

P2S2-2010 Panel
Is Hybrid Programming
a Bad Idea Whose Time Has Come ?

Taisuke Boku
Center for Computational Sciences
University of Tsukuba

Definition

- Term of “Hybrid Programming” sometime means “Hybrid Memory Programming” such as a combination of shared-memory and distributed-memory:
ex) MPI + OpenMP
- Term of “Heterogeneous Programming” sometime means “Hybrid Programming over Heterogeneous CPU Architecture” such as a combination of general purpose CPU and special purpose accelerator:
ex) C + CUDA
- In this panel, “Hybrid Programming” includes both meaning

Has the time of Hybrid Programming come ?

- Today's most typical hybrid architecture is "multi-core general CPU + (multiple) GPU", and on this architecture, we are doing hybrid programming such as C + CUDA, everyday
- Up to 10+ PFLOPS, it is OK to provide the performance with general-purpose CPU only (ex. Japan's "KEI" Computer, Sequoia or Blue Water), but beyond, it will be quite harder
- To prepare the upcoming days of 100 PFLOPS to 1 EFLOPS, we have to prepare because productive application programming requires a couple of years at least

Is it a good or thing to be accepted ?

- We have not been released yet from the curse of hybrid memory programming:
MPI + OpenMP is the most efficient way for current multi-core + multi-socket node architecture with interconnection network
- Regardless of the programmer's pain, we are forced to do it, and we need a strong model, language and tools to release these pains
- Issues to be considered
 - Memory hybridness (shared and distributed)
 - CPU hybridness (general and accelerator)
 - "flat" model is not a solution – we need to exploit the goodness of all these architecture as well as hybrid programming does

Necessity of overcoming memory hybridness

- Many of today's parallel applications are still not ready for memory hybridness
 - many of them are written only with MPI
- For really many cores such as 1M cores, it is impossible to continue MP-only programming
 - Increased cost for collective communication at least with $\log(P)$ order
 - Memory footprint cost to manage huge number of processes is not negligible while memory capacity per core is reducing
- It is relatively easy to apply automatic parallelization on hybrid memory architecture because such a huge parallelism must include multiple level of nested loops
 - Multi-level loop decomposition into memory hierarchy (and network hierarchy perhaps)

An example of effort

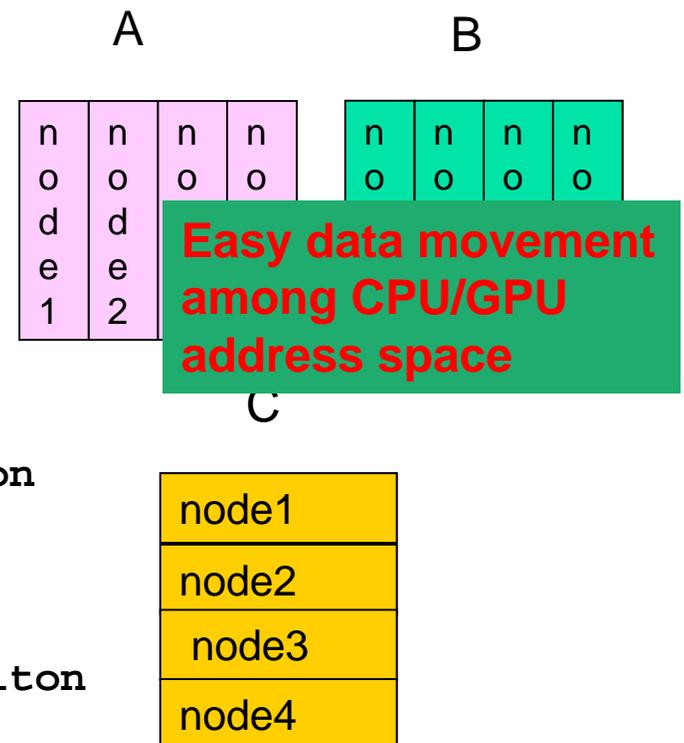
- Hybridness of CPU/GPU memory on a computation node
 - GPU is currently attached to CPU as a peripheral device as an I/O device with communication over PCI-E bus
 - It causes distributed memory (different address space) structure even on a single node
 - “Message Passing” in a node must be performed additionally to that among multiple nodes
- XcalableMP (XMP) programming language
 - Programming of large and multiple data array distributed over multiple computation node to be translated as local index access and message passing (similar to HPF)
 - Both “global view” (for easy access to a unified data image) and “local view” (for performance tuning) are provided and unified
 - Data movement in global view makes the data transfer among nodes as like as simple data assignment

