Comparing One-Sided Communication with MPI, UPC and SHMEM

EPCC

University of Edinburgh

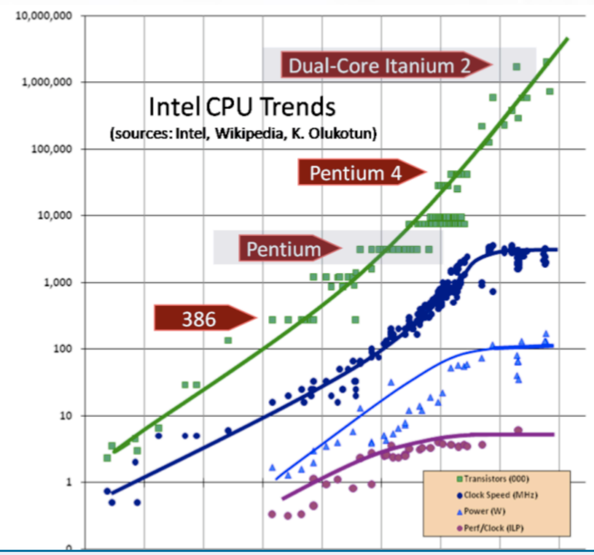
Dr Chris Maynard
Application Consultant, EPCC
c.maynard@ed.ac.uk
+44 131 650 5077

The Future ain't what it used to be

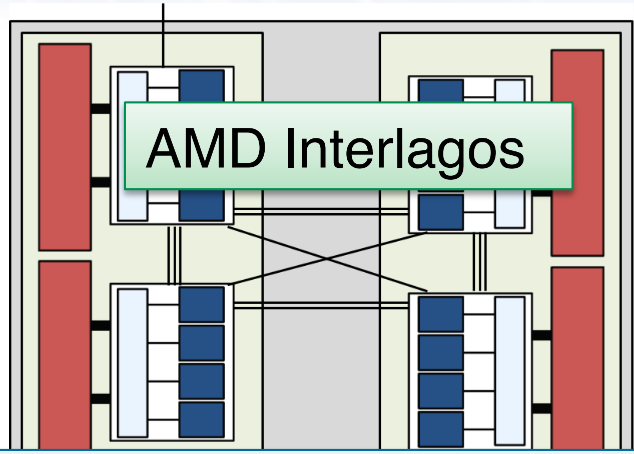
Past performance is not a guide to future performance. The value of the investment and the income deriving from it can go down as well as up and can't be guaranteed. You may get back less than you invested.

Parallel computing is changing

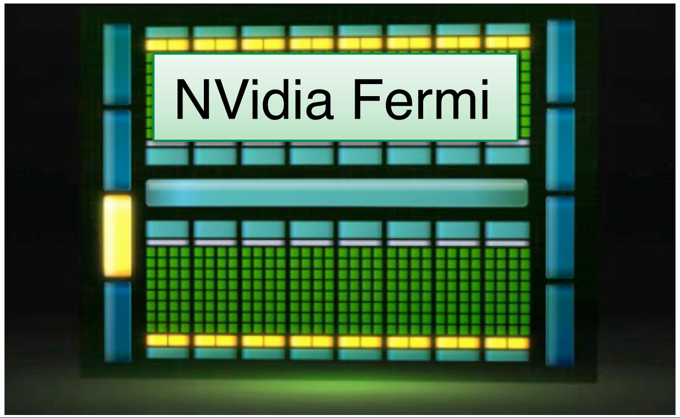
Parallel Programming just got harder!



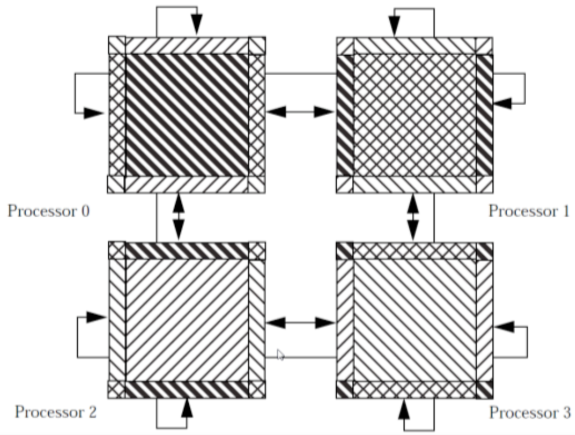
Moore's Law: More not faster



Some cores are more equal than others. NUMA



Heterogeneous Architectures: Accelerators



Data parallel: cores → MPI task

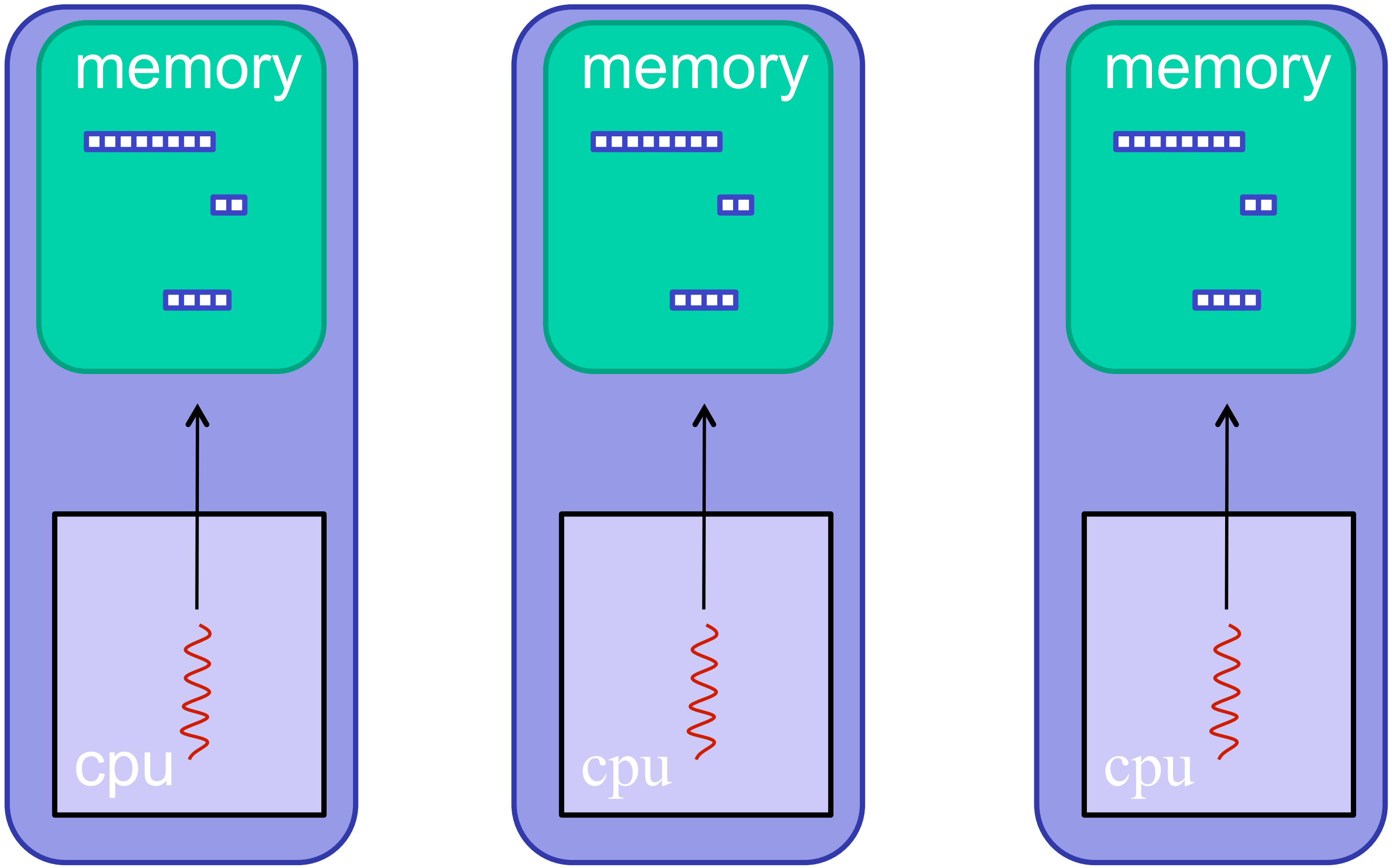
scale 2^{30} heterogeneous cores?

