

OpenMP

Kenjiro Taura

Contents

- 1 Overview
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Contents

- 1 Overview
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Goal

- learn OpenMP, by far the most widespread standard API for shared memory parallel programming
- learn that various schedulers execute your parallel programs differently

- *de fact* standard model for programming shared memory machines
- C/C++/Fortran + parallel directives + APIs
 - by `#pragma` in C/C++
 - by comments in Fortran
- many free/vendor compilers, including GCC

OpenMP reference

- official home page: <http://openmp.org/>
- specification:
<http://openmp.org/wp/openmp-specifications/>
- latest version is 4.5
(<http://www.openmp.org/mp-documents/openmp-4.5.pdf>)
- section numbers below refer to those in OpenMP spec 4.0
(<http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>)

GCC and OpenMP

- <http://gcc.gnu.org/wiki/openmp>
- gcc 4.2 → OpenMP spec 2.5
- gcc 4.4 → OpenMP spec 3.0 (task parallelism)
- gcc 4.7 → OpenMP spec 3.1
- gcc 4.9 → OpenMP spec 4.0 (SIMD)

Compiling/running OpenMP programs with GCC

- compile with `-fopenmp`

```
1 $ gcc -Wall -fopenmp program.c
```

- run the executable specifying the number of threads with `OMP_NUM_THREADS` environment variable

```
1 $ OMP_NUM_THREADS=1 ./a.out # use 1 thread  
2 $ OMP_NUM_THREADS=4 ./a.out # use 4 threads
```

- see 2.5.1 “Determining the Number of Threads for a parallel Region” for other ways to control the number of threads

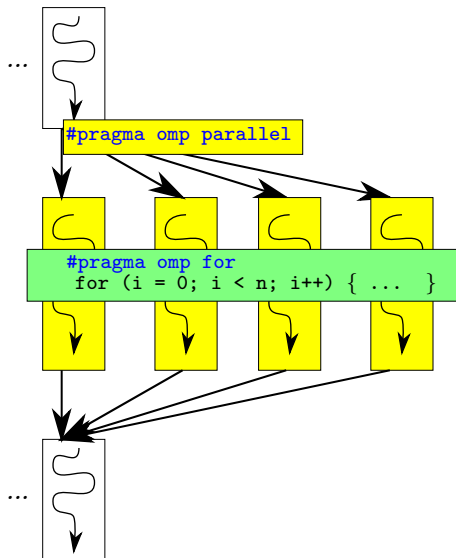
Contents

- 1 Overview
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Two pragmas you must know first

- `#pragma omp parallel` to launch a team of threads (2.5)
- then `#pragma omp for` to distribute iterations to threads (2.7.1)

Note: all OpenMP pragmas have the common format: `#pragma omp ...`



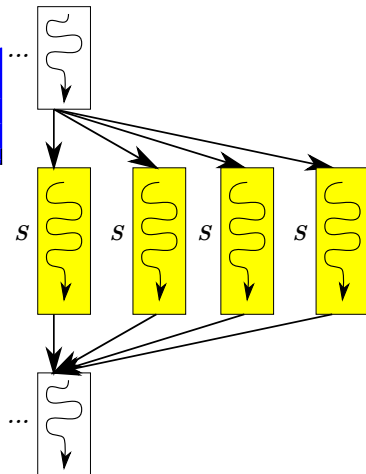
#pragma parallel

- basic syntax:

```
1  ...  
2  #pragma omp parallel  
3  S  
4  ...
```

- basic semantics:

- create a team of OMP_NUM_THREADS threads
- the current thread becomes the *master* of the team
- *S will be executed by each member of the team*
- the master thread waits for all to finish *S* and continue



parallel pragma example

```
1 #include <stdio.h>
2 int main() {
3     printf("hello\n");
4     #pragma omp parallel
5     printf("world\n");
6     return 0;
7 }
```

```
1 $ OMP_NUM_THREADS=1 ./a.out
2 hello
3 world
4 $ OMP_NUM_THREADS=4 ./a.out
5 hello
6 world
7 world
8 world
9 world
```

Remarks : what does `parallel` do?

- you may assume an OpenMP thread \approx OS-supported thread (e.g., Pthread)
- that is, if you write this program

```
1 int main() {  
2   #pragma omp parallel  
3     worker();  
4 }
```

and run it as follows,

```
1 $ OMP_NUM_THREADS=50 ./a.out
```

you will get 50 OS-level threads, each doing `worker()`

How to distribute work among threads?

