并行与分布式计算基础：第十四讲

杨超
chao_yang@pku.edu.cn

2019 秋

课程基本情况

- 课程名称：并行与分布式计算基础
- 授课教师：杨超（chao_yang@pku.edu.cn，理科 1 号楼 1520）
- 课程助教：尹鹏飞（pengfeiyin@pku.edu.cn）

授课内容（暂定）

- 引言
- 硬件体系架构
- 并行计算模型
- 编程与开发环境
- MPI 编程与实践
- OpenMP 编程与实践
- GPU 编程与实践
- 前沿问题选讲

上课时间（地点：二教 211）

上课时间	星期一	星期二	星期三	星期四	星期五
第 1 节 (8:00-8:50)					
第 2 节 (9:00-9:50)					
第 3 节 (10:10-11:00)				单周	
第 4 节 (11:10-12:00)				单周	
第 5 节 (13:00-13:50)		每周			
第 6 节 (14:00-14:50)		每周			
第 7 节 (15:10-16:00)					
第 8 节 (16:10-17:00)					
第 9 节 (17:10-18:00)					
第 10 节 (18:40-19:30)					
第 11 节 (19:40-20:30)					
第 12 节 (20:40-21:30)					

内容提纲

1. OpenMP 编程-5：补遗篇
   - 线程控制 (含复习)
   - 持久变量
   - 向量化
   - 作业

线程控制 (含复习)

动态线程

- 动态线程：系统动态选择并行区的线程数 (默认：一般为关闭)。
- 打开/关闭动态线程
  - 库函数：

```
void omp_set_dynamic(int flag)
```

  - 环境变量：

```
export OMP_DYNAMIC=true
```

- 检查动态线程是否打开
  - 库函数：

```
int omp_get_dynamic (void)
```

- 一个小例子：
  - flag 为 0：并行区开启 10 个线程；
  - flag 非 0：并行区开启 1-10 个线程 (系统决定)。

```
1   ...
2   omp_set_dynamic(flag);
3 #pragma omp parallel num_threads(10)
4   {
5     /* do work here */
6   }
7   ...
```

嵌套并行 (nested parallelism)

- 嵌套并行：指在并行区之内开启并行区 (默认：一般为开启)。
- 打开/关闭嵌套并行
  - 库函数：

```
void omp_set_nested(int flag)
```

  - 环境变量：

```
export OMP_NESTED=true
export OMP_NUM_THREADS=n1,n2,n3,...
```

- 检查嵌套并行是否打开
  - 库函数：

```
int omp_get_nested (void)
```

- 思考：下面的例子运行结果是什么？

omp_nested.c

```
1   ...
2   omp_set_dynamic(0);
3  #pragma omp parallel num_threads(2)
4   {
5
6     omp_set_nested(1);
7  #pragma omp parallel num_threads(3)
8     {
9  #pragma omp single
10      printf ("Inner: num_thds=%d\n", omp_get_num_threads());
11    }
12
13 #pragma omp barrier
14    omp_set_nested(0);
15 #pragma omp parallel num_threads(3)
16    {
17 #pragma omp single
```

```
18        printf ("Inner: num_thds=%d\n", omp_get_num_threads());
19      }
20
21  #pragma omp barrier
22  #pragma omp single
23      printf ("Outer: num_thds=%d\n", omp_get_num_threads());
24
25    }
26    ...
```

