PARALLEL PROGRAMMING...

Parallel Programming: Overview

GOAL

Programming Interface for parallel computing

Hybrid = OpenMP (Open Multi-Processing) + MPI (Message Passing Interface)

병렬 컴퓨팅을 위한 프로그래밍
인터페이스

Hybrid MPI and OpenMP

Hybrid application programs using **MPI + OpenMP** are now common place on large HPC systems.

There are basically two main motivations:

1. Reduced memory footprint, both in the application and in the MPI library (eg communication buffers).
2. Improved performance, especially at high core counts where pure MPI scalability runs out.

Hybrid MPI and OpenMP

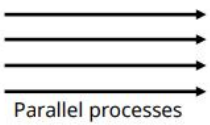
Parallel programming models

Parallel execution is based on threads or processes (or both) which run at the same time on different CPU cores

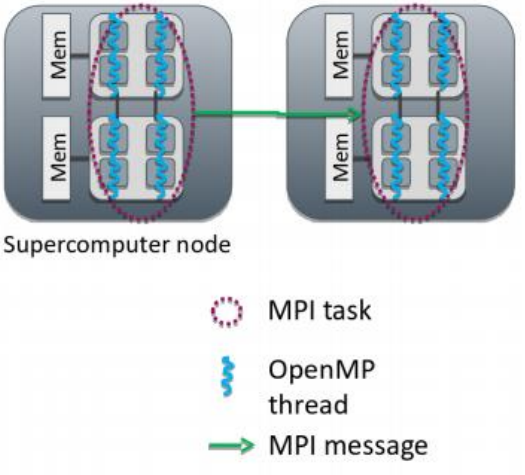
Processes
- Interaction is based on exchanging messages between processes
- MPI (Message passing interface)

Threads
- Interaction is based on shared memory, i.e. each thread can access directly other threads data
- OpenMP

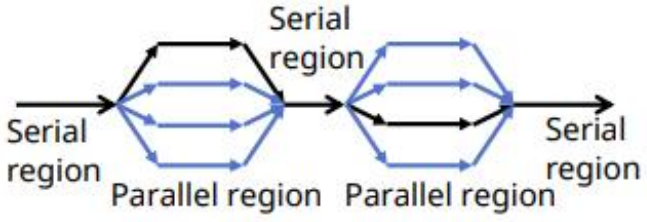
Hybrid MPI and OpenMP



1: MPI: Processes
- Independent execution units
- Have their own memory space
- MPI launches N processes at application startup
- Works over multiple nodes



2: OpenMP: Threads
- Threads share memory space
- Threads are created and destroyed (parallel regions)
- Limited to a single node



3: Hybrid programming: Launch threads (OpenMP) within processes (MPI)

- Shared memory programming inside a node, message passing between nodes
- Optimum MPI task per node ratio depends on the application and should always be experimented.

The Best From Both Worlds

- **MPI** allows for inter-node communication.
- **MPI** facilitates efficient inter-node reductions and sending of complex data structures.
- Program state synchronization is **explicit**.

OpenMP allows for high performance intra-node threading.

OpenMP provides an interface for the concurrent utilization of each SMP's shared memory;

Program state synchronization is **implicit**.

Hybrid Programming

In hybrid programming each process can have **multiple threads executing simultaneously**
All threads within a process **share all MPI objects** Communicators, requests, etc.

MPI defines 4 levels of thread safety:

- **MPI_THREAD_SINGLE** : One thread exists in program
- **MPI_THREAD_FUNNELED** : Multithreaded but only the master thread can make MPI calls Master is one that calls MPI_Init_thread()
- **MPI_THREAD_SERIALIZED:** Multithreaded, but only one thread can make MPI calls at a time
- **MPI_THREAD_MULTIPLE:** Multithreaded and any thread can make MPI calls at any time. Use MPI_Init_thread instead of MPI_Init if more than single thread