Partition Global Address Space (PGAS)

- Distributed memory is globally addressable - GAS
  - Partitioned into shared and local – P
  - Direct read/write access to remote memory
- Asynchronous put/get
  - remote node is not interrupted whilst memory access occurs
  - no explicit buffering
- May map directly onto hardware / Direct compiler support
  - Cray XE6 hdw and Cray compiler
- Language extensions
  - Unified Parallel C (UPC)
  - Co-Array Fortran (CAF)
- Libraries
  - SHMEM (Cray SHMEM)
  - One-sided MPI

How Super is PGAS?

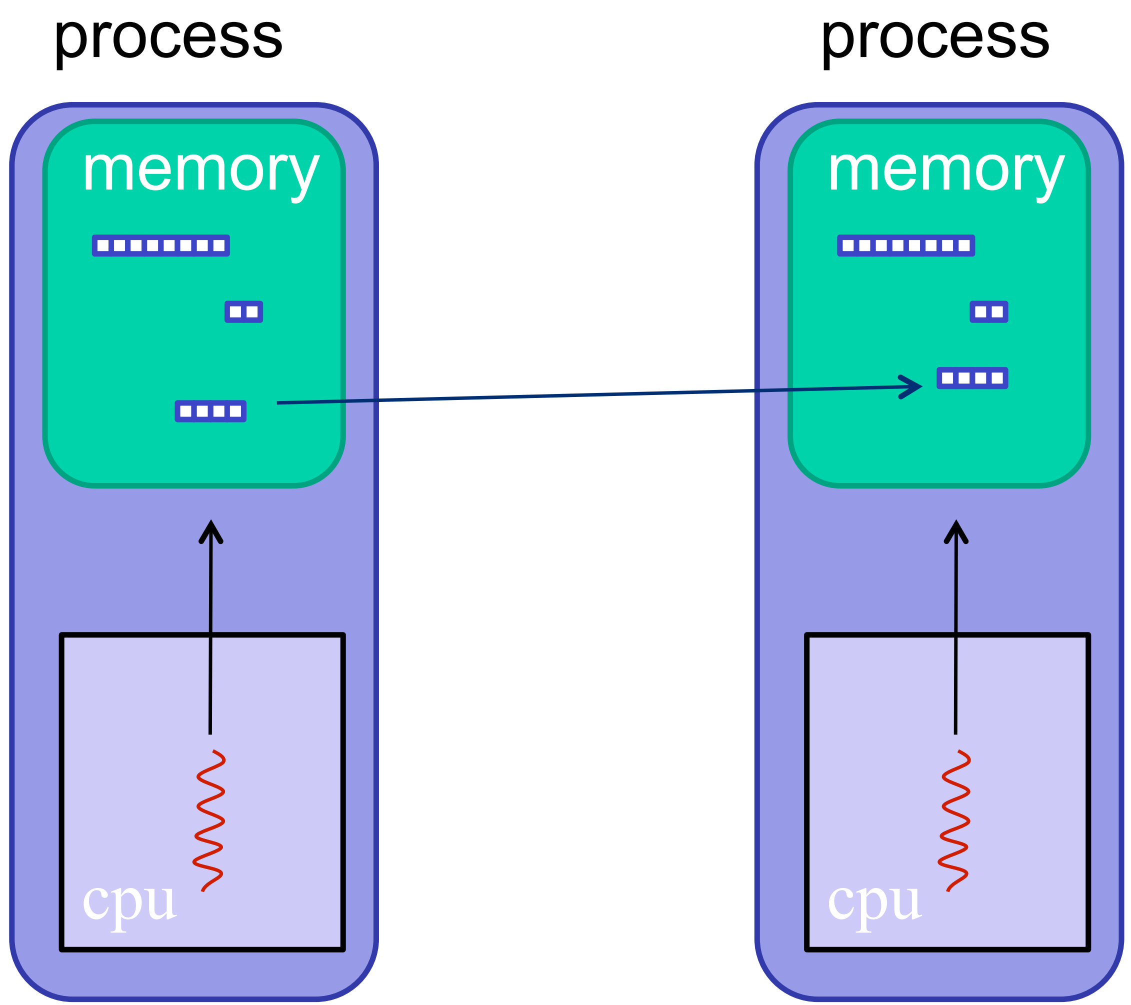
Distributed memory machine



MPI

- Mature, widely used and very portable
- Implemented as calls to an external library
  - Linked **`send`** and **`receive`** messages (both known to the other)
  - Collective calls, Broadcasts, gather/scatter, reductions, synchronisation
- One sided MPI calls in MPI 2 standard
  - Remote memory access (RMA)
  - **`puts`**, **`gets`** and synchronisation
  - Not widely used

