OpenMP performance of FDPS

Jun Makino

Internal Seminar July 13, 2022

Talk Summary

- I present the current (improved...) OpenMP performance of FDPS.
- It turned out that there are several parts in the current FDPS which prevent the good OpenMP performance scaling.
- I have identified most (but not yet all) of them.
- Current OpenMP performance: 64-thread run: using 64 cores of AMD EPYC 7742, 2M particles, $\theta = 0.5$, 0.3 sec/step. (1 core: 4.1 sec)
- Still not quite ideal. MPI 64 core: 0.12 sec.

Talk plan

1. Why OpenMP performance of FDPS matter?
2. Performance of public(+) version (7.1b+)
3. Problems found and improvement made so far
4. Current Performance
5. Summary

Why OpenMP performance of FDPS matter?

Many modern HPC systems have very large number of cores, and thus using large number of cores per MPI process is desirable for many reasons.

- **Small number of MPI processes $\rightarrow$ less communication overhead.**
- **Some systems (in particular K and Fugaku) are designed that way, and flat MPI does not work well.**

Performance of public(+) version (7.1b+)

Test run performance, C++ nbody.cpp sample code, $N = 2^{21}$, $\theta = 0.5$, $n_b = 512$. 1 and 64 cores on a dual EPYC 7742 (128 core) machine

- **single core: 4.05 sec/step**
- **64-thread OpenMP: 0.60 sec/step**
- **64-process 1thread/process MPI: 0.114 sec/step**

64-core MPI run is 36 times faster than single-core run, while 64-core OpenMP run is only 7 times faster.
This is not quite ideal and we should be able to improve the OpenMP parallel performance.

Problems found and improvements made so far

What I found so far

- **Implicit automatic array initialization within sorting routine**
- **Post-sort array movement not parallelized**
- **kick/drift not parallelized**
- **Still more things...**

Implicit automatic array initialization within sorting routine

In my new samplesort routine I have the following line of code:

```
T bodylocalcopy[nt][destarraysize];
```

where `T` is the FDPS data type passed to be sorted. It is essentially

```
class TreeParticle{
    public: KeyT key_;  U32 adr_ptcl_;
}
```

and

```
class KeyT{
    public: U64 hi_; U64 lo_;
}
```

Implicit automatic array ... (2)

I expected the allocation of automatic array at runtime in process stack would take O(1) time.

It actually did not take any time in my test program for the sort routine.

However, when called from within FDPS, the allocation consumes a very long time (0.06 sec).

I thought variables (even arrays) created on stack are not initialized and thus do not consume calculation time.

However, if a C++ class variable has explicit (user-defined) default constructor, it is still called even for variable on stack.

Implicit automatic array ... (3)

I replaced

```
KeyT() : hi_(0), lo_(0){}
```

which was in the original FDPS class definition with

```
KeyT() {}
```

and the initialization time disappeared.

Implicit automatic array ... (3)

I replaced

```
KeyT() : hi_(0), lo_(0){}
```

which was in the original FDPS class definition with

```
KeyT() {}
```

and the initialization time disappeared.
C++ is too difficult for me...

Post-sort array movement not parallelized

The following loop was not OMP parallelized.

```
for(S32 i=0; i<n_glb_tot_; i++){
    const U32 adr = adr_org_from_adr_sorted_glb_[i];
    if( GetMSB(adr) == 0){
        epj_sorted_[n_cnt_ep] = epj_org_[adr];
        tp_glb_[i].adr_ptcl_ = n_cnt_ep; n_cnt_ep++;
    } else{
        spj_sorted_[n_cnt_sp] = spj_org_[ClearMSB(adr)];
        tp_glb_[i].adr_ptcl_ = SetMSB(n_cnt_sp); n_cnt_sp++;
    }
}
```

(Actually, a parallel version was there, but with a comment “thread parallelized, but not fast.”...)

How can we parallelize this loop?

This loop makes separate lists of treenodes and particles from combined sorted list.

Parallelization seems difficult since where to store the i-th element depends on the results up to i-1.

What I did:

First divide the source array to p pieces where p is the number of threads, and let each thread count treenodes and particles in its array piece.

Then determine the first location for each piece (taking summations of the number of elements), and then in the second loop actually store the treenodes and particles.

Improvement: 0.12 sec

kick/drift not parallelized

Since nbody.cpp was intended as a simple sample code, loops in the sample code are not OMP parallelized. I added macros `PS_OMP_PARALLEL_FOR` like the following

```
template<class Tpsys>
void kick(Tpsys & system,
          const PS::F64 dt) {
    PS::S32 n = system.getNumberOfParticleLocal();
PS_OMP_PARALLEL_FOR
    for(PS::S32 i = 0; i < n; i++) {
        system[i].vel  += system[i].acc * dt;
    }
}
```

Still more things...

There still seem to exist one or two locations where either unnecessary O(N) operations are done or not OMP parallelized.

I hope to be able to figure out....

Current Performance

Cores	OpenMP	MPI
1	4.078	–
2	2.815	2.264
4	1.503	1.12
8	0.792	0.584
16	0.518	0.304
32	0.373	0.175
64	0.316	0.114
128	0.332	0.097

Not too bad for up to 8 threads

Still more than a factor of two slower for more than 32 threads...

Calculation time breakdown

Part	time(sec)
Make local tree	0.0546
Local moment	0.0041
Make global tree	0.0342
Global moment	0.0056
Treewalk&Force	0.0987
Misc(kick etc)	0.0286
Unknown	0.095

Need to identify and fix “Unknown” part...

Should optimize the sorting routine for the data type used in FDPS. Current one is designed for general purpose.

Probably we can reduce the time down to 0.15 sec. but not much faster....

Talk Summary

- I present the current (improved...) OpenMP performance of FDPS.
- It turned out that there are several parts in the current FDPS which prevent the good OpenMP performance scaling.
- I have identified most (but not yet all) of them.
- Current OpenMP performance: 64-thread run: using 64 cores of AMD EPYC 7742, 2M particles, $\theta = 0.5$, 0.3 sec/step. (1 core: 4.1 sec)
- Still not quite ideal. MPI 64 core: 0.114 sec.