Hybrid Programming

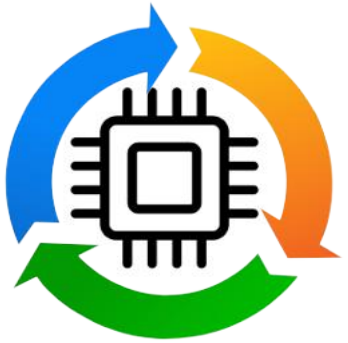
Potential advantages of the hybrid approach

- Fewer MPI processes for a given amount of cores
  - Improved load balance
  - All-to-all communication bottlenecks alleviated
- Decreased memory consumption if an implementation uses replicated data
- Additional parallelization levels may be available
- Possibility for dedicating threads for different tasks
  - e.g. dedicated communication thread or parallel I/O
- Dynamic parallelization patterns often easier to implement with OpenMP

Disadvantages of hybridization

- Increased overhead from thread creation/destruction
- More complicated programming
  - Code readability and maintainability issues
- Thread support in MPI and other libraries needs to be considered

Hybrid Examples



Hybrid Programming: Example

```c
#include <mpi.h>
#include <omp.h>

int main(int argc, char *argv[]) {
    int my_id, omp_rank;
    int provided, required=MPI_THREAD_FUNNELED;

    MPI_Init_thread(&argc, &argv, required,
                    &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
#pragma omp parallel private(omp_rank)
{
    omp_rank = omp_get_thread_num();
    printf("I'm thread %d in process %d\n",
           omp_rank, my_id);
}
    MPI_Finalize();
}
```

```
$ mpicc -fopenmp hybrid-hello.c -o hybrid-hello
$ srun  --ntasks=2 --cpus-per-task=4
  ./hybrid-hello

I'm thread 0 in process 0
I'm thread 0 in process 1
I'm thread 2 in process 1
I'm thread 3 in process 1
I'm thread 1 in process 1
I'm thread 3 in process 0
I'm thread 1 in process 0
I'm thread 2 in process 0
```

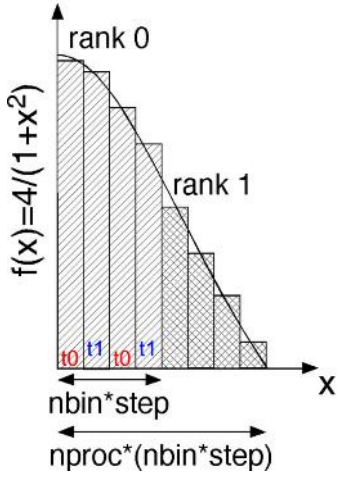
Hybrid: Comput PI With Serial Code

```
#include <stdio.h>
#include <math.h>
#include "timer.h"
int main(int argc, char *argv[]){
const int N = 10000000000;
const double h = 1.0/N;
const double PI = 3.14159265358979323846264 3;
double x,sum,pi,error,time; int i;

time = ctimer();
sum = 0.0;
for (i=0;i<=N;i++){
  x = h * (double)i;
  sum += 4.0/(1.0+x*x);}

pi = h*sum;
time += ctimer();

error = pi - PI;
error = error<0 ? -error:error;
printf("pi = %18.16f +/- %18.16f\n",pi,error);
printf("time = %18.16f sec\n",time);
return 0;}
```

GOAL : The following code computes the π number by using a numerical evaluation of an integral by a rectangle method.

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \cong \Delta \sum_{i=0}^{N-1} \frac{4}{1+x_i^2}$$

- **User-defined timer**
- **Calculation loop**
- **Print out result**



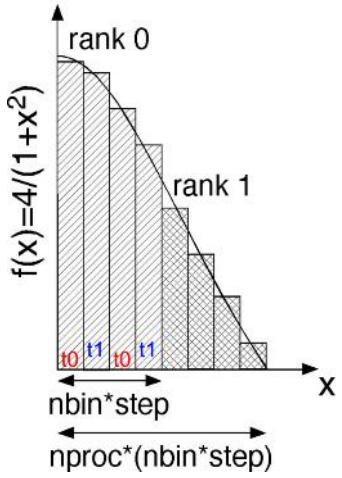
Hybrid: Comput PI With OpenMP Code

```
#include <stdio.h>
#include <math.h>
#include "timer.h"
int main(int argc, char *argv[]){
const int N = 10000000000;
const double h = 1.0/N;
const double PI = 3.14159265358979323846264338;
double x,sum,pi,error,time; int i;

time = -ctimer();
sum = 0.0;
#pragma omp parallel for shared(N,h),private(i,x),reduction(+:sum)
for (i=0;i<=N;i++){
  x = h * (double)i;
  sum += 4.0/(1.0+x*x);}

pi = h*sum;
time += ctimer();

.......

return 0;}
```

GOAL : The following code computes the π number by using a numerical evaluation of an integral by a rectangle method.