MPI

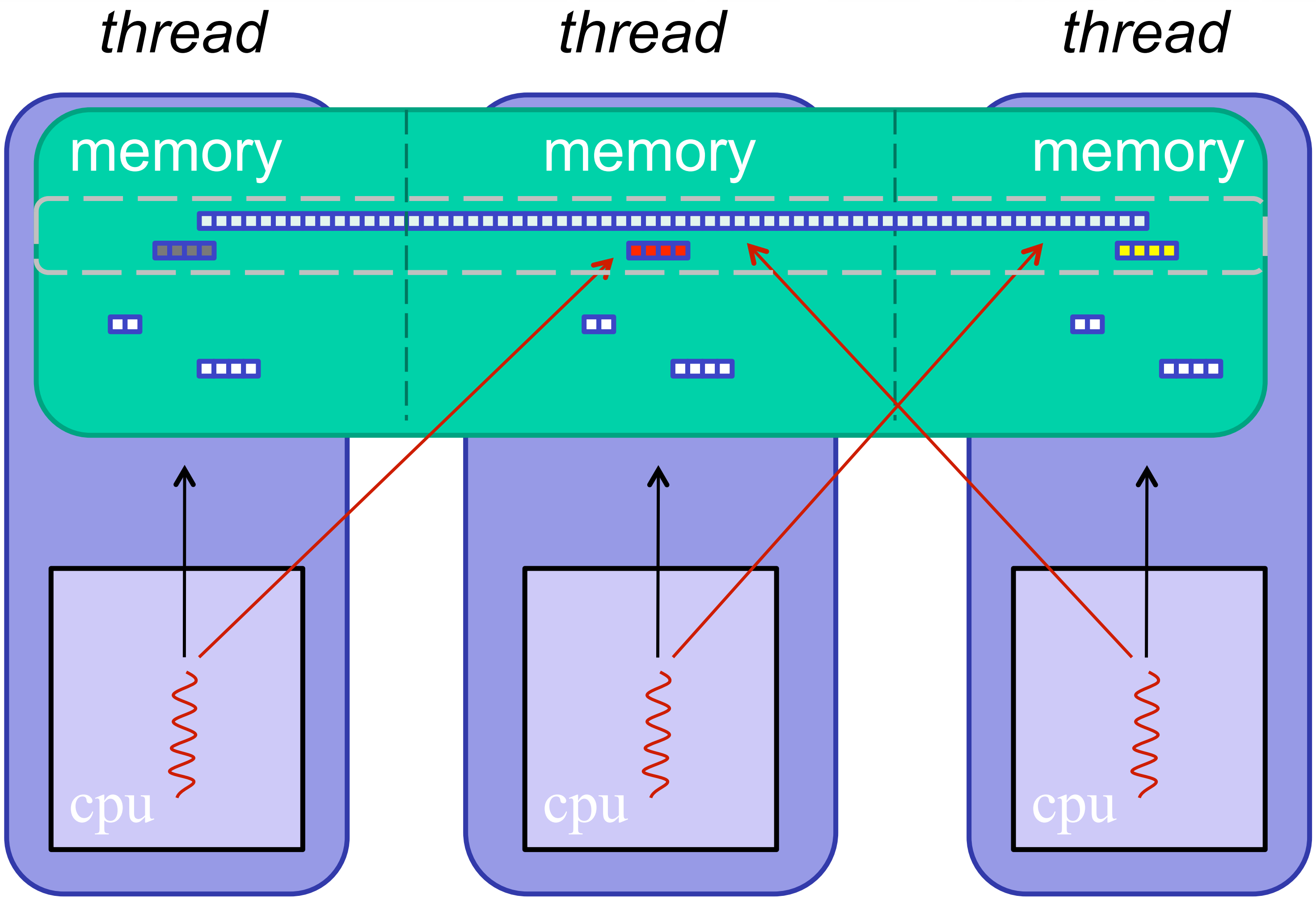


`MPI_Send(a,...,1,…)` `MPI_Recv(a,...,0,…)`

UPC

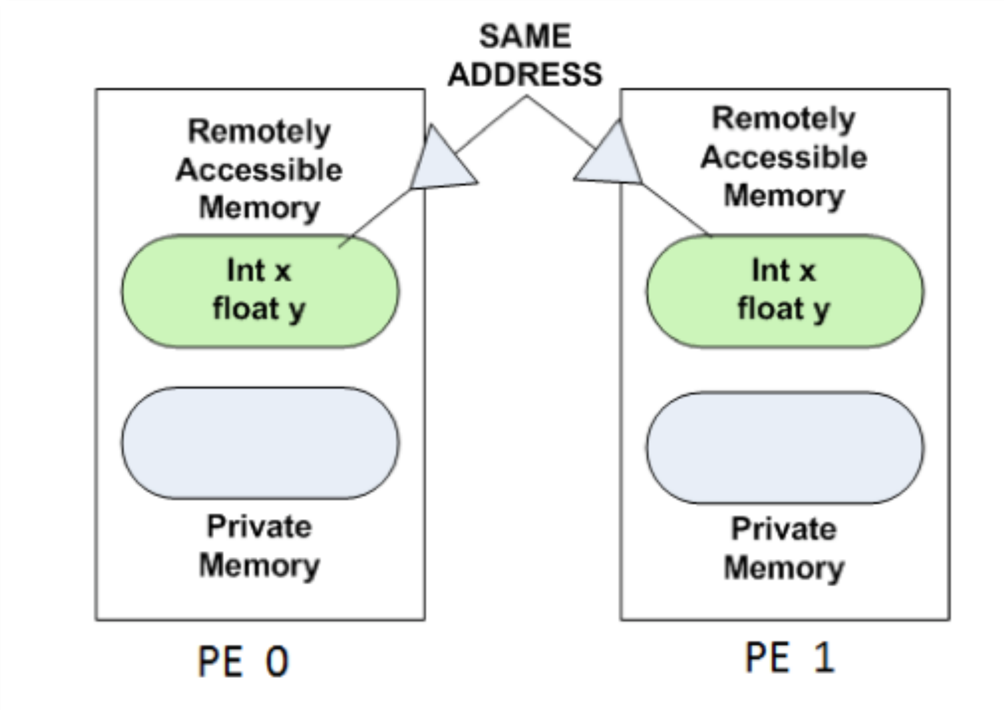
- Parallel extension to ISO C 99
  - multiple threads with shared and private memory
- Direct addressing of shared memory
- Synchronisation blocking and non-blocking `upc_barrier;`
- work sharing `upc_forall(exp;exp;exp;affinity);`
- private and shared pointers to private and shared objects
- Data distribution of shared arrays
- Cray compiler on XE6 supports UPC
- Portable Berkely UPC compiler
  - built with GCC

UPC



SHMEM

- Symmetric variables for RMA
- Same size, type and relative address on all PEs
- Non-stack variables, global or local static
- Dynamic allocation with shmalloc()
- shmem_put() and shmem_get routines for RMA
- Cray SHMEM
- OpenSHMEM
- Synchronisation
  - shmem_barrier_all();



The hash table code

- Simple C hash table code
  - supplied by Max Neunhoffer, St Andrews
- creates integer like objects
- computes the hash of the object
- populates the hash table, with a *pointer* to the object
- if entry already exists, inserts pointer at next free place
- revisits the hash table

Distributed hash table

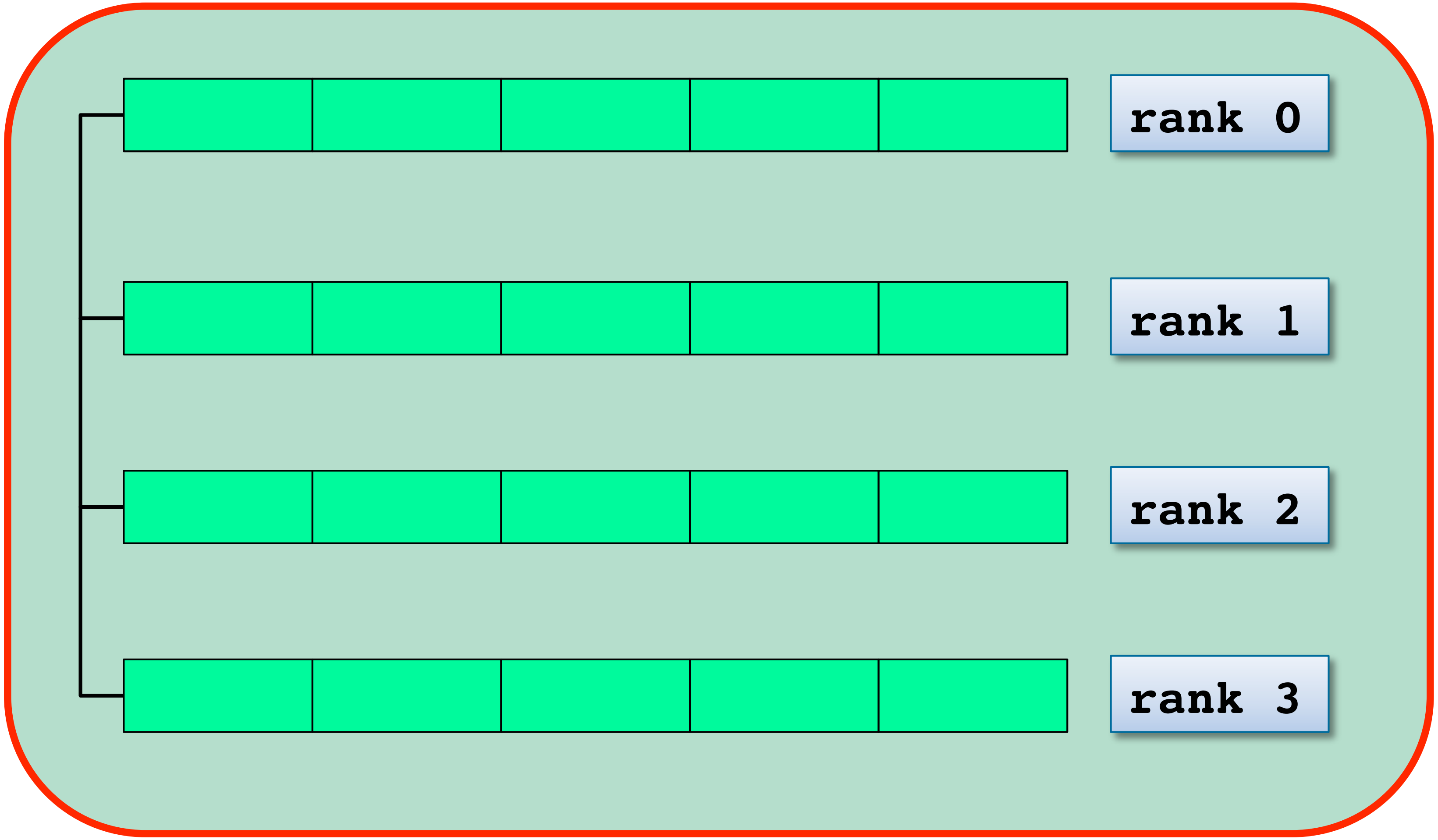
- The almost no computation in the test code
  - code is memory bound
- In parallel cannot use a local pointer to a remote object
  - distributed table has hold a copy of the *object itself*
  - Increases memory access cost compared to sequential code
  - UPC version shared pointer to shared object possible
    - consumes more shared memory, not considered
- RMA access cost much greater than direct memory access