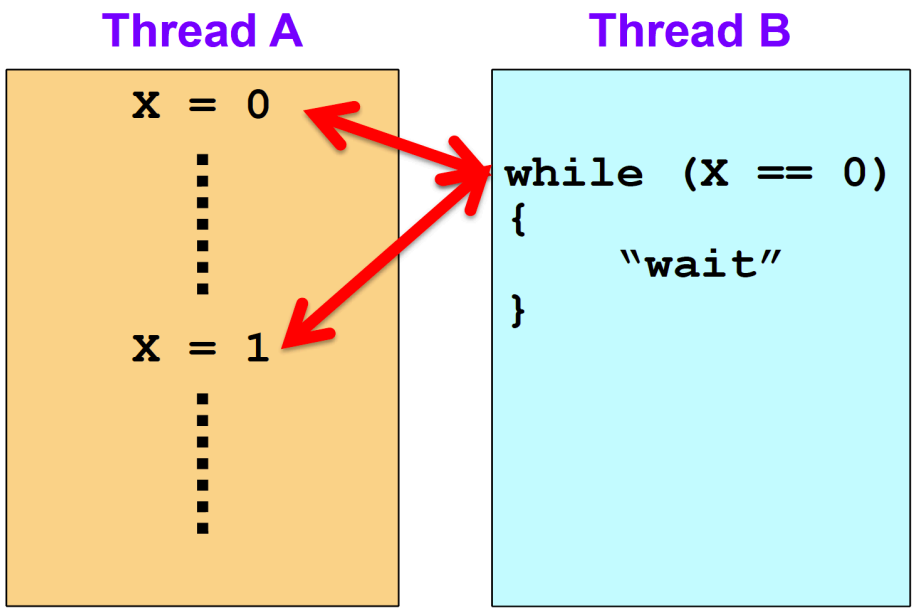
flush 构造

- OpenMP 的松弛一致性 (relaxed consistency)：
  - 数据不仅在内存，还在缓存 (以及寄存器等) 中有多份拷贝；
  - 实际上 OpenMP 的共享变量在本地缓存中并不随时更新。
- flush 构造：手动更新当前线程本地缓存中的数据。

```
#pragma omp flush [acq_rel | release | acquire] (list)
```

- OpenMP 的一些同步操作隐含包含了 flush，比如：
  - 并行区入口，critical/ordered 区的入口、出口；
    (注意：工作共享构造的入口/出口是不隐含包含 flush 的)
  - 显式、隐式的 barrier 操作等。

- 举例:



If shared variable X is kept within a register, the modification may not be made visible to the other thread(s)

- 若确需 `flush`，一般置于共享变量的写操作后，或读操作前；
- 合理的算法设计一般不需要显式的 `flush` (因为容易出错).

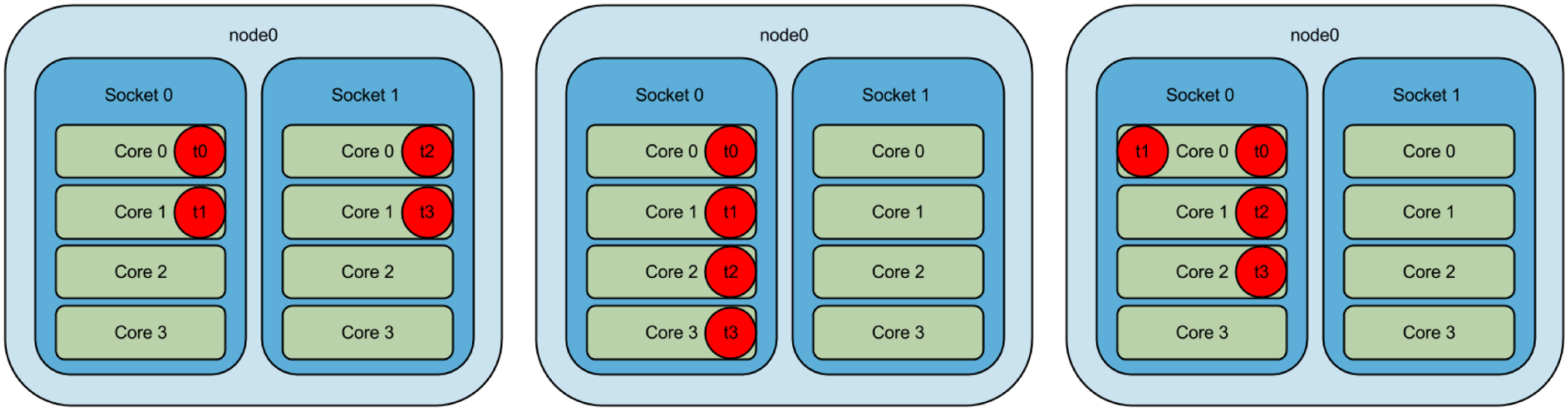
堆栈大小 (stack size)

- 除了主线程，OpenMP 的每个线程的私有变量存储空间受线程的堆栈大小控制。
- OpenMP 标准并不规定具体的堆栈大小，依赖于具体实现：
  - Intel 编译器：默认大小一般为 4MB；
  - gcc/gfortran 编译器：默认大小一般为 2MB。
- 如果超出堆栈大小，程序的行为不可控；
- 可以通过环境变量修改默认堆栈大小，比如：

```
export OMP_STACKSIZE=32M
export OMP_STACKSIZE=8192K
```