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \cong \Delta \sum_{i=0}^{N-1} \frac{4}{1+x_i^2}$$

- **OpenMP directive**



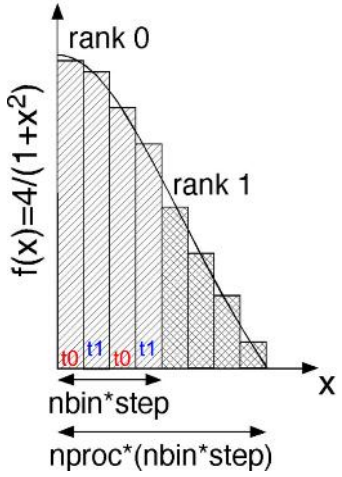
Hybrid: Comput PI With MPI Code

```
#include <stdio.h>
#include <math.h>
#include "timer.h"
int main(int argc, char *argv[]){
const int N = 10000000000;
const double h = 1.0/N;
const double PI = 3.14159265358979323846264 3;
double x,sum,pi,error,time,mypi; int i;
int myrank,nproc;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Comm_size(MPI_COMM_WORLD,&nproc);

time = -ctimer();
sum = 0.0;
for (i=myrank;i<=N;i=i+nproc){
  x = h * (double)i;
  sum += 4.0/(1.0+x*x);}
mypi = h*sum;
MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
time += ctimer();
......
return 0;}
```

- **MPI initialization**
- **Distributed loop**
- **Global reduction**

GOAL : The following code computes the π number by using a numerical evaluation of an integral by a rectangle method.

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \cong \Delta \sum_{i=0}^{N-1} \frac{4}{1+x_i^2}$$



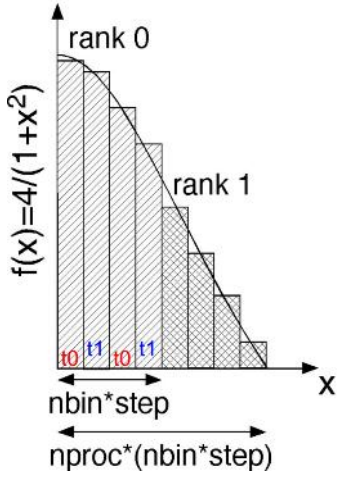
Hybrid: : Comput PI With MPI-OpenMP Code

```
#include <stdio.h>
#include <math.h>
#include "timer.h"
int main(int argc, char *argv[]){
const int N = 10000000000;
const double h = 1.0/N;
const double PI = 3.14159265358979323846264643;
double x,sum,pi,error,time,mypi; int i;
int myrank,nproc;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Comm_size(MPI_COMM_WORLD,&nproc);

time = -ctimer();
sum = 0.0;

#pragma omp parallel for shared(N,h,myrank,nproc),private(i,x),reduction(+:sum)
for (i=myrank;i<=N;i=i+nproc){
  x = h * (double)i;
  sum += 4.0/(1.0+x*x);}
mypi = h*sum;
MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
time += ctimer();
......
return 0;}
```

GOAL : The following code computes the π number by using a numerical evaluation of an integral by a rectangle method.