Parallel code is slower than sequential code
More nodes → more communications

MPI hash table

- Declare memory structures on each MPI task
- link them together with **`MPI_Win_create()`**

```
MPI_Win win;
numb *hashtab;
numb nobj;
hashtab = (numb *)calloc(nobj,sizeof(numb));
MPI_Win_create(hashtab,nobj*sizeof(int),sizeof(numb),MPI_INFO_
NULL,comm,&win);
```



Window 0

MPI_Put, MPI_Get

- MPI ranks cannot see data on other ranks
- Data is accessed by MPI RMA calls to the data structure in the window

```
MPI_Get(&localHash,1,MPI_INT,destRank,destPos,1,MPI_INT,win);
```

origin_addr, count and type, where, count and type, Win

- Similarly for **`MPI_Put`**

Synchronisation

- Several mechanisms
  - MPI_Win_fence - barrier
- MPI_Win_lock, MPI_Win_unlock
  - ensure no other MPI_task can access data element
  - avoid race condition

```
MPI_Win_lock(MPI_LOCK_EXCLUSIVE,destRank,0,win);
```

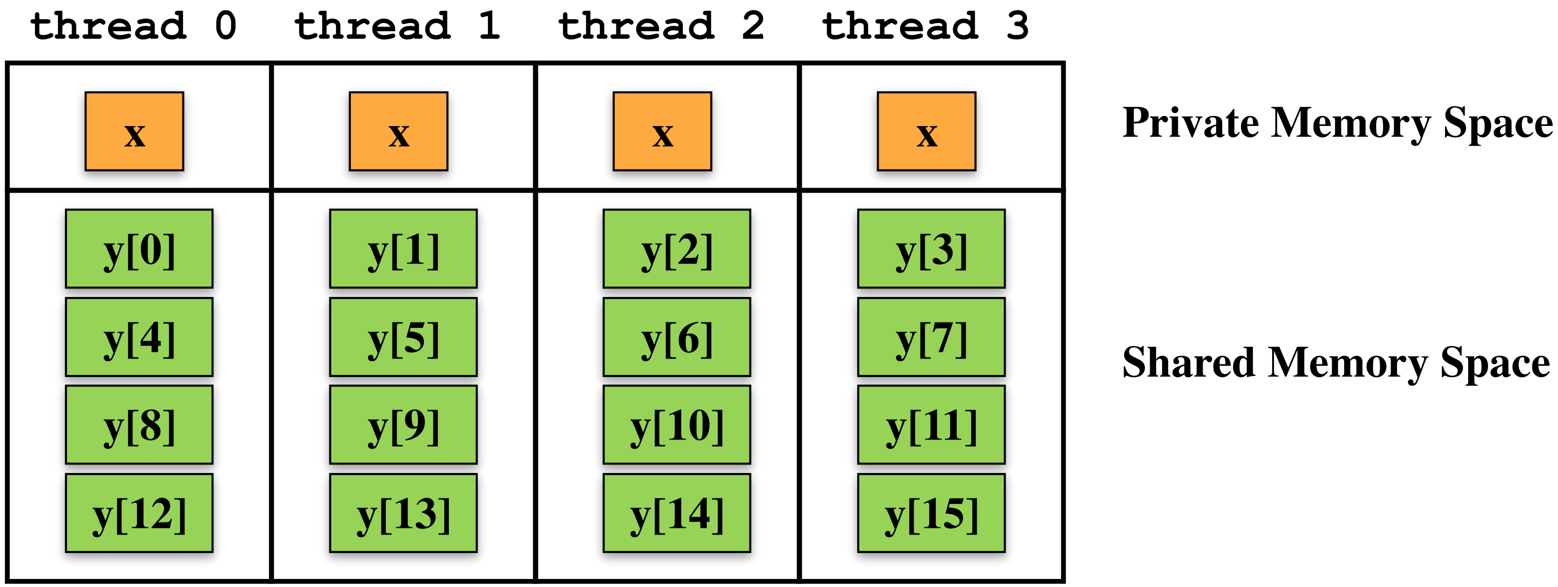
lock type, assertion, which rank

- Locks *all* the memory of specified MPI task in the specified window
- MPI_Win_unlock to release the lock

UPC: Shared arrays

- If shared variable is array, space allocated across shared memory space in cyclic fashion by default

```
int x;
shared int y[16];
```

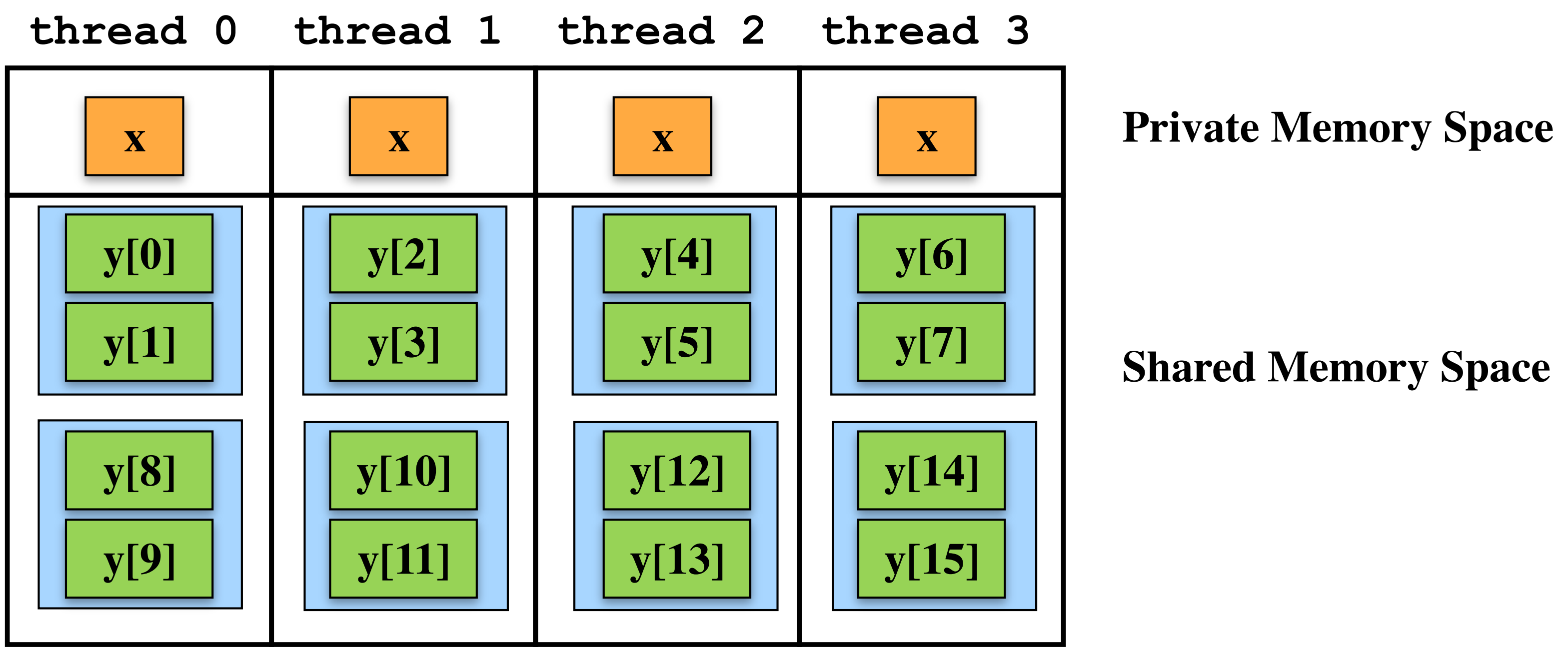


UPC: Data Distribution

- Possible to "block" shared arrays by defining

```
shared[blocksize] type array[n]
```