线程亲和性 (affinity) 和线程绑定 (binding)

- 线程亲和性：决定了 NUMA 架构的系统上线程在物理计算核心的映射策略。



- 线程绑定：显式确定线程与物理计算核心的对应关系，以提升性能。

- OpenMP3.1 标准仅提供了非常有限的支持，可通过如下方式开启：

```
export OMP_PROC_BIND=TRUE
```

- 如果使用 Intel 编译器，可通过如下方式设置线程亲和性：

```
export KMP_AFFINITY={scatter,compact..}
```

(参考网上教程：https://software.intel.com/en-us/node/522691)

- OpenMP 4.5 开始对上述功能提供了较好支持 (参阅相关手册)；
- `numactl` 工具有时候可以发挥重要作用 (参阅网上教程，如：http://www.glennklockwood.com/hpc-howtos/process-affinity.html)。

持久变量

持久 (persistent) 变量的线程私有化

- 持久 (persistent) 变量：一般指生存周期为整个程序的变量数据，例如全局变量、静态变量等.
- 如果希望每个线程拥有自己的持久变量，并且可以随心所欲地在不同线程间传递各自持久变量的值，怎么办?
- OpenMP 的线程私有型变量提供了上述机制：
  - `threadprivate` 构造提供了持久变量的私有化机制；
  - `copyin` 从句提供了将主线程的 `threadprivate` 变量的值传递给其他线程的机制；
  - `copyprivate` 从句提供了将某线程的 `threadprivate` 变量的值广播给其他线程的机制.

threadprivate 型变量

- threadprivate 构造：将持久变量置为线程私有类型

```
#pragma omp threadprivate (list)
```

- 对全局变量：必须置于全局变量声明列表之后并在被首次使用之前，否则不起作用；
- 对静态变量：必须置于 static 变量声明列表之后并在被首次使用之前，否则不起作用.
- 注意：与 private 类型变量的最大差别是，threadprivate 型变量的值可以跨并行区有效 (前提是动态线程关闭，并且每个并行区线程数一致).
- 思考：下面的程序运行结果是什么?

omp_threadprivate.c

```
1  ...
2  int a = 0, b = 0;
3  #pragma omp threadprivate(a)
4
5  int compare(int x) {
6    static int n = 2;
7  #pragma omp threadprivate(n)
8    if (n < x) n = x;
9    return n;
10 }
11
12 int main(int argc, char *argv[]){
13   int tid, c, d;
14   omp_set_dynamic(0);
15   printf("1st Parallel Region:\n");
16 #pragma omp parallel num_threads(4) private(tid,b,c,d)
17   {
```

```
18      tid = omp_get_thread_num();
19      a = tid + 1;
20      b = tid + 2;
21      c = compare(a);
22      d = compare(b);
23      printf("Thread %d: a,b,c,d = %d %d %d %d\n",tid,a,b,c,d);
24    }
25    printf("Serial Region: a,b = %d %d\n",a,b);
26    printf("2nd Parallel Region:\n");
27  #pragma omp parallel num_threads(4) private(tid,c,d)
28    {
29      tid = omp_get_thread_num();
30      c = compare(a + 2);
31      d = compare(b + 3);
32      printf("Thread %d: a,b,c,d = %d %d %d %d\n",tid,a,b,c,d);
33    }
34    ...
```

- 运行结果 (请正确使用 sbatch 或者 salloc):

```
 $ ./threadprivate
1st Parallel Region:
Thread 0: a,b,c,d = 1 2 2 2
Thread 1: a,b,c,d = 2 3 2 3
Thread 2: a,b,c,d = 3 4 3 4
Thread 3: a,b,c,d = 4 5 4 5
Serial Region: a,b = 1 0
2nd Parallel Region:
Thread 2: a,b,c,d = 3 0 5 5
Thread 1: a,b,c,d = 2 0 4 4
Thread 0: a,b,c,d = 1 0 3 3
Thread 3: a,b,c,d = 4 0 6 6
```

全局或静态变量的传递与广播

- copyin 从句用于将主线程的 threadprivate 变量的值传递给其他线程，仅能用于并行区构造的初始化：

```
#pragma omp parallel [for | sections] copyin(list)
{ ... }
```

