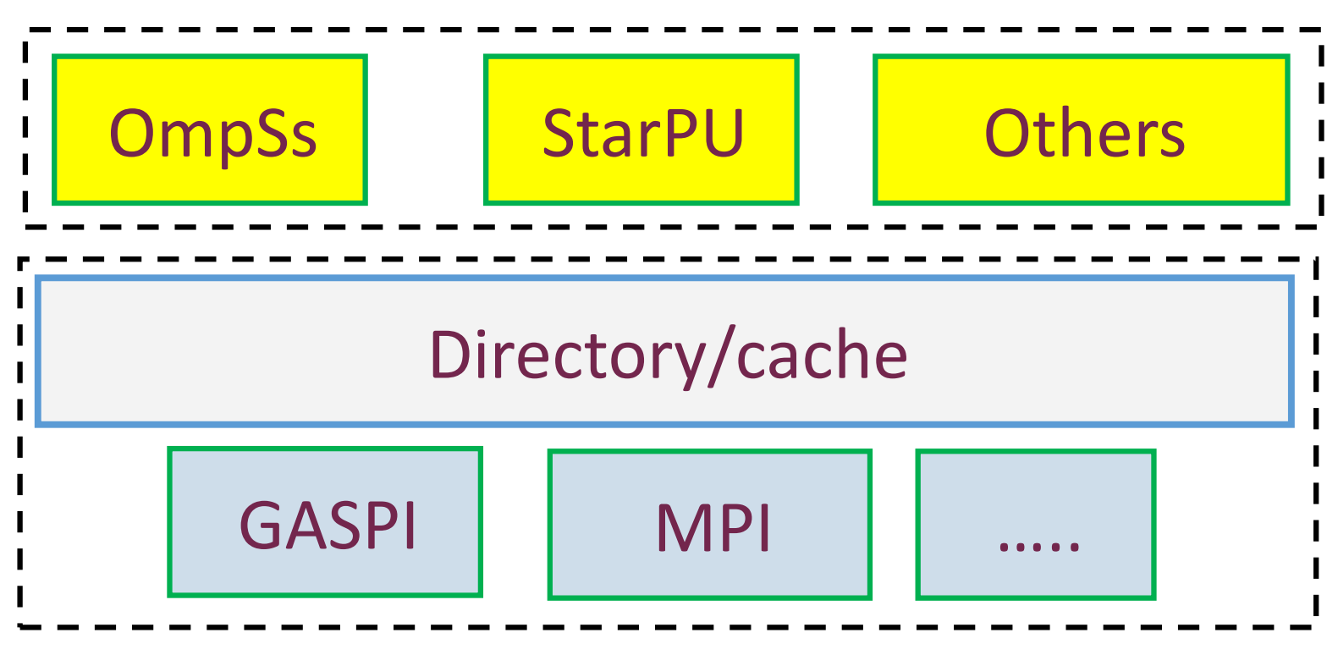
- Writing parallel codes is difficult, time consuming and the domain of a few experts
  - Numerous programming languages and models have been developed to counter this challenge, many of which specialise in specific paradigms such as task driven parallelism.
  - In order to reach exa-scale it is well accepted that we will need to address parallelism at many different levels, which inevitably means the mixing of programming models.
  - INTERTWinE— Enabling scalable and efficient programming model interoperability

Our challenge: Task based models are traditionally limited to a single memory space, how can we support these running over large scale distributed memory machines where a programmer's code is transparently interoperating with these models and distributed technologies such as MPI?