gmove directive

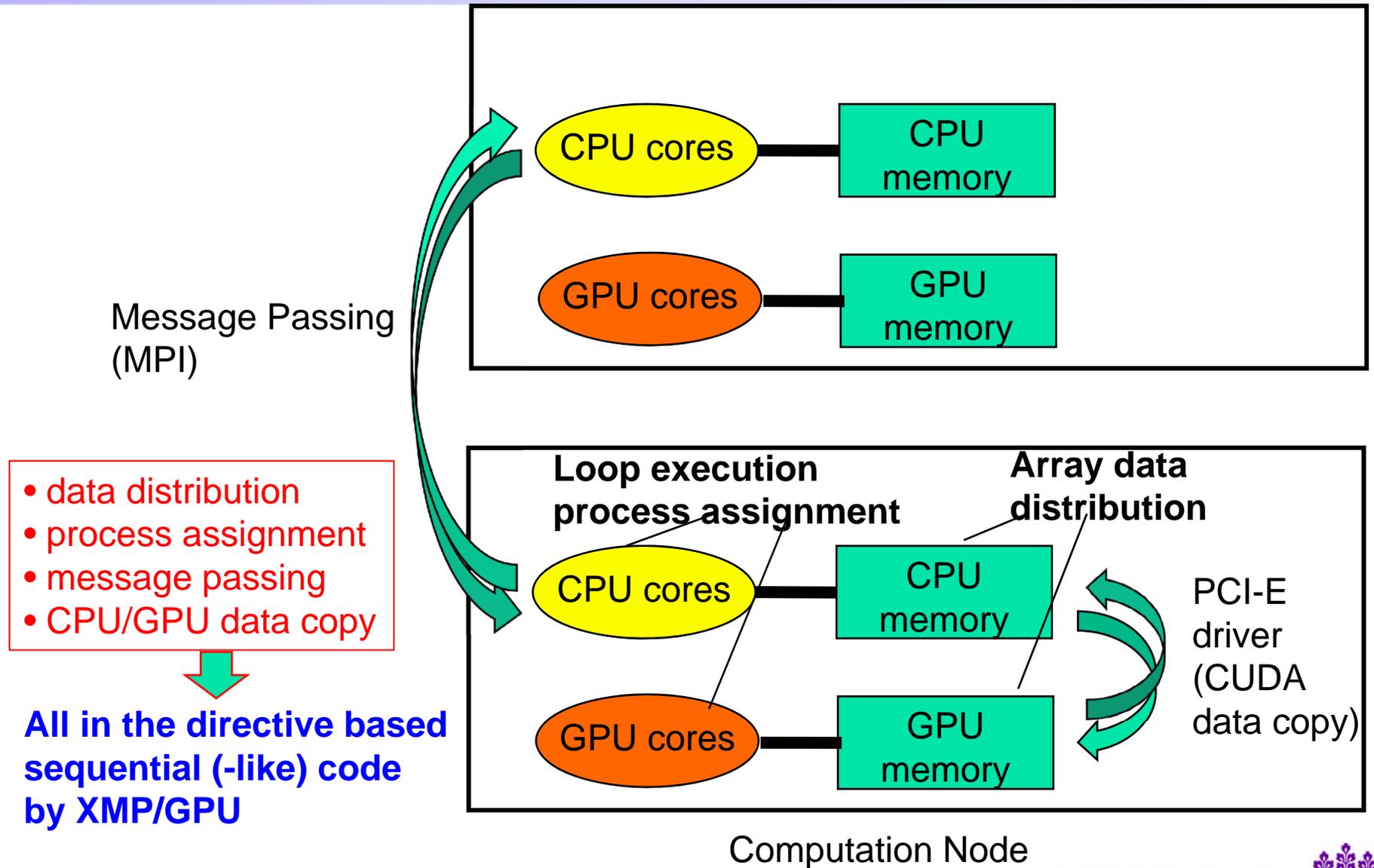
- The "gmove" construct copies data of distributed arrays in global-view.
 - When no option is specified, the copy operation is performed *collectively* by all nodes in the executing node set.
 - If an "in" or "out" clause is specified, the copy operation should be done by one-side communication ("get" and "put") for remote memory access.

```
!$xmp nodes p(*)
!$xmp template t(N)
!$xmp distributed t(block) to p
real A(N,N),B(N,N),C(N,N)
!$xmp align A(i,*), B(i,*),C(*,i) with t(i)

    A(1) = B(20)    // it may cause error
!$xmp gmove
    A(1:N-2,:) = B(2:N-1,:) // shift operation
!$xmp gmove
    C(:, :) = A(:, :)    // all-to-all
!$xmp gmove out
    X(1:10) = B(1:10,1) // done by put operaiton
```



CPU/GPU coordination data management



XMP/GPU image (dispatch to GPU)

```
#pragma xmp nodes p(*)           // node declaration
#pragma xmp nodes gpu g(*)      // GPU node declaration
...
#pragma xmp distribute AP() onto p(*) // data distribution
#pragma xmp distribute AG() onto g(*)
#pragma xmp align G[i] with AG[i]   // data alignment
#pragma amp align P[i] with AP[i]
int main(void) {
...
#pragma xmp gmove                // data movement by gmove (CPU⇒GPU)
    AG[:] = AP[:];
#pragma xmp loop on AG(i)
    for(i=0; ...)                 // computatio on GPU (passed to CUDA compiler)
        AG[i] = ...
#pragma xmp gmove                // data movement by gmove (GPU⇒CPU)
    AP[:] = AG[:];
}
```

What we need ?

- Unified easy programming language and tools with additional performance tuning feature is required
- At the first step of programming, easy import from sequential or traditionally parallel code is important
- Directive-base additional feature is useful to keep the basic construct of the language as well as the room of performance tuning
- How to specify a reasonable and effective standard directive to be applied for many of heterogeneous architectures ?