- copyprivate 从句用于将某线程的 threadprivate 变量的值广播给其他线程，仅能用于 single 构造，并在其出口处起作用：

```
#pragma omp single copyprivate(list)
{ ... }
```

- 思考：下面的程序运行结果是什么？

omp_copyprivate.c

```
1  ...
2  int counter = 0;
3  #pragma omp threadprivate(counter)
4
5  int increment_counter(){
6    return(++counter);
7  }
8
9  int main(int argc, char *argv[]){
10   int tid, c;
11   omp_set_dynamic(0);
12   omp_set_num_threads(4);
13   printf("1st Parallel Region:\n");
14 #pragma omp parallel private(tid,c)
15   {
16     tid = omp_get_thread_num();
17 #pragma omp single copyprivate(counter)
```

```
18      counter = 50 + tid;
19      c = increment_counter();
20      printf("ThreadId: %d, count = %d\n", tid, c);
21  #pragma omp barrier
22      counter = 100 + tid;
23      c = increment_counter();
24      printf("ThreadId: %d, count = %d\n", tid, c);
25    }
26    printf("2nd Parallel Region:\n");
27  #pragma omp parallel private(tid,c) copyin(counter)
28    {
29      tid = omp_get_thread_num();
30      c = increment_counter();
31      printf("ThreadId: %d, count = %d\n", tid, c);
32    }
33    ...
```

- 运行结果 (请正确使用 sbatch 或者 salloc):

```
 $ ./copyprivate
1st Parallel Region:
ThreadId: 3, count = 51
ThreadId: 0, count = 51
ThreadId: 1, count = 51
ThreadId: 2, count = 51
ThreadId: 3, count = 104
ThreadId: 0, count = 101
ThreadId: 1, count = 102
ThreadId: 2, count = 103
2nd Parallel Region:
ThreadId: 1, count = 102
ThreadId: 3, count = 102
ThreadId: 0, count = 102
ThreadId: 2, count = 102
```

并行区与工作共享构造的从句汇总

	parallel	for	parallel for	sections	parallel sections	single
if	●		●		●	
num_threads	●		●		●	
default	●		●		●	
shared	●	●	●		●	
private	●	●	●	●	●	●
reduction	●	●	●	●	●	
firstprivate	●	●	●	●	●	●
lastprivate		●	●	●	●	
copyin	●		●		●	
copyprivate						●
schedule		●	●			
ordered		●	●			
collapse		●	●			
nowait		●		●		●

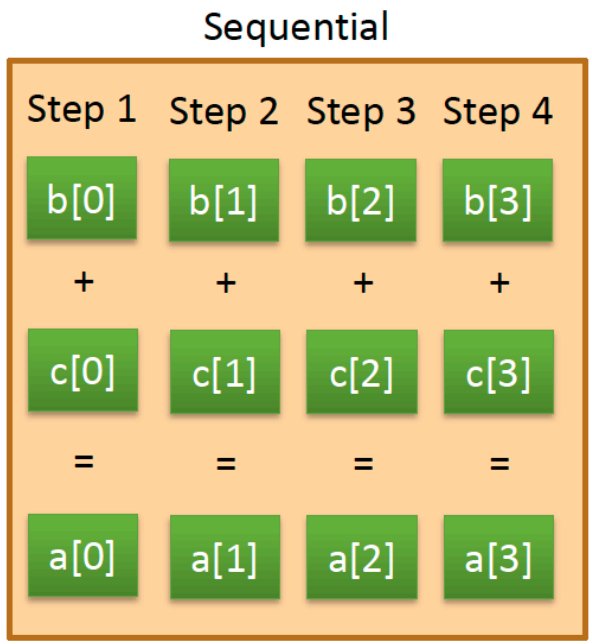
向量化

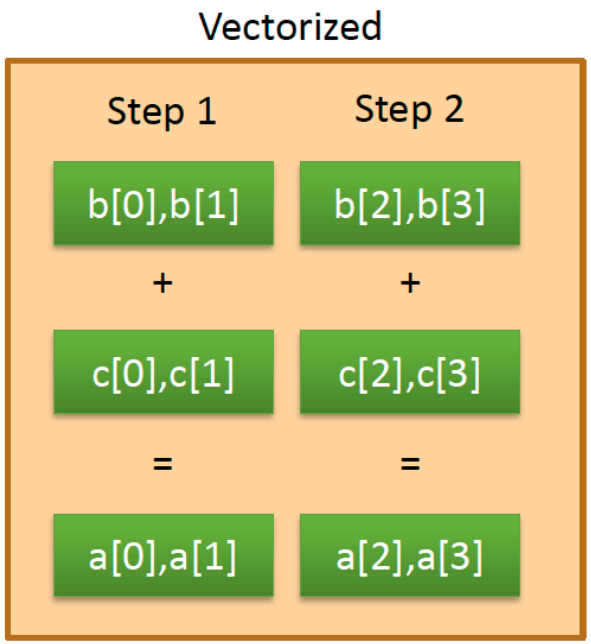
向量化 (vectorization)