Directory/cache: Abstracting the transport layer

We do not intend the end programmer to interact directly with the directory/cache:

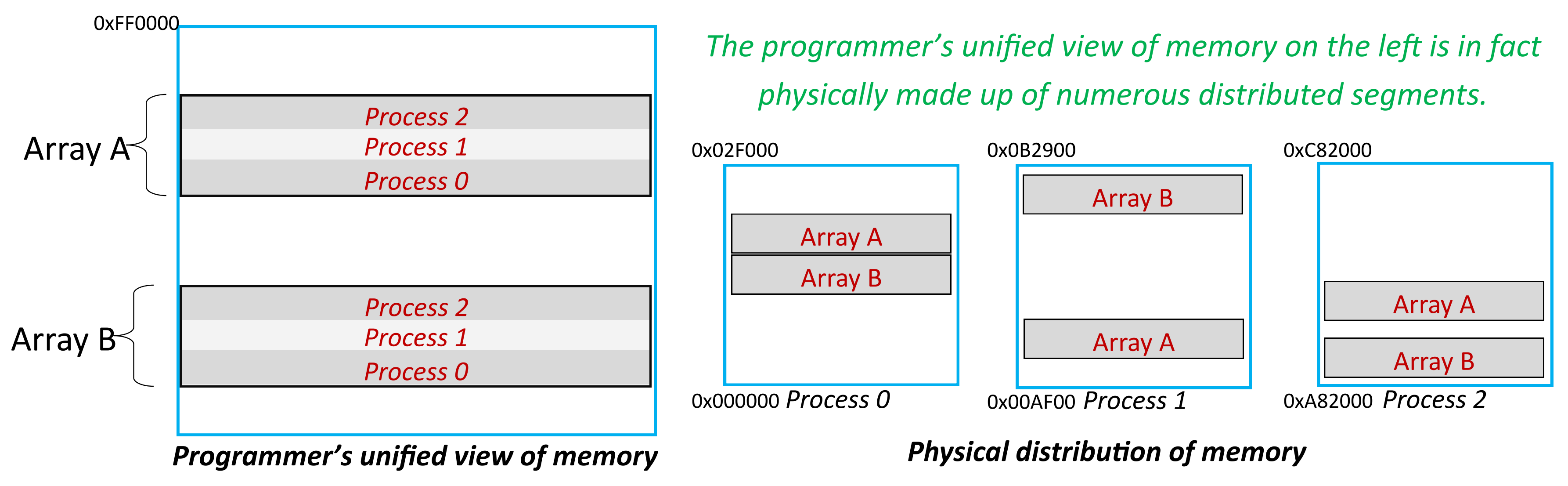
- The runtimes of the higher level programming models sit on top of the directory cache layer.
- These runtimes are independent of the physical distributed memory and technology (such as MPI RMA) used to support it.



These higher level runtime technologies interoperate with distributed memory using the abstraction of a single, unified, address space. The directory/cache supports seamless integration with many transport layers such as MPI, GASPI and BeeGFS which follow a unified interface.

Asynchronous requests are issued from the runtimes to the directory/cache which are served by the activation of threads from a pool. These requests can be tested for completion and waited upon.

Directory/cache: Distributed memory interoperability



The programmer's unified view of memory on the left is in fact physically made up of numerous distributed segments.

Achieving this relies on two crucial concepts:

Data directory: The programmer has a unified view of memory, whereas in reality it is split up into chunks which are distributed amongst the memory segments of different nodes. The directory tracks both what data is held where (known as the *home node*) and also versioning information.

Cache: Each process has it's own cache and, for performance reasons, remote data is transparently retrieved and then copied into the local cache before a pointer is returned and used in code like any normal local variable. Data in the cache is marked as either constant or mutable, cache coherence protocols ensure global consistency.

Memory segments

Segments of memory are defined and physically distributed amongst the processes. Different memory spaces such as GPU memory, MCDRAM or memory hierarchies can be represented by different segments and interact seamlessly. An abstraction of memory ranges is provided for data retrieval & writing. These ranges represent some chunk of memory in a segment and transparently span over multiple physically distributed allocations.

How we implement this: MPI RMA as a transport layer

Why choose MPI RMA as a transport layer?

- ✓ The ubiquity of MPI
- ✓ MPI's generally good, predictable, performance and scalability
- ✓ The directory/cache memory model maps well to MPI RMA (passive target synchronisation)

How do we utilise MPI RMA?

