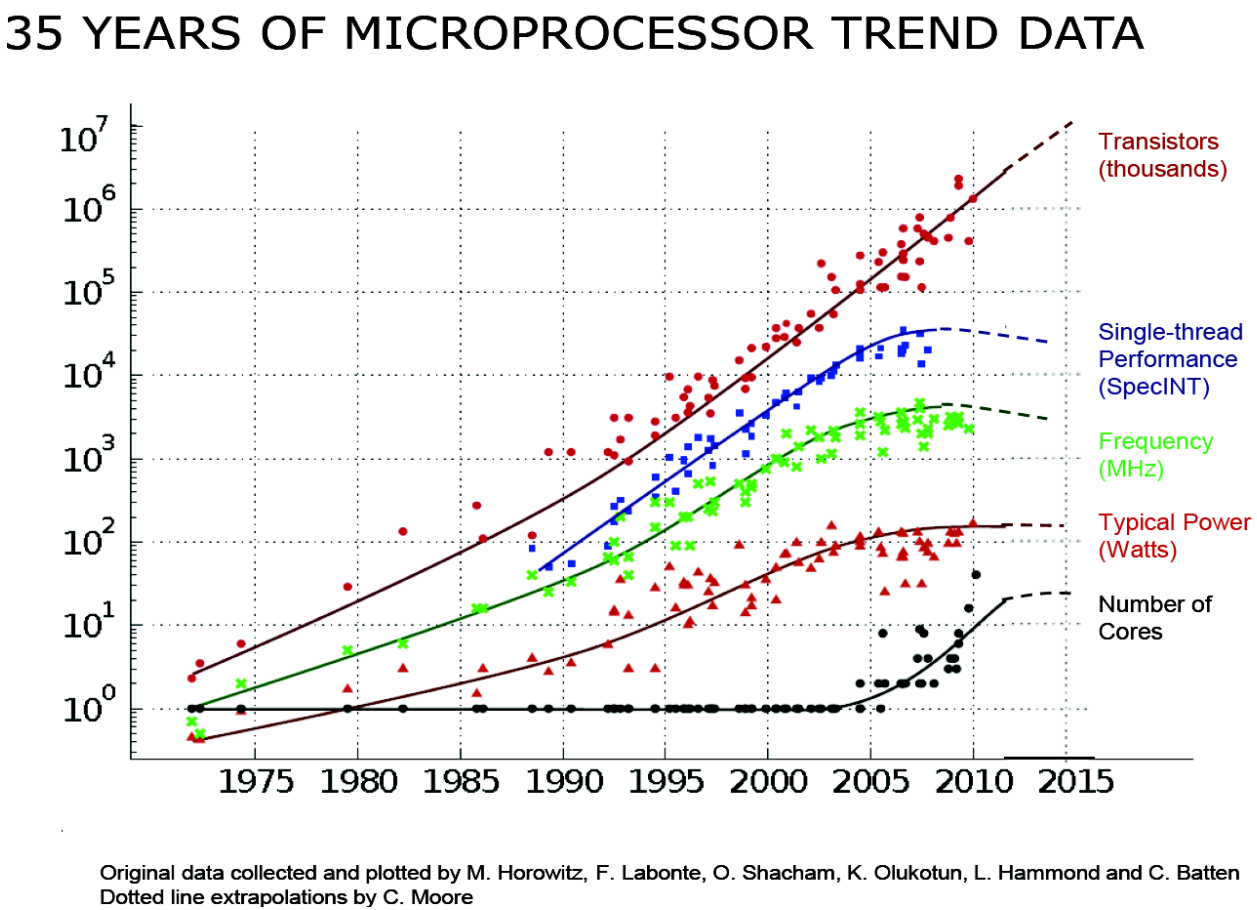
Cineca
TRAINING
High Performance
Computing 2016

Porting on Manycore architectures: A case study

V. Ruggiero (v.ruggiero@cineca.it)
Roma, 4 November 2016
SuperComputing Applications and Innovation Department

Trends


35 YEARS OF MICROPROCESSOR TREND DATA
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1975 1980 1985 1990 1995 2000 2005 2010 2015
Transistors (thousands)
Single-thread Performance (SpecINT)
Frequency (MHz)
Typical Power (Watts)
Number of Cores
Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Trends: summarizing...