- SIMD = Single Instruction Multiple Data
- 大多数处理器均提供具有 SIMD 向量化功能的硬件指令;
- 这些 SIMD 指令一般作用于向量化寄存器中;
- 通过 SIMD 向量化，可以加速计算，例如:

```
double a[4],b[4],c[4];
...

for(i=0 ; i < 4 ; i++)
{
      a[i]=b[i]+c[i];
}
```





- 不同处理器支持的 SIMD 向量化宽度依赖于硬件本身：

Vector lengths on Intel architectures

→ 128 bit: SSE = Streaming SIMD Extensions

2 x double

4 x float

→ 256 bit: AVX = Advanced Vector Extensions

4 x double

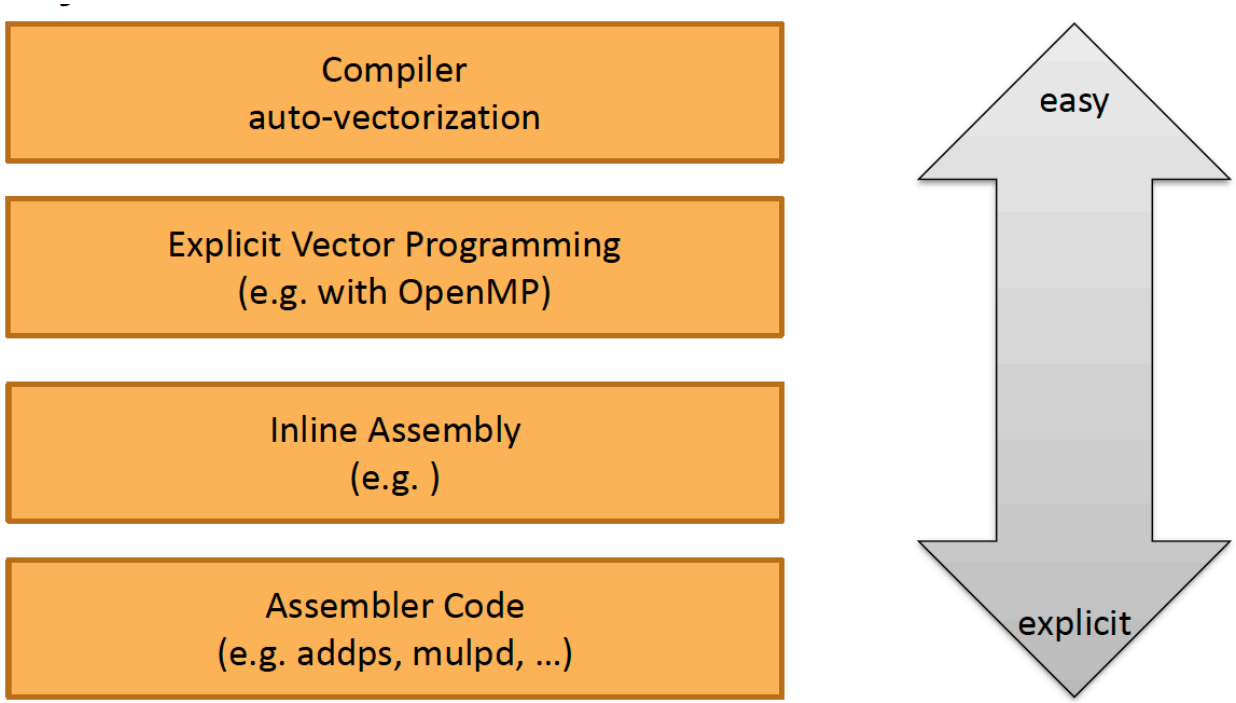
8 x float

→ 512 bit: AVX-512

8 x double

16 x float

- 实现 SIMD 向量化有几种不同的手段：



OpenMP 的 simd 构造

- OpenMP 提供 simd 构造对循环进行向量化计算:

```
#pragma omp simd [clause1 | clause2 | ...]
  for_loops
```