- Each segment is represented by its own MPI window
- Upon the initialisation of a segment, processes all issue *MPI_Win_lock_all* to start an access epoch with every other process in the window. The *MPI_MODE_NOCHECK* flag is used to avoid any consistency checking in the MPI library for performance reasons as we handle this at the transport layer.
- When a segment is destroyed the epoch is ended via *MPI_Win_unlock_all* and the window freed.

With remote data retrieval, for instance, a non-blocking request based read (*Rget*) is issued:

```
MPI_Win_Rget(source, num_elements, MPI_BYTE, homeNode, offset, num_elements,
             MPI_BYTE, segment_window, &request);
```

Retrieve num_elements bytes

The directory/cache's calling thread will sleep via a condition variable, this context is stored in a C++ map keyed by the request handle

From the home node

Starting at a specific offset

A separate thread periodically checks for completion of all outstanding requests. It does this by extracting out the MPI request handles into a *request_handles* vector and tests for the completion of any of these.

```
for (auto req : outstandingRMA) {
    request_handles.push_back(req.first);
    storedrequest_handles.push_back(req.first);
}
MPI_Test_some(num_requests, request_handles.data(),
              num_completed, completed_indexes, ……);
for (int i=0;i<num_completed;i++) {
    auto it = outstandingRMA.find(storedrequest_handles[
                                          completed_indexes[i]]);
    if (it != outstandingRMA.end()) {
        it->second->activate();
        outstandingRMA.erase(it);
    }
}
```

MPI_Test_some returns the number of completed requests and their input array indexes, but modifies completed requests handle to be *MPI_REQUEST_NULL*. Due to keying paused thread contexts by the MPI request, *storedrequest_handles* is also built and will remain unmodified during the *Test_some*. Completed request handles are retrieved from this vector to obtain the thread context from the *outstandingRMA* map for reactivation.

By default MPICH's asynchronous progress engine only checks for progress inside an MPI call (without helper threads.) The directory/cache sleeps rather than MPI barriers. We ensure that our thread issues MPI calls periodically by testing request handles, if there are no request handles to test then a dummy one is used.