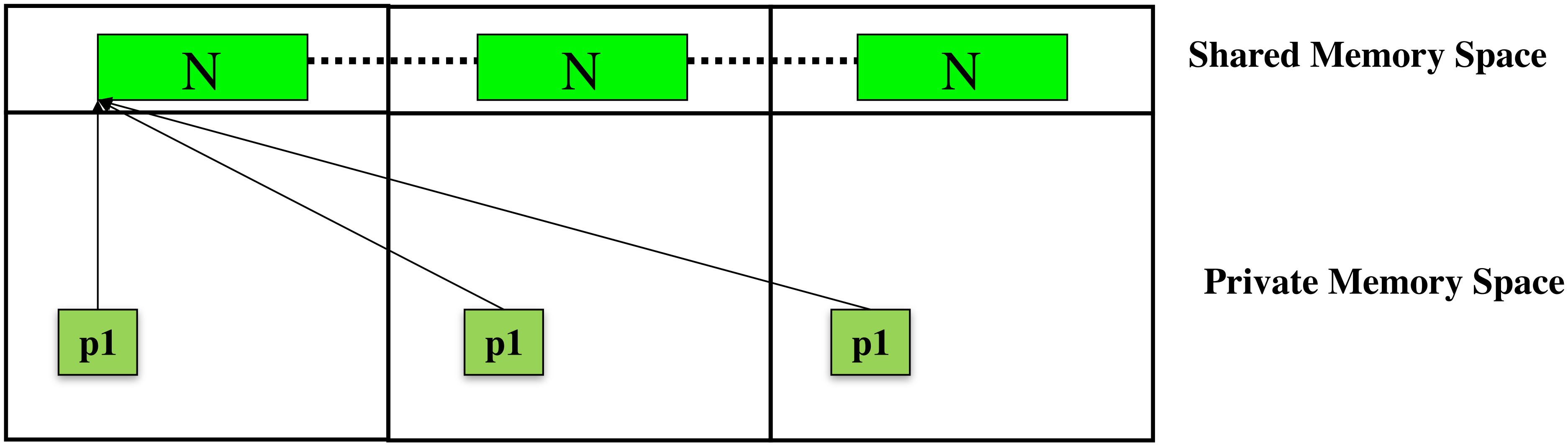
```
int x;
shared[2] int y[16];
```



Collective dynamic allocation

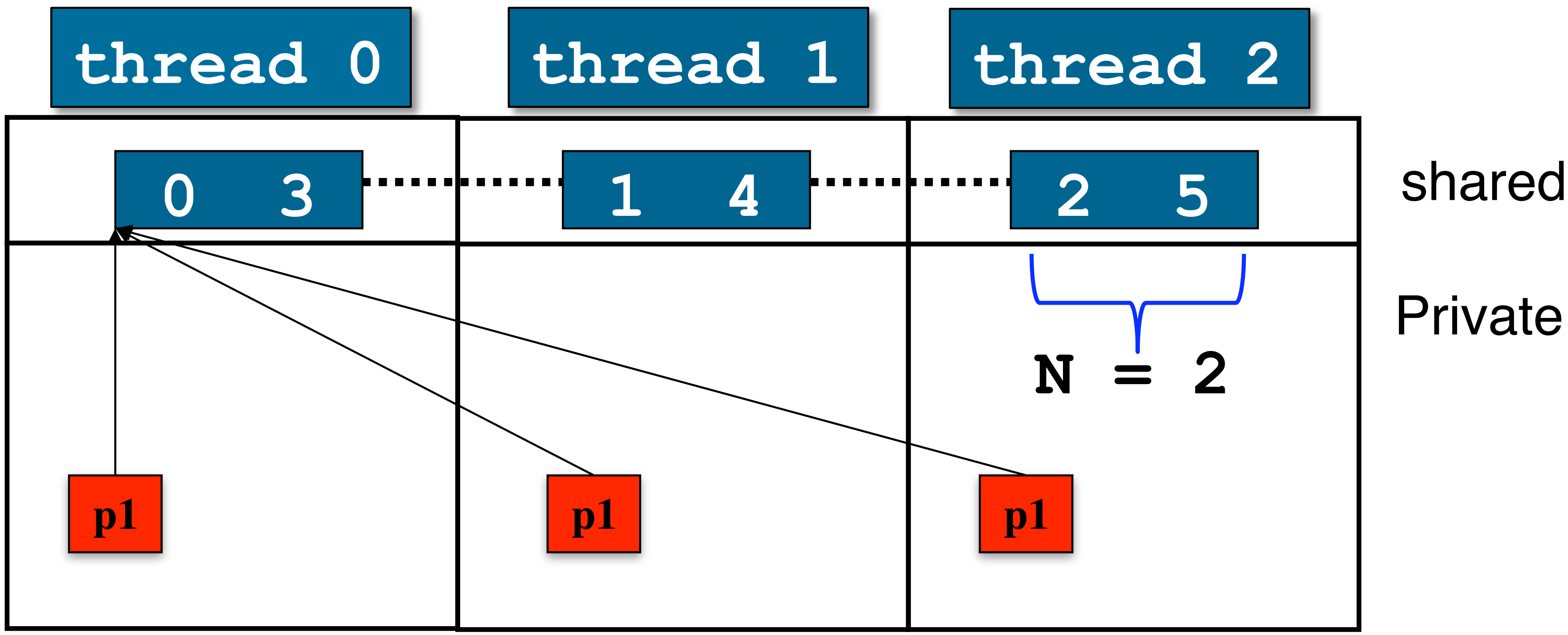
- The shared memory allocated is contiguous. Similar to an array

```
shared [B] int *ptr;
ptr = (shared [B] int *)upc_all_alloc(THREADS,
                                      N*sizeof(int));
```



Memory allocation and affinity

```
shared int *p1;
p1 = (shared int * )
upc_all_alloc(THREADS,N*sizeof(int));
```



UPC hash table

- Use upc pointer and collective memory allocation

```
shared numb *hashtab;
hashtab = (shared  numb *)
upc_all_alloc(THREADS,nobj*sizeof(numb));
```

- No blocking factor
  - cyclic distribution is as good as any other
  - if hash function is good enough
- Shared memory → all threads can see all data elements
- Use upc functions to determine which thread “own” data element

```
vthread = upc_threadof(&hashtab[v]);
```

UPC: Synchronisation

- As with MPI, use *locks* to control access to data
  - UPC declare array of locks
- Allocate memory – collective call
  - all locks would have affinity of thread 0
  - use follow trick to ensure distributed affinity

```
static upc_lock_t *shared lock[THREADS];

  for ( i=0; i<THREADS; ++i ) {
    upc_lock_t* temp = upc_all_lock_alloc();
    if ( upc_threadof( &lock[i] ) == MYTHREAD ) {
      lock[i] = temp;
    }
  }
```

- Array of locks can be any size
  - one, NTHREADS, NDATA, NDATA/NTHREADS
  - In this example NTHREADS Lock entire threads local data