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \cong \Delta \sum_{i=0}^{N-1} \frac{4}{1+x_i^2}$$

- **OpenMP directive to parallelize local loop using threads**
- **Sum local values of π**



Hybrid: Deterministic Boltzmann Solver

Two-body elastic interactions

- Spatially *homogenous*
- *Isotropic* particle distribution
- Scattering amplitude may be momentum dependent

$$\partial_t f_1 = \frac{1}{E_1} \int_{p_{2,3,4}} |M(1,2,3,4)|^2 \left[f_3 f_4 (1+f_1)(1+f_2) - f_1 f_2 (1+f_3)(1+f_4) \right]$$

- Discretize $|\mathbf{p}| \quad \rightarrow \quad p_i \quad (0 \leq i < N)$
- At each time-step:
  1. For each i, compute Δf_i (given by a 4-dim integral $\Rightarrow$ slow)
  2. Check if $f_i + \Delta f_i \geq 0$ (for all i)
     If FALSE, re-run the previous loop with a smaller timestep
  3. Do $f_i + \Delta f_i \rightarrow f_i$ and return to step 1

- Collision integral reduces to 4-dim integral thanks to momentum conservation and *isotropy*

Hybrid: Deterministic Boltzmann Solver Sequential Version

```
// Core of the function that evolves f[i]

double *df = (double *)malloc(N*sizeof(double));
for (i=0;i<N;i++) df[i] = dt*C(i,f); // depends on f[k] with k!=i; computation of C(i,f) very slow

status = 0; for (i=0;i<N;i++) if (f[i]+df[i]<0) status = 1;
if (status==1) return 1;

for (i=0;i<N;i++) f[i] = f[i]+df[i]; free(df);
return 0;
```

- **Remarks**
  - Large fraction of the time spent in the first loop
  - For a reasonable grid size: 30 minutes/timestep
  - Typical evolution: 2000 timesteps (1000 hours, or 42 days...)

Hybrid: Deterministic Boltzmann Solver OpenMP Version

```
// Core of the function that evolves f[i]

double *df = (double *)malloc(N*sizeof(double));
#pragma omp parallel for num_threads(NT) schedule(dynamic) // <———— ONLY ADDITION
for (i=0;i<N;i++) df[i] = dt*C(i,f);

status = 0; for (i=0;i<N;i++) if (f[i]+df[i]<0) status = 1;
if (status==1) return 1;

for (i=0;i<N;i++) f[i] = f[i]+df[i]; free(df);
return 0;
```

Remarks

- Define NT (number of threads) in the scope of the function
- Better use of threads if NT divides N

Hybrid: Deterministic Boltzmann Solver MPI OpenMP Version

```
// Core of the function that evolves f[i]

int n = N/size, imin = rank*n, imax = (rank+1)*n;                    // <—— workload of each mode
double *df=NULL, *local_df = (double *)malloc(n*sizeof(double)); // <—— Node—local storage
if (rank==0) df = (double *)malloc(N*sizeof(double));                // <—— MASTER needs a buffer for df[i]

#pragma omp parallel for num_threads(NT) schedule(dynamic)
for (i=imin;i<imax;i++) local_df[i] = dt*C(i,f);

MPI_Gather(local_df,N1,MPI_DOUBLE,df,N1,MPI_DOUBLE,0,MPI_COMM_WORLD); // <—— Gather partial results on MASTER
if (rank==0) {status = 0; for (i=0;i<N;i++) if (f[i]+df[i]<0) status = 1;}
MPI_Bcast(&status,1,MPI_INT,0,MPI_COMM_WORLD);                       // <—— Broadcast test result
if (status==1) return 1;

if{rank==0}{for (i=0;i<N;i++) f[i] = f[i]+df[i]; free(df);}
MPI_Bcast(f,N,MPI_DOUBLE,0,MPI_COMM_WORLD);                          // <—— Broadcast updated f[i]
free(local_df);
return 0;
```

Hybrid: Deterministic Boltzmann Solver Timing

- Running time:
  - Sequential: 1830 seconds/timestep
  - OpenMP (16 cores): 154 seconds/timestep (×12 speedup, 75% efficiency)
  - MPI+OpenMP (32 nodes × 16 cores): 6 seconds/timestep (×306 speedup, 60% efficiency)

  Computation time reduced from 1000 hours to 3 hours 16 minutes

- Coding time:
  - Sequential: one week (about 600 lines of code)
  - +OpenMP: +one minute (+1 line)
  - +MPI: +one hour (+10 lines)

Thank you for your attention !