- The number of transistors increases
- The power consumption must not increase
- The density can not increase on a single chip

Solution :

- Exposing parallelism increasing the number of cores

Multicore vs ManyCore?

- Doubling the number of standard cores per die with every semiconductor process generation starting with a single processor.
- Switching from sequential to modestly parallel computing makes programming much more difficult without rewarding this greater effort with a dramatic improvement in power-performance. Hence, multicore is unlikely to be the ideal answer.
- The alternative approach moving forward is to adopt the manycore architecture which employs simpler cores running at modestly lower clock frequencies. Rather than progressing from two to four to eight cores with the multicore approach, a manycore design has hundreds of cores.
- Even if the simpler core offers only one-third the computational efficiency of the more complex out-of-order cores, a manycore design stills be an order of magnitude more power- and area-efficient in terms of sustained performance.

What is Many-Core?

"The terms many-core and massively multi- core are sometimes used to describe multi- core architectures with an especially high number of cores(tens or hundreds)"

(Andras Vajda)

What is Intel Xeon Phi?

- 7100 / 5100 / 3100 Series available
- 5110P:
  - Intel Xeon Phi clock: 1053 MHz
  - 60 cores in-order
  - 1 TFlops/s DP peak performance (2 Tflops SP)
  - 4 hardware threads per core
  - 8 GB DDR5 memory
  - 512-bit SIMD vectors (32 registers)
  - Fully-coherent L1 and L2 caches
  - Max Memory bandwidth (theoretical) 320 GB/s
  - Max Thermal Design Power (TDP): 225 W

Intel Xeon Phi

- It does not require learning
  - a new programming language
  - new parallelization techniques
- The programming paradigm is based on
  - C/ Fortran
  - OMP/MPI standards
- Achieving good performance is not simple
  - the hardware has own characteristics that must understood in order to port the code in an efficient manner.

Parallelization

Data	Thread/Task	Process
Vectorization Automatic Directives/Pragmas Libraries	Multi-Threading OpenMP TBB, CilkTM Plus OpenCL pthreads	Message Passing MPI

Cluster

SIMD

Multicore Manycore

SIMD

- What are the microprocessor vector extensions or SIMD (Single Instruction Multiple Data Units)
- How to use them
  - Through the compiler via automatic vectorization
    - Manual transformations that enable vectorization
    - Directives to guide the compiler
  - Through intrinsics
- Main focus on vectorizing through the compiler
  - Code more readable
  - Code portable

What is Vectorization?