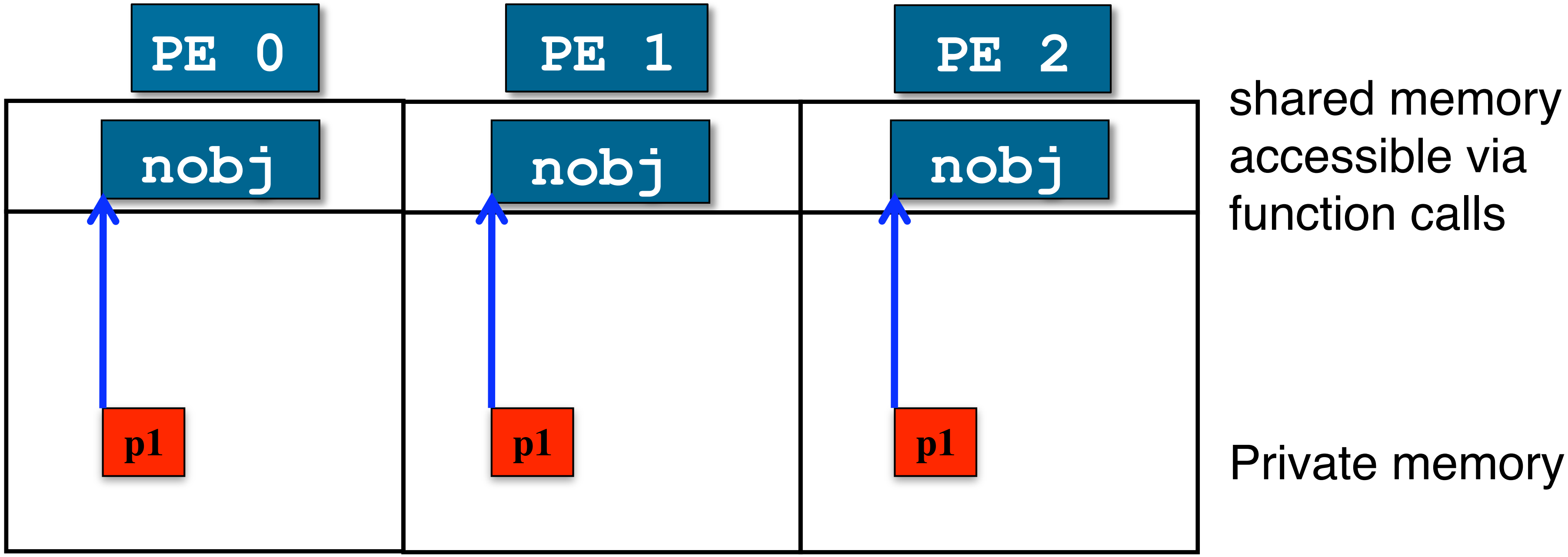
SHMEM: Memory

- Declare symmetric pointers with *file* scope

```
numb *hashtab;
```

- Allocate memory symmetrically
  - variables have the *same* address on different pes

```
hashtab = shmalloc(nobj*sizeof(numb));
```



SHMEM: RMA and synchronisation

- RMA is achieved by calls to SHMEM library

```
shmem_get(&localHash,&(hashtab[destpos]),1,destpe);
```

target source array position remote pe

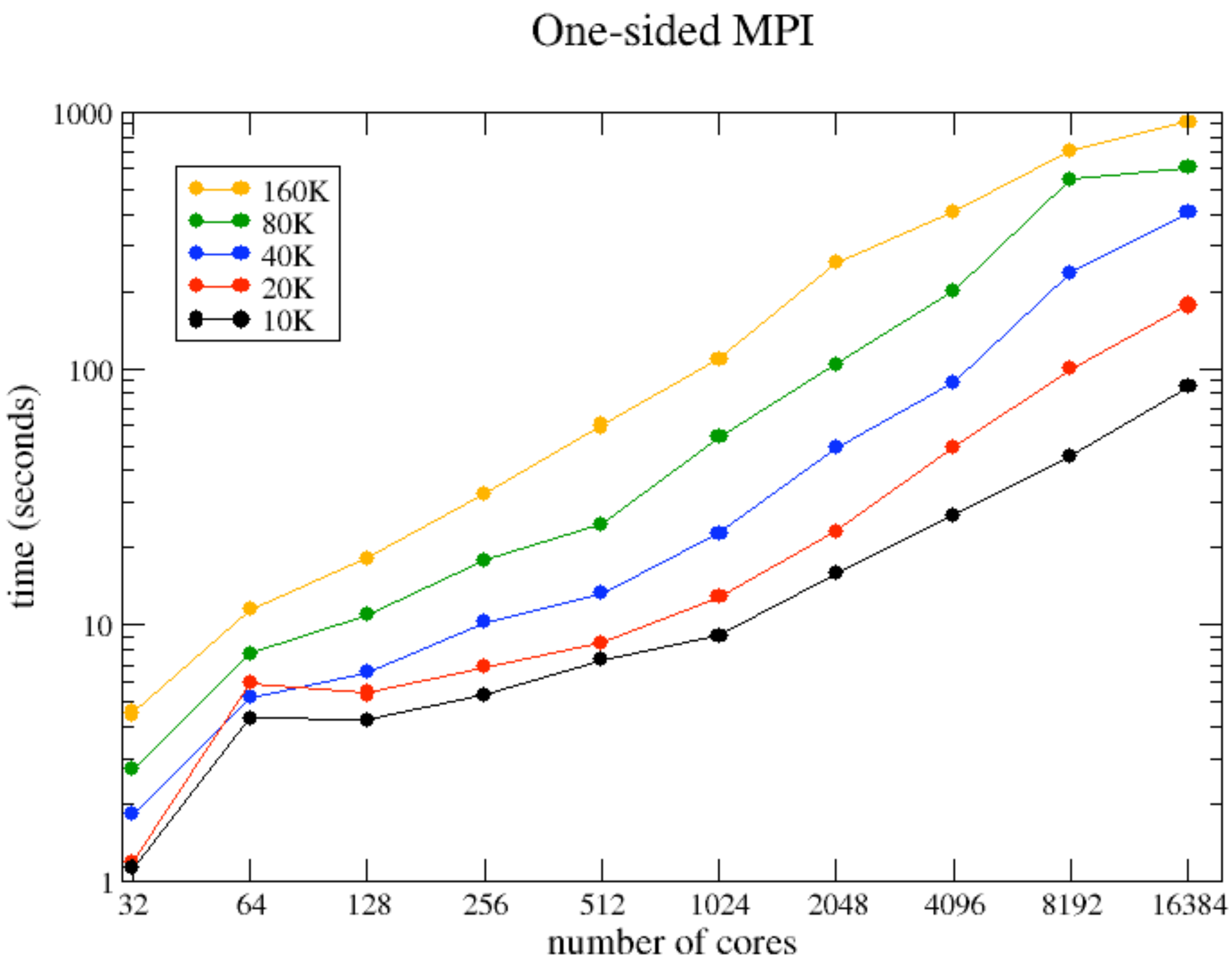
- Global synchonisation `shmem_barrier_all();`
- Control access to data elements with *locks*
- create array of locks of type long, size number pes
  - Locks must be symmetric
  - initialise (unlock) locks with `shmem_clear_lock(&lock[destpe])`
  - set lock with `shmem_set_lock(&lock[destpe])`