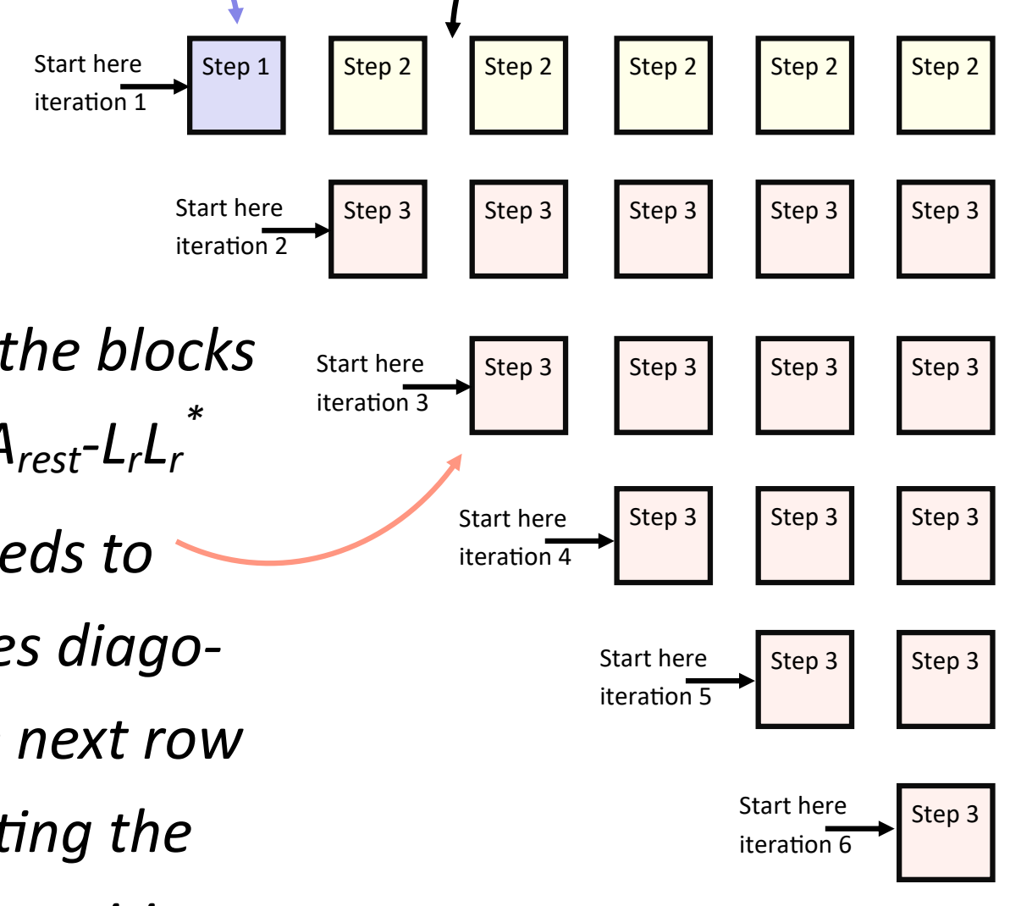
Cholesky matrix factorization benchmark

Step 1: Cholsky factorisation is performed for the current block, $A_b=L_bL_b^*$

Step 2: $L_{row}=A_{row}(L_b)^{-1}$ is computed for each block in the current row

Step 3: Update the rest of the blocks in the matrix with $A_{rest}=A_{rest}-L_rL_r^*$

The algorithm then proceeds to the next iteration & moves diagonally onto the block in the next row and next column, repeating the steps relative to this new position



Cholesky factorization decomposes a matrix (A) into its lower triangular matrix (L) and conjugate transpose (L*), where A=LL*. This has numerous applications in computational science including the solving of linear systems, Monte Carlo simulations and matrix inversion. The blocks of the matrix are distributed via the columns between processes.

This benchmark has been implemented using the directory/cache and compared against direct MPI RMA implementations

- Experiments performed on ARCHER, a Cray XC30, using Cray's implementation of MPI and the GNU 6.3 compiler
- Strong scaling, global size of 16384 by 16384 elements and a block size of 16 by 16 elements

Create a segment on each process which will hold distributed blocks of the matrix for Cholesky

Use MPI RMA as the transport layer

Evenly distribute the data (of length global_size) amongst the segments

```
segment_id_t data_segment(segment_create(global_size,
    mpi::equally_distributed_segment_description()));

data_id_t matrix_data(allocate(global_size, data_segment);
```

Physically allocate some data (of length global_size) amongst the individual segments of data_segment driven by the segment type. This represents physically allocating the space for the distributed matrix blocks in this benchmark.

Create a global range, it is transparent to the programmer if this spans multiple distributed memory spaces (segments)