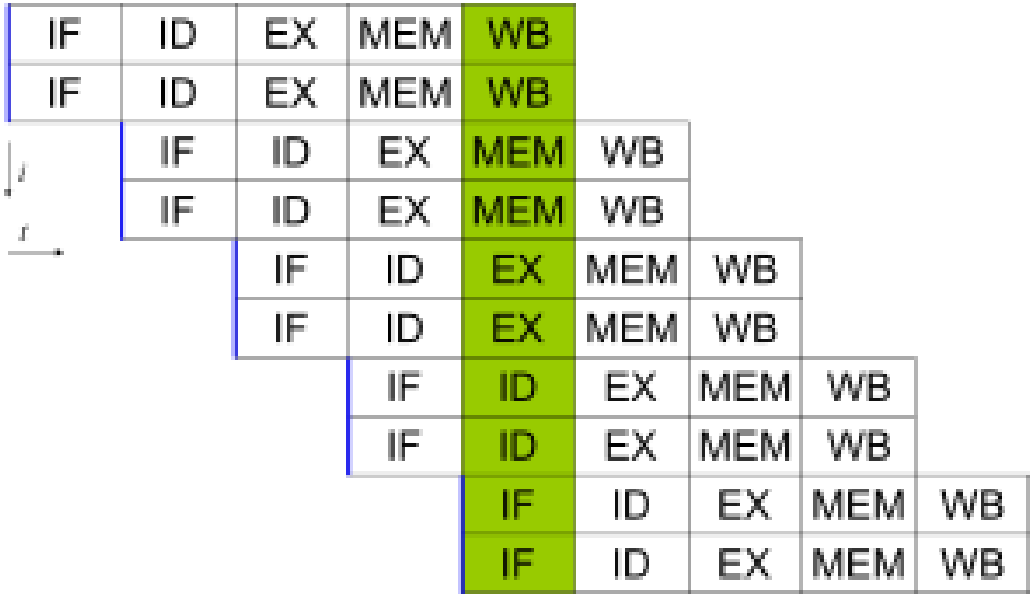
- Hardware Perspective: Specialized instructions, registers, or functional units to allow in-core parallelism for operations on arrays (vectors) of data.
- Compiler Perspective: Determine how and when it is possible to express computations in terms of vector instructions
- User Perspective: Determine how to write code in a manner that allows the compiler to deduce that vectorization is possible.

What Happened To Clock Speed?

- Everyone loves to misquote Moore's Law:
  - "CPU speed doubles every 18 months."
- Correct formulation:
  - "Available on-die transistor density doubles every 18 months."
- For a while, this meant easy increases in clock speed
- Greater transistor density means more logic space on a chip

Clock Speed Wasn't Everything

- Chip designers increased performance by adding sophisticated features to improve code efficiency.
- Branch-prediction hardware.
- Out-of-order and speculative execution.
- Superscalar chips.
- Superscalar chips look like conventional single-core chips to the OS.
- Behind the scenes, they use parallel instruction pipelines to (potentially) issue multiple instructions simultaneously.



IF: Instrunction Fetch
ID: Instruction Decode
EX: Execute
MEM: Memory access
WB: Register Write Back

SIMD Parallelism

- CPU designers had, in fact, been exposing explicit parallelism for a while.
- MMX is an early example of a SIMD (Single Instruction Multiple Data) instruction set.
  - Also called a vector instruction set.
- Normally, scalar instructions operate on single items in memory.
  - Can be different size in terms of bytes, of course.
  - Standard x86 arithmetic instructions are scalar. (ADD, SUB, etc.)
- Vector instructions operate on packed vectors in memory.
- A packed vector is conceptually just a small array of values in memory.
  - A 128-bit vector can be two doubles, four floats, four int32s, etc.
  - The elements of a 128-bit single vector can be thought of as v[0], v[1], v[2], and v[3].

SIMD Parallelism

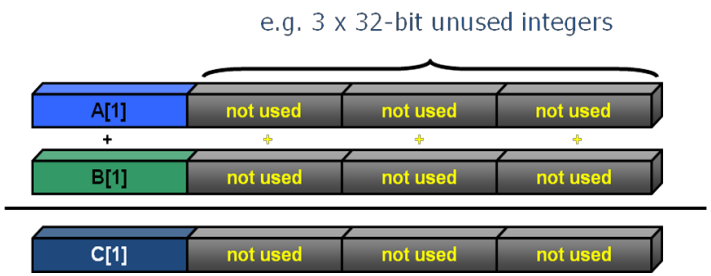
- Vector instructions are handled by an additional unit in the CPU core, called something like a vector arithmetic unit.
- If used to their potential, they can allow you to perform the same operation on multiple pieces of data in a single instruction.
  - Single-Instruction, Multiple Data parallelism.
  - Your algorithm may not be amenable to this...
  - ... But lots are. (Spatially-local inner loops over arrays are a classic.)
- It has traditionally been hard for the compiler to vectorise code efficiently, except in trivial cases.
  - It would suck to have to write in assembly to use vector instructions...