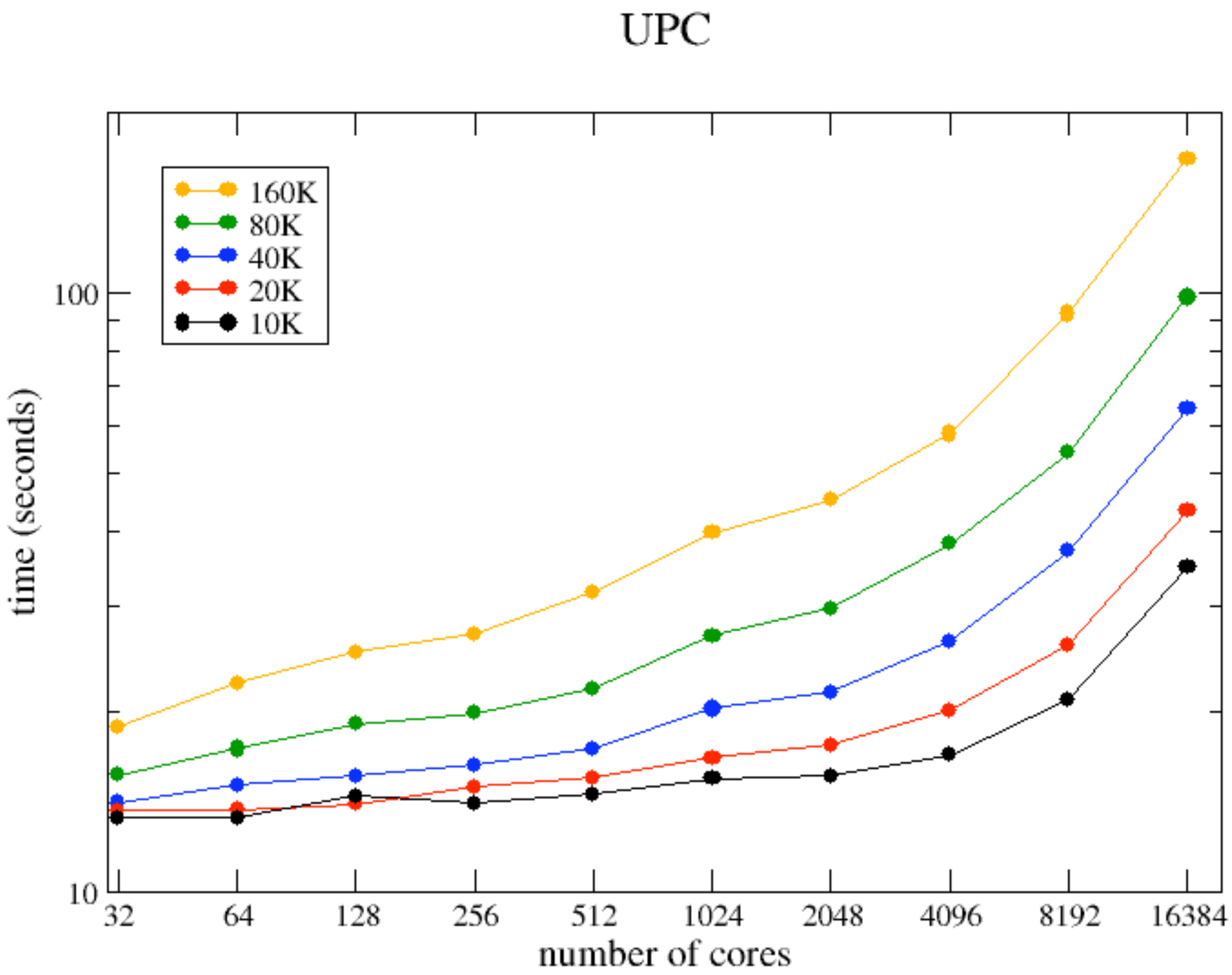
Hardware and Environment

- HECToR – UK National supercomputer service. Cray XE6
- All codes compiled with Cray C compiler (and Cray SHMEM)

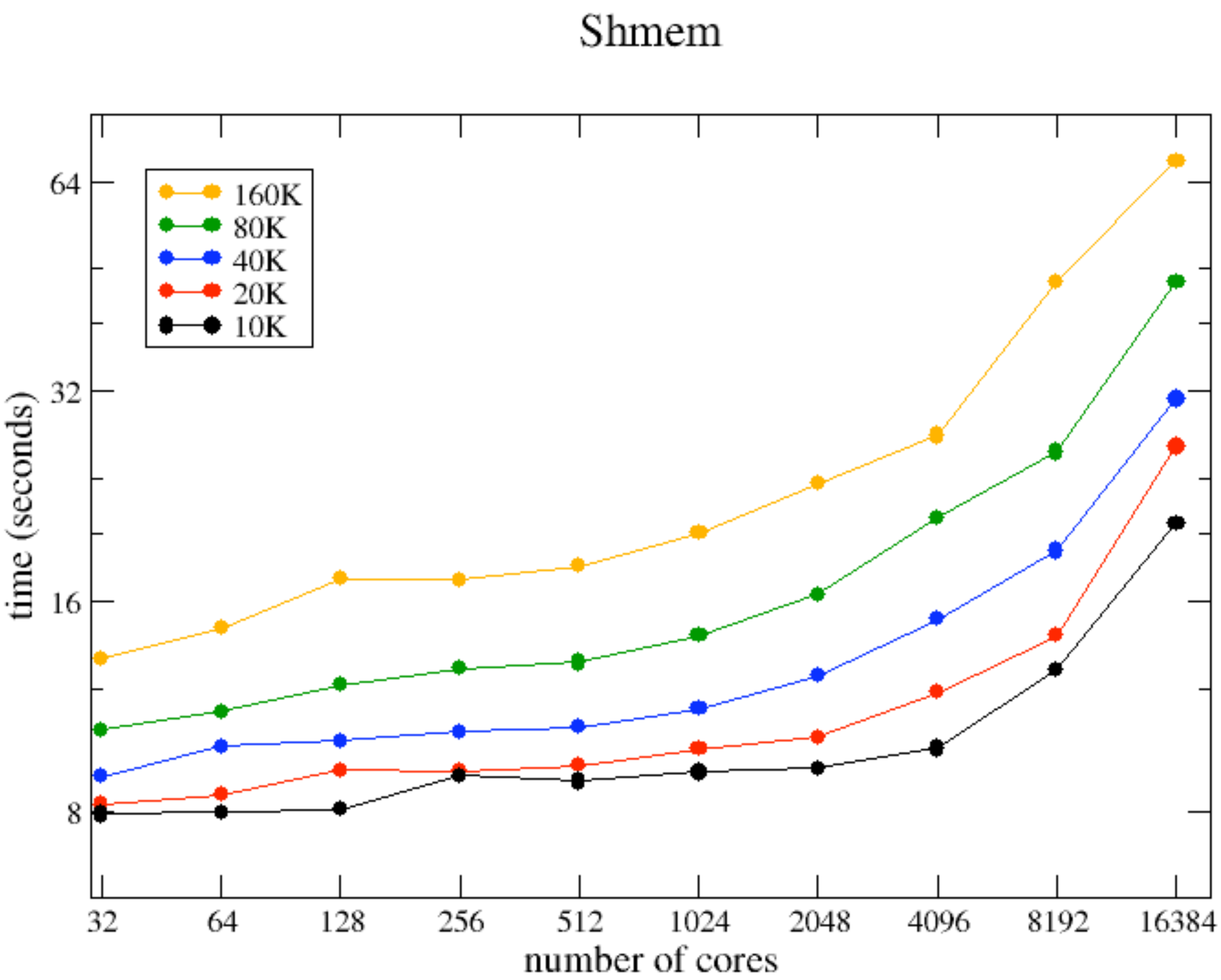
- benchmarks run on XE6 Gemini interconnect
  - Phase 3 AMD 32 core interlagos
  - Phase 2 AMD 24 core Magny-cours
- Integer object of 8 bytes with two passes of the hash table
- Weak scaling results show
  - wall clock time versus number of cores
  - fixed size of local hash table
  - different sizes are shown as different curves on the plots

MPI


One-sided MPI
160K
80K
40K
20K
10K
time (seconds)
1000
100
10
1
32
64
128
256
512
1024
2048
4096
8192
16384
number of cores