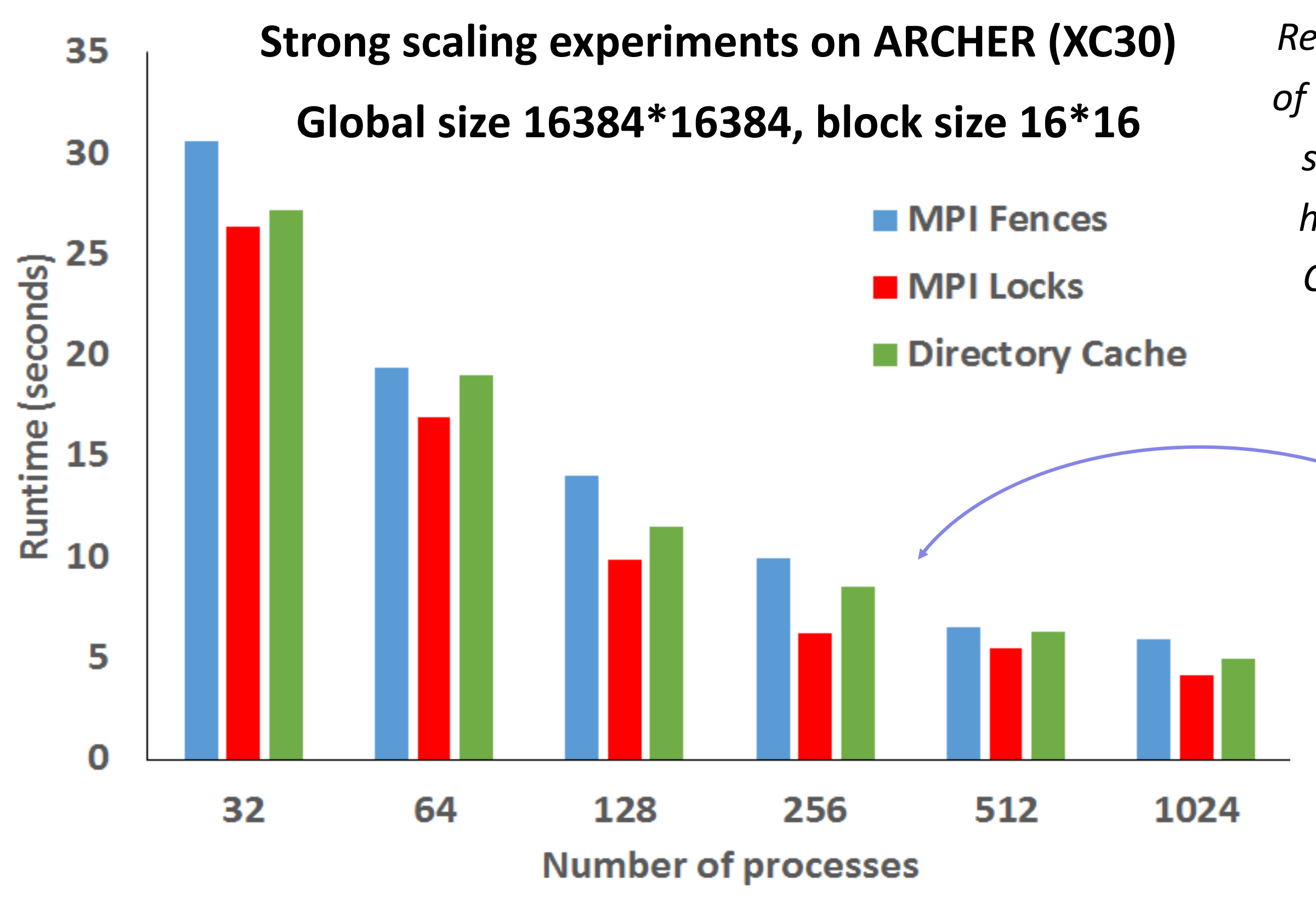
The range references some globally distributed data (via the id), starting offset and length

```
global_range_t global_range_toget={matrix_data,
    offset_t(data_start), b * b * sizeof (double)};

const_local_range_t local_data=execute_sync(
    get_const(global_range_toget, cache));
double * data = local_data.pointer();
…………

execute_sync(release(local_data));
```

Retrieve a read only copy of the data into the cache such as a matrix block held remotely for the Cholesky benchmark

Get the direct memory pointer of this retrieved read only data

Once processing is complete release the locally allocated cache memory



Performance & scalability sits between MPI fences and locks.

- *The directory/cache is built on top of MPI RMA locks and our layer adds some additional overhead in managing locality and cache*
- *But we present a higher level of abstraction to the programmer/runtime, greater flexibility (one can easily modify aspects such as the decomposition of data by changing segment type) and interoperability by using other transport layers seamlessly.*