Vector units

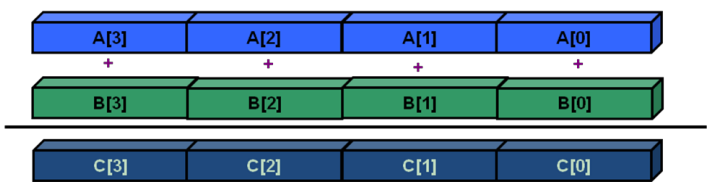
- Auto-vectorization is transforming sequential code to exploit the SIMD (Single Instruction Multiple Data) instructions within the processor to speed up execution times
- Vector Units performs parallel floating/integer point operations on dedicate SIMD units
  - Intel: MMX, SSE, SSE2, SSE3, SSE4, AVX
- Think vectorization in terms of loop unrolling
- Example: summing 2 arrays of 4 elements in one single instruction

```
C(0) = A(0) + B(0)
C(1) = A(1) + B(1)
C(2) = A(2) + B(2)
C(3) = A(3) + B(3)
```

no vectorization



vectorization



SIMD - evolution

- SSE: 128 bit register (Intel Core - AMD Opteron)
  - 4 floating/integer operating in single precision
  - 2 floating/integer operating in double precision
- AVX: 256 bit register (Intel Sandy Bridge - Intel Broadwell - Intel Haswell - AMD Bulldozer)
  - 8 floating/integer operating in single precision
  - 4 floating/integer operating in double precision
- MIC: 512 bit register (Intel Knights Corner - Intel Knights Landing)
  - 16 floating/integer operating in single precision
  - 8 floating/integer operating in double precision

How do we access the SIMD units?

- C or fortran code and vectorizing compiler

```
for (i=0; i<LEN; i++)
c[i] = a[i] + b[i];
```

- Macros or Vector Intrinsics

```
void example(){
__m128 rA, rB, rC;
for (int i = 0; i <LEN; i+=4){
rA = _mm_load_ps(&a[i]);
rB = _mm_load_ps(&b[i]);
rC = _mm_add_ps(rA,rB);
_mm_store_ps(&C[i], rC);
}}
```

- Assembly Language

```
..B8.5
movaps a(,%rdx,4), %xmm0
addps b(,%rdx,4), %xmm0
movaps  %xmm0, c(,%rdx,4)
addq $4, %rdx
cmpq $rdi, %rdx
ji ..B8.5
```

Vector-aware coding

- Know what makes vectorizable at all
  - "for" loops (in C) or "do" loops (in fortran) that meet certain constraints
- Know where vectorization will help
- Evaluate compiler output
  - Is it really vectorizing where you think it should?
- Evaluate execution performance
  - Compare to theoretical speedup
- Know data access patterns to maximize efficiency
- Implement fixes: directives, compilation flags, and code changes
  - Remove constructs that make vectorization impossible/impractical
  - Encourage and (or) force vectorization when compiler doesn't, but should
  - Better memory access patterns

Writing Vector Loops

- Basic requirements of vectorizable loops:
  - Countable at runtime
    - Number of loop iterations is known before loop executes
    - No conditional termination (break statements)
  - Have single control flow
    - No Switch statements
    - 'if' statements are allowable when they can be implemented as masked assignments
  - Must be the innermost loop if nested
    - Compiler may reverse loop order as an optimization!
  - No function calls
    - Basic math is allowed: pow(), sqrt(), sin(), etc
    - Some inline functions allowed