- 支持的从句:

```
private (list)
lastprivate (list)
reduction (op:list)
collapse (n)
linear (list[:step])
aligned (list[:step])
safelen (length)
```

- collapse 从句：先对多重循环进行合并，然后进行向量化

```
collapse (n)
```

- linear 从句：列出与迭代变量有线性关系的变量

```
linear (list[:step])
```

- aligned 从句：列出内存地址对齐的数组或指针

```
aligned (list[:step])
```

- safelen 从句：给出没有循环间数据依赖的最大步长

```
safelen (length)
```

数据对齐

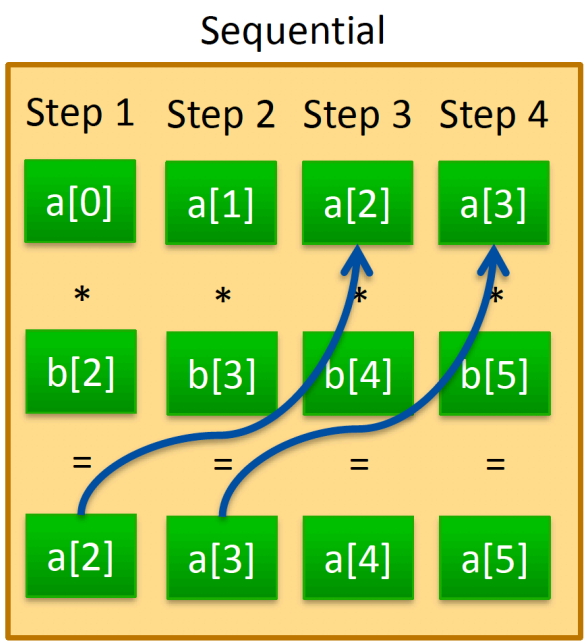
- SIMD 向量化的效果与数据是否对齐 (aligned) 有很大关系，例如：

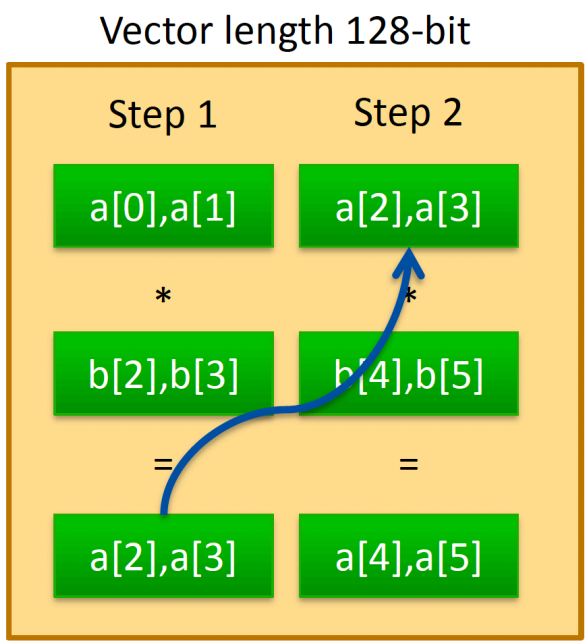
向量化的 safelen

- 进行向量化时需要注意不要破坏循环携带的数据依赖；
- safelen 从句用于给出没有循环间数据依赖的最大步长；
- 与 for 构造不同，simd 构造向量化的循环仍然按照顺序依次执行；
- 下例中，safelen=1：

```
double a[6],b[6];
...

for(i=2 ; i < 6 ; i++)
{
      a[i]=a[i-2]*b[i];
}
```





for simd 构造

- simd 构造可与 for 构造合并:

```
#pragma omp for simd [clause1 | clause2 | ...]
  for_loops
```

- 此时，循环任务将先按照线程进行分配，每个线程分配的任务将进一步按照 SIMD 向量化进行二次分配;
- 对于双方共有的从句，将同时起作用;
- 对于各自专有的从句，将各自起作用.