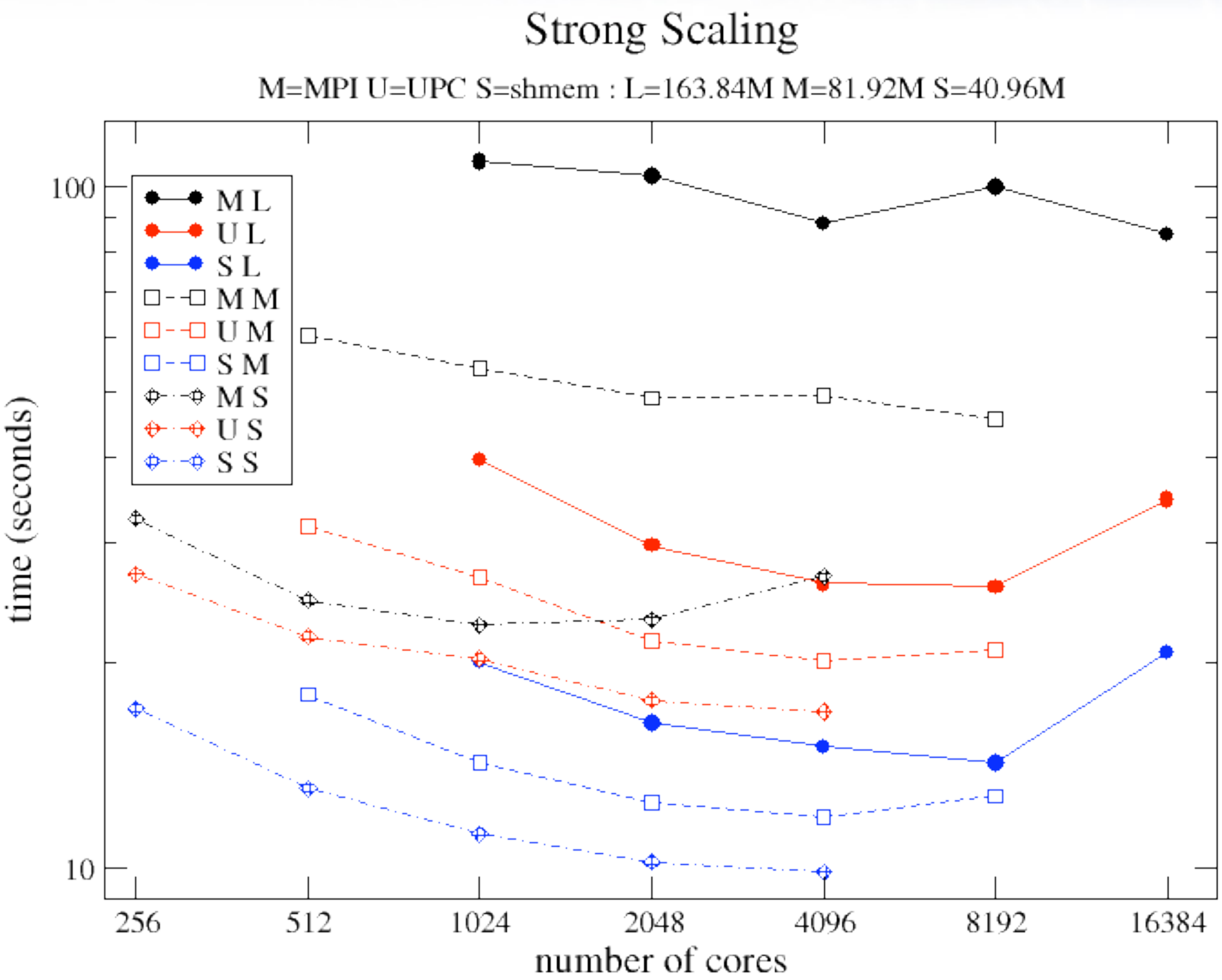

UPC
160K
80K
40K
20K
10K
100
10
time (seconds)
32
64
128
256
512
1024
2048
4096
8192
16384
number of cores

SHMEM

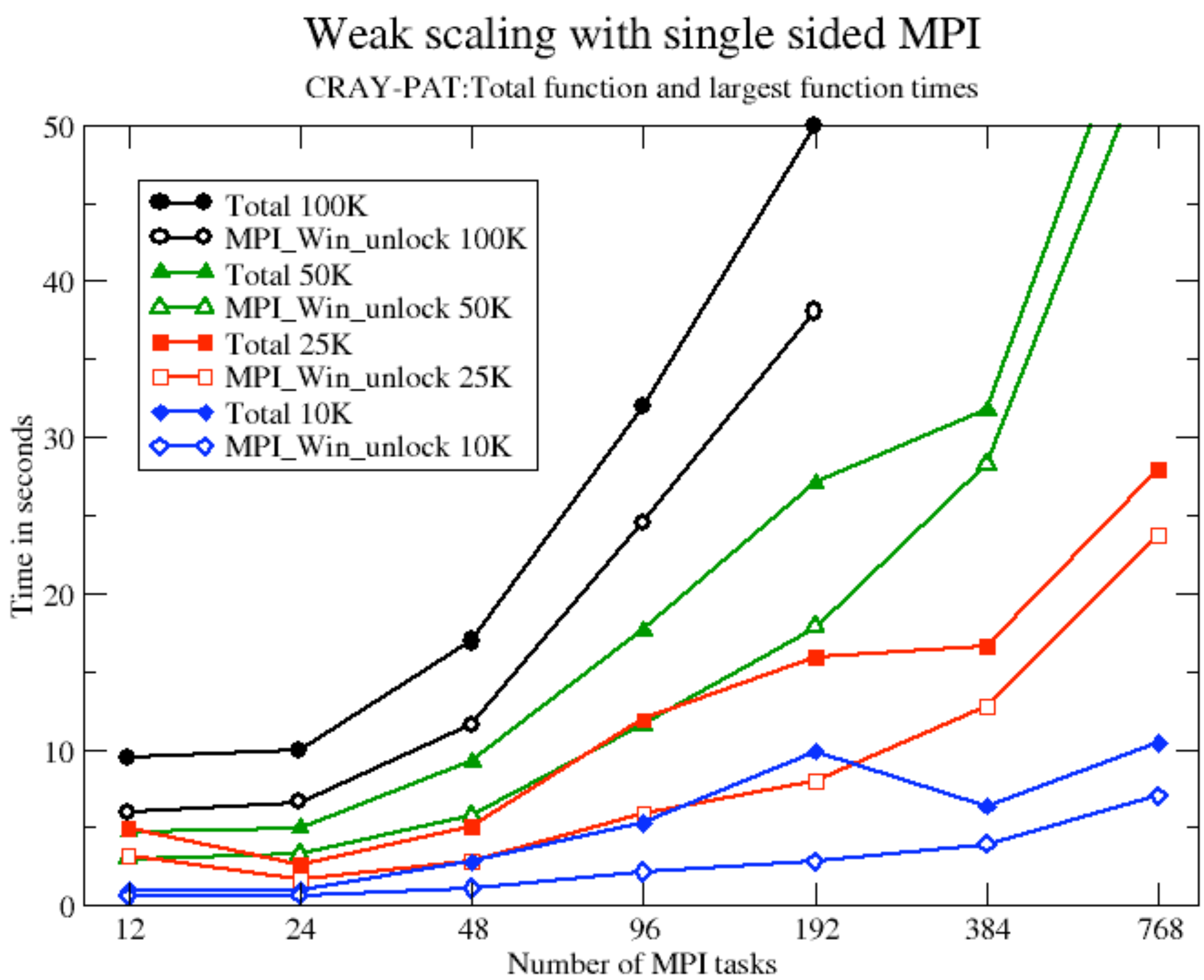


Shmem
160K
80K
40K
20K
10K
time (seconds)
64
32
16
8
32
64
128
256
512
1024
2048
4096
8192
16384
number of cores

Strong scaling



Profile MPI with Cray-PAT



MPI profile

- MPI_Win_unlock() taking significant amounts of time
- When does RMA occur?
- Not defined in standard
- MPI_Win_unlock() is the synchronisation point
  - all MPI RMA and synchronisation effectively occurring at that point
  - Hence function taking most time
- Profiling cannot resolve this further
- Nothing obviously wrong here
  - code
  - MPI implementation

One-sided MPI performance

- Why is it so bad?
- Other PGAS RMA models have memory restrictions
  - SHMEM “symmetric” variables
  - UPC static or global variables
  - CAF non-stack memory
- One-sided MPI widow can be constructed from any memory
  - This is hard to implement efficiently (apparently in MPICH2)
  - Likely cause poor performance

MPI 3

- Working group on RMA recognise this
- In MPI3 (July 2011) proposal new memory management call

```
MPI_WIN_ALLOCATE(size,disp_unit,info,comm,baseptr,win)
```

- "*Rationale*. By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access significantly. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes)."
- This is very welcome and should allow for better implementations

MPI 3

- However, the RMA working group lists 8 design goals …
- *1. In order to support RMA to arbitrary locations, no constraints on memory, such as symmetric allocation or collective window creation, can be required.*
- What is the rationale for this?
- This is the case in MPI-2
  - Difficult to implement this
  - poor performance
  - prevents take up by users
  - Failure

Harsh criticism
Speaker is unaware of any applications using one-sided MPI

SHMEM and UPC performance

- Both scale reasonably well with
  - size of hash table
  - number of cores
- Benchmark is effectively testing point-to-point, anywhere-to-anywhere communication
- SHMEM is faster than UPC
  - SHMEM maps directly to underlying dmapp protocol
  - UPC has more complicated mapping to underlying protocol and hardware

Conclusions

- Dominance of MPI send-receive communication pattern may be coming to an end
- PGAS languages/libraries are an alternative
  – can be simpler to program than MPI
  – How to code for Heterogeneous architecture?
- Compared one-sided MPI, UPC and SHMEM
  – One-sided MPI performs poorly
  – UPC and SHMEM perform well (on vender specific hdw and sfw)
- Many instances of PGAS
  – community uptake? Portability?
- Can Future MPI standard successfully adopt PGAS features?