When vectorization fails

- Not Inner Loop: only the inner loop of a nested loop may be vectorized, unless some previous optimization has produced a reduced nest level. On some occasions the compiler can vectorize an outer loop, but obviously this message will not then be generated.
- Low trip count:The loop does not have sufficient iterations for vectorization to be worthwhile.
- Vectorization possible but seems inefficient:the compiler has concluded that vectorizing the loop would not improve performance. You can override this by placing **`#pragma vector always`** (C C++) or **`!dir$ vector always`** (Fortran) before the loop in question
- Contains unvectorizable statement: certain statements, such as those involving switch and printf , cannot be vectorized

When vectorization fails

- Subscript too complex: an array subscript may be too complicated for the compiler to handle. You should always try to use simplified subscript expressions
- Condition may protect exception: when the compiler tries to vectorize a loop containing an if statement, it typically evaluates the RHS expressions for all values of the loop index, but only makes the final assignment in those cases where the conditional evaluates to TRUE. In some cases, the compiler may not vectorize because the condition may be protecting against accessing an illegal memory address. You can use the `#pragma ivdep` to reassure the compiler that the conditional is not protecting against a memory exception in such cases.
- Unsupported loop Structure: loops that do not fulfill the requirements of countability, single entry and exit, and so on, may generate these messages

https://software.intel.com/en-us/articles/vectorization-diagnostics-for-intelr-c-compiler-150-and-above

When vectorization fails

- Operator unsuited for vectorization: Certain operators, such as the % (modulus) operator, cannot be vectorized
- Non-unit stride used: non-contiguous memory access.
- Existence of vector dependence: vectorization entails changes in the order of operations within a loop, since each SIMD instruction operates on several data elements at once. Vectorization is only possible if this change of order does not change the results of the calculation

Vectorized loops? (intel compiler)

- Vectorization is enabled by the flag -vec and by default at -O2.

```
-vec-report[N] (deprecated)
-qopt-report[=N] -qopt-report-phase=vec
```

N	Diagnostic Messages
0	No diagnostic messages; same as not using switch and thus default
1	Tells the vectorizer to report on vectorized loops.
2	Tells the vectorizer to report on vectorized and non-vectorized loops.
3	Tells the vectorizer to report on vectorized and non-vectorized loops and any proven or assumed data dependencies.
4	Tells the vectorizer to report on non-vectorized loops.
5	Tells the vectorizer to report on non-vectorized loops and the reason why they were not vectorized.
6	Tells the vectorizer to use greater detail when reporting on vectorized and non-vectorized loops and any proven or assumed data dependencies.
7	Tells the vectorizer to emit vector code quality message ids and corresponding data values for vectorized loops. It provides information such as the expected speedup, memory access patterns, and the number of vector idioms for vectorized loops.

Vectorization Report (intel compiler):example

```
ifort -O3 -qopt-report=5
```

```
 LOOP BEGIN at matmat.F90(51,1)
      remark #25427: Loop Statements Reordered
      remark #15389: vectorization support: reference C has unaligned access
      remark #15389: vectorization support: reference B has unaligned access
[ matmat.F90(50,1) ]
      remark #15389: vectorization support: reference A has unaligned access
[ matmat.F90(49,1) ]
      remark #15381: vectorization support: unaligned access used inside loop body
[ matmat.F90(49,1) ]
      remark #15301: PERMUTED LOOP WAS VECTORIZED
      remark #15451: unmasked unaligned unit stride stores: 3
      remark #15475: --- begin vector loop cost summary ---
      remark #15476: scalar loop cost: 229
      remark #15477: vector loop cost: 43.750
      remark #15478: estimated potential speedup: 5.210
      remark #15479: lightweight vector operations: 24
      remark #15480: medium-overhead vector operations: 2
      remark #15481: heavy-overhead vector operations: 1
      remark #15482: vectorized math library calls: 2
      remark #15487: type converts: 2
      remark #15488: --- end vector loop cost summary ---
      remark #25015: Estimate of max trip count of loop=28
   LOOP END
```

When vectorization fails

- Programmers need to provide the necessary information
- Programmers need to transform the code