向量化与函数调用

- SIMD 向量化的循环中，如果有外部函数调用，有可能带来严重的性能瓶颈，因为此时函数的执行是完全串行的，例如：

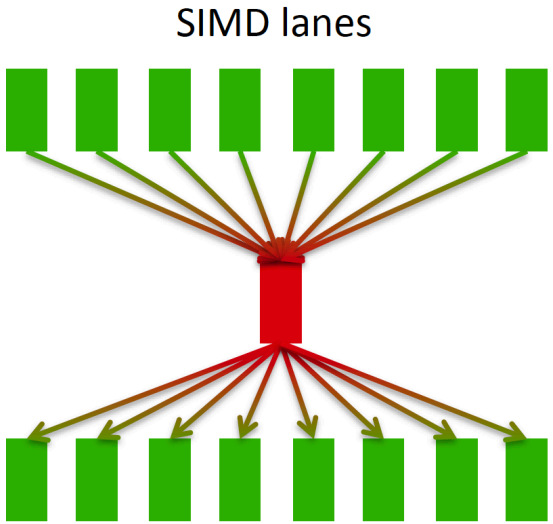
```
for(i=0 ; i < N ; i++)
{

      a[i]=b[i]+c[i];

      d[i]=sin(a[i]);

      e[i]=5*d[i];

}
```



Solutions:

- avoid or inline functions
- create functions which work on vectors instead of scalars

- 上述问题的解决手段包括：使用内联函数或者创建向量化的函数.

declare simd 构造

- declare simd 构造：用于提示编译器根据需要生成一个至多个具有 SIMD 向量化功能的函数

```
#pragma omp declare simd [clause1 | clause2 | ...]
  function_definition/declaration
```

- 支持的从句：

```
linear (list[:step])
aligned (list[:step])
uniform (list)
simdlen (length)
inbranch | notinbranch
```

- linear 从句：列出与迭代变量有线性关系的变量

```
linear (list[:step])
```

- aligned 从句：列出内存地址对齐的变量

```
aligned (list[:step])
```

- uniform 从句：列出不变量

```
uniform (list)
```

- simdlen 从句：给出需要同时向量化计算的变量个数

```
simdlen (length)
```

- inbranch/notinbranch 从句：声明在/不在分支判断中被调用

```
inbranch | notinbranch
```

编译器的自动向量化

- 事实上，不少编译器都提供了较为不错的自动向量化功能；
- 下表总结了几种主流编译器中开启和关闭自动向量化的选项：

Compiler	Compilers Options	Disabling Vectorizer
Intel C/C++ 17.0	-O3 -xHost -qopt-report3 -qopt-report-phase=vec,loop -qopt-report-embed	-no-vec
GCC C/C++ 6.3.0	-O3 -ffast-math -fivopts -march=native -fopt-info-vec -fopt-info-vec-missed	-fno-tree-vectorize
LLVM/Clang 3.9.1	-O3 -ffast-math -fvectorize -Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize	-fno-vectorize
PGI C/C++ 16.10	-O3 -Mvect -Minfo=loop,vect -Mneginfo=loop,vect	-Mnovect

示例程序：计算 π

omp_cpi2.c

```
1  ...
2  #pragma omp declare simd
3  double f(double x) {
4    return (16.0*(x-1.0)/(x*x*x*x-2.0*x*x*x+4.0*x-4.0));
5  }
6  int main(int argc, char *argv[]){
7    ...
8  #pragma omp parallel for simd private(x) linear(i) reduction(+:
       pi)
9    for (i = 0; i < n; i++) {
10     x = h * ((double)i + 0.5);
11     pi += h * f(x);
12   }
13   ...
14 }
```

- 注：这里选择了一个计算量更大的积分公式用来检验 SIMD 的效果
- 编译方式

```
$ module load gcc # load gcc 9.2.0
$ gcc -o cpi2 omp_cpi2.c -O3 -Wall -fopenmp \
-fopt-info-vec
(gcc 显示成功进行向量化的信息)
$ gcc -o cpi2nosimd omp_cpi2nosimd.c -O3 -Wall \
-fopenmp -fopt-info-vec -fno-tree-vectorize
(gcc 没有显示成功进行向量化的信息)
```