- Add compiler directives
- Transform the code
- Program using vector intrinsics

The Code

- Fortran code for the compressible turbulent boundary layers over a flat plate.
- Solve the NS equations using a FD 6th-order energy consistent method.
- A full (3D) MPI splitting is allowed.
- For an easy implementation of the recycling method the number of partitions in the third direction must be 1.
- The authors are Sergio Pirozzoli and Matteo Bernardini

Starting point

- grid size 64X64X64
- running on 1 process
- compiler options: -mmic -r8
- elapsed mean time in seconds per time step

Optimization Level	no-vec	vec
O0	43.9	43.9
O1	9.96	9.96
O2	4.33	6.91
O3	4.28	6.86
fast	3.80	6.65

Profiling (500 time steps)

no-vec	vec	
2013.62	3468.67	Total Time [s]

time[s]	time[%]	time/visit[us]
37.73	1.9	6281.44
28.11	1.4	9361.68
263.78	13.1	87867.08
545.82	27.1	363882.92
260.60	12.9	173733.02
288.00	14.3	192001.64
105.40	5.2	70267.38
273.38	13.6	182254.33
78.93	3.9	157866.76
...		

time[s]	time[%]	time/visit[us]
14.97	0.4	2492.57
28.13	0.8	9367.47
90.03	2.6	29988.86
1145.89	33.0	763924.02
402.93	11.6	268620.46
669.23	19.3	446156.29
338.96	9.8	225971.02
554.56	16.0	369708.88
81.84	2.4	163678.48
...		

visits	function
6006	bcwall__
3003	bcswap_
3002	getpress_
1500	visflx_lap_
1500	euler_i_
1500	euler_k_
1500	pgrad_
1500	euler_j_
500	rk_
...	

visflx_lap.f

```
...
remark #25451: Advice: Loop Interchange, if possible,
might help loopnest. Suggested Permutation: ( 1 2 3 ) --> ( 3 2 1  )
   remark #15542: loop was not vectorized: inner loop was already vectorized
   remark #25015: Estimate of max trip count of loop=64
   LOOP BEGIN at /galileo/home/userinternal/vruggie1/OLD_CHANNEL/CHANNEL/visflx_lap.f(77,8)
...
```

```
do k=1,nz
  do j=1,ny
    do i=1,nx
      ...
     do l=1,mm
     ...
     do l=1,mm
     ...
```

```
!dir$ simd
   do i=1,nx
  do j=1,ny
do k=1,nz
     ...
!dir$ novector
     do l=1,mm
     ...
!dir$ novector
     do l=1,mm
     ...
```

763924.02

232511.55

30% gain per visit

euler_*.f

What can be done to improve these routines?

- Where the autovectorization is implemented?
- Insert compiler directives to vectorize the loops
- Run the code and in all loops check the difference between estimated potential speedup and measured speedup
- Where the vectorization is inefficient inderdict vectorization

euler_*.f

- Elapsed time per visit
- The comparison is between new version of the routine and old autovectorized version of the routine

routine	old	new	% gain
euler_i	268620.46	169546.83	26.9
euler_j	369708.88	170571.67	53.8
euler_k	446156.29	180130.26	59.6

pgrad.f

```
...
remark #25451: Advice: Loop Interchange, if possible,
might help loopnest. Suggested Permutation: ( 1 2 3 ) --> ( 3 2 1 )
remark #15542: loop was not vectorized: inner loop was already vectorized
..
```

```
do i=1,nx
  dx = 1./dcsidx(i)
    do j=1,ny
      dy = 1./detady(j)
        do k=1,nz
          dz = 1./dzitdz(k)
```

```
!dir$ novector
do k=1,nz
   dz = 1./dzitdz(k)
!dir$ novector
     do j=1,ny
       dy = 1./detady(j)
         do i=1,nx
           dx = 1./dcsidx(i)
```

225971.02

21629.31

89% gain per visit

rk.f

- 4 autovectorized loops
- Inserting novector directive in one inefficient loop

163678.48 $\Longrightarrow$ 136289.40

17% gain per visit

no-vector vs vector

routine	% gain
euler_i	2.4
euler_j	6.4
euler_k	6.2
bcwall	60.4
getpress	65.9
visflx_lap	36.2
pgrad	69.2
rk	14.3

no-vector vs vector

- elapsed mean time in seconds per time step

grid size	novec	vec	% gain
64X64X64	3.80	2.76	27.4
128X64X64	7.78	5.49	29.4
128X128X64	15.4	10.9	29.2
128X128X128	30.8	27.3	11.4
256X128X128	64.6	54.3	16.0
256X256X128	141.3	111.6	21.1

Speedup vs Grid Size

	Speedup					
grid size	4	8	16	32	64	128
64X64X64	3.6	6.7	11.5	18.4	25.1	25.1
128X64X64	3.9	7.0	12.8	20.3	28.9	36.6
128X128X64	4.0	7.7	12.4	18.9	33.0	40.4
128X128X128	4.1	8.2	13.9	22.2	39.6	58.1
256X128X128	4.0	8.1	16.1	30.2	46.4	66.2
256X256X128	4.1	8.3	16.5	32.4	51.4	72.9

Test Case

- Grid Size 256X256X128
- Running 500 time steps
- 3 different number of processes

type	max_buf[B]	visits	time[s]	time[%]	time/visit[us]	region
			32 processes			
ALL	2,104,704	1,320,912	75806.35	100.0	57389.40	ALL
MPI	1,526,736	661,824	10861.47	14.3	16411.42	MPI
USR	349,512	354,480	57090.80	75.3	161055.07	USR
COM	228,456	304,608	7854.08	10.4	25784.21	COM
			64 processes			
ALL	2,104,704	2,591,728	99595.04	100.0	38428.05	ALL
MPI	1,526,736	1,323,648	22079.52	22.2	16680.81	MPI
USR	349,512	658,864	65600.23	65.9	99565.67	USR
COM	228,456	609,216	11915.29	12.0	19558.40	COM
			128 processes			
ALL	2,104,704	5,178,720	152831.16	100.0	29511.38	ALL
MPI	1,526,736	2,644,608	54731.92	35.8	20695.66	MPI
USR	349,512	1,315,680	83500.89	54.6	63465.96	USR
COM	228,456	1,218,432	14598.34	9.6	11981.25	COM

MPI functions overview

type	max_buf[B]	visits	time[s]	time[%]	time/visit[us]	region
			32 processes			
MPI	1,369,824	576,768	1396.63	1.8	2421.47	MPI_Sendrecv
MPI	143,298	80,280	4962.52	6.5	61815.12	MPI_Allreduce
MPI	3,888	1,068	4182.50	5.5	3916196.38	MPI_File_write_all
MPI	1,938	164	29.30	0.0	178661.71	MPI_Bcast
...						
			64 processes			
MPI	1,369,824	1,153,536	3202.97	3.2	2776.66	MPI_Sendrecv
MPI	143,298	160,560	7719.17	7.8	48076.52	MPI_Allreduce
MPI	3,888	2,136	9732.89	9.8	4556595.41	MPI_File_write_all
MPI	1,938	328	1051.28	1.1	3205114.76	MPI_Bcast
...						
			128 processes			
MPI	1,369,824	2,307,072	8423.17	5.5	3651.02	MPI_Sendrecv
MPI	143,298	321,072	24903.02	16.3	77562.10	MPI_Allreduce
MPI	3,888	3,096	18083.24	11.8	5840841.00	MPI_File_write_all
MPI	1,938	392	2610.34	1.7	6659024.05	MPI_Bcast
...						

The next step

- Increasing the scalability
- Increasing the parallelism
- Hybrid parallelism (MPI-OpenMP)