- 测试结果

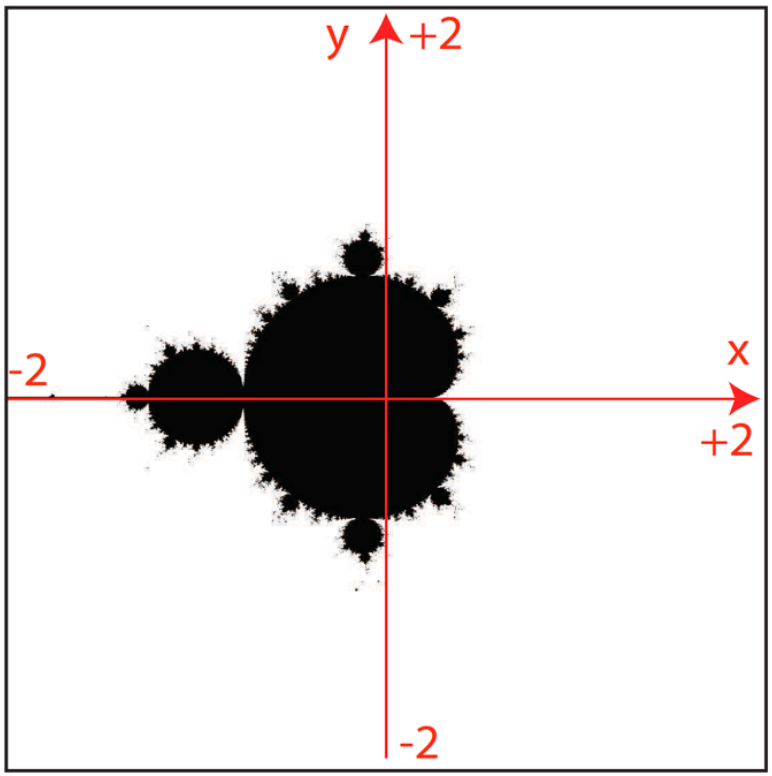
线程数	1	2	4	8
无向量化	3.92s	2.07s	1.17s	0.65s
向量化	1.98s	1.08s	0.63s	0.33s

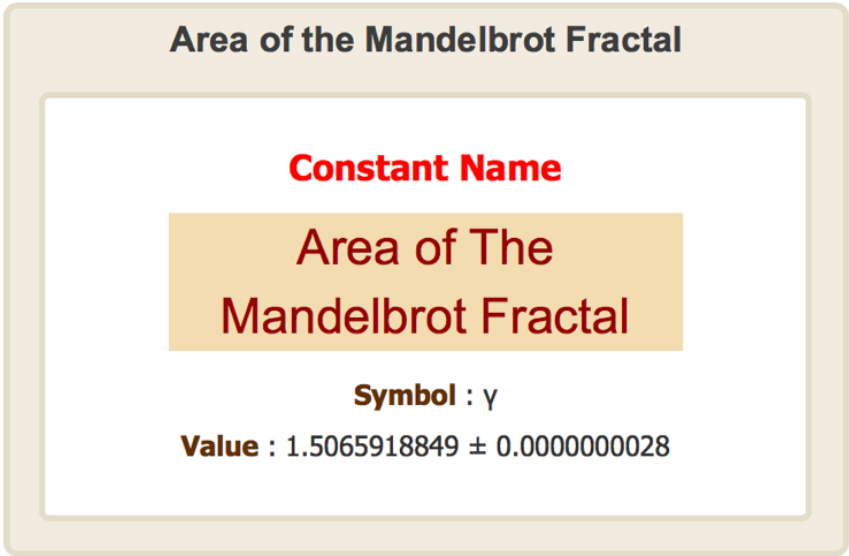
作业

作业-3：计算 Mandelbrot Set 的面积

- Mandelbrot Set：

$$M = \left\{ c \in \mathbf{C} : \exists s \in \mathbf{R}^+, \forall n \in \mathbf{N}, |P_c^n(0)| \leq s \right\}, P_c : z \mapsto z^2 + c.$$





- 要求：
  - 数院机器，MPI 或 OpenMP 并行，8 个核；
  - 用墙钟时间函数完整计时 (从程序开始到结束)；
  - 提交时间：2019 年 11 月 28 日 24 点前发给助教。
- 计分方式：
  - 正确性：误差小于 5×10^{-5}，否则无效；
  - 性能：时间越短分数越高，超过 5 分钟不得分。
- 思路提示：
  - 模型：迭代上界 $s = 2$，大于 2 则逃逸；
  - 算法：Monte Carlo 法、grid search 法；
  - 实现：减小搜索区域，提高负载平衡，降低任务额外开销等。