- `#pragma omp parallel` creates threads, *all executing the same statement*
- it's not a means to parallelize work, *per se*, but just a means to create a number of similar threads (SPMD)
- so how to distribute (or partition) work among them?
 - ① do it yourself
 - ② use *work sharing* constructs

Do it yourself: functions to get the number/id of threads

- `omp_get_num_threads()` (3.2.2) : the number of threads *in the current team*
- `omp_get_thread_num()` (3.2.4) : the current thread's id (0, 1, ...) in the team
- they are primitives with which you may partition work yourself by whichever ways you prefer
- e.g.,

```
1  #pragma omp parallel
2  {
3      int t  = omp_get_thread_num();
4      int nt = omp_get_num_threads();
5      /* divide n iterations evenly amongst nt threads */
6      for (i = t * n / nt; i < (t + 1) * n / nt; i++) {
7          ...
8      }
9  }
```

Contents

- 1 Overview
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Contents

- 1 Overview
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Work sharing constructs

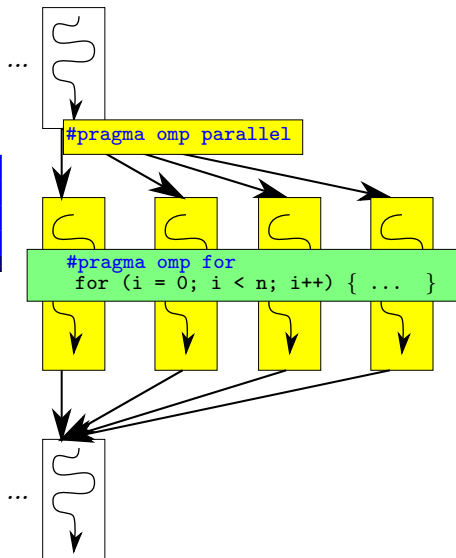
- in theory, `parallel` construct is all you need to do things in parallel
- but it's too inconvenient
- OpenMP defines ways to *partition* work among threads (*work sharing constructs*)
 - for
 - task
 - section

#pragma omp for (work-sharing for)

- basic syntax:

```
1 #pragma omp for
2 for(i=...; i...; i+=...){
3     S
4 }
```

- basic semantics:
the threads in the team
divide the iterations among
them
- but how? \Rightarrow scheduling



#pragma omp for restrictions

- not arbitrary for statement is allowed after a for pragma
- strong syntactic restrictions apply, so that *the iteration counts can easily be identified at the beginning* of the loop
- roughly, it must be of the form:

```
1 #pragma omp for
2 for(i = init; i < limit; i += incr)
3     S
```

except < and += may be other similar operators

- *init*, *limit*, and *incr* must be loop invariant

Contents

- 1 Overview
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Scheduling (2.7.1)

- `schedule` clause in work-sharing for loop determines how iterations are divided among threads
- There are three alternatives (`static`, `dynamic`, and `guided`)

static, dynamic, and guided

- `schedule(static[,chunk])`:
predictable round-robin
- `schedule(dynamic[,chunk])`:
each thread repeats fetching *chunk* iterations
- `schedule(guided[,chunk])`:
threads grab many iterations in early stages; gradually reduce iterations to fetch at a time
- *chunk* specifies the minimum granularity (iteration counts)

#pragma omp for schedule(static)



#pragma omp for schedule(static,3)



#pragma omp for schedule(dynamic)



#pragma omp for schedule(dynamic,2)



#pragma omp for schedule(guided)



#pragma omp for schedule(guided,2)



Other scheduling options and notes

- `schedule(runtime)` determines the schedule by `OMP_SCHEDULE` environment variable. e.g.,

```
1 $ OMP_SCHEDULE=dynamic,2 ./a.out
```

- `schedule(auto)` or `no schedule clause` choose an implementation dependent default

Parallelizing loop nests by `collapse`

- `collapse(l)` can be used to partition nested loops. e.g.,

```
1 #pragma omp for collapse(2)
2 for (i = 0; i < n; i++)
3     for (j = 0; j < n; j++)
4         S
```

will partition n^2 iterations of the doubly-nested loop

- `schedule` clause applies to nested loops as if the nested loop is an equivalent flat loop
- restriction: the loop must be “*perfectly nested*” (the iteration space must be a rectangular and no intervening statement between different levels of the nest)

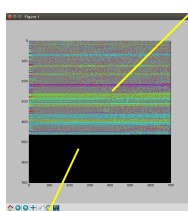
Visualizing schedulers

- seeing is believing. let's visualize how loops are distributed among threads
- write a simple doubly nested loop and run it under various scheduling options

```
1 #pragma omp for collapse(2) schedule(runtime)
2 for (i = 0; i < 1000; i++)
3     for (j = 0; j < 1000; j++)
4         unit_work(i, j);
```

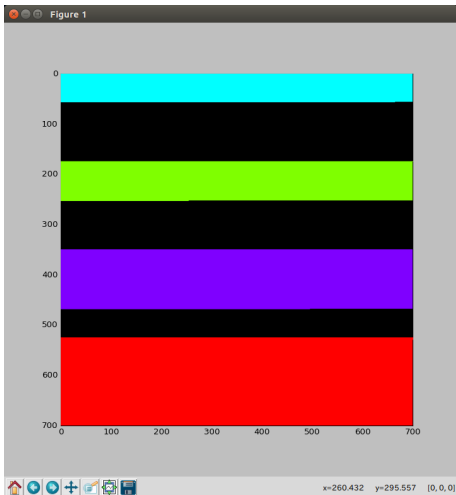
- load per point is systematically skewed:
 - ≈ 0 in the lower triangle
 - drawn from $[100, 10000]$ (clocks) in the upper triangle

load $\sim [100, 10000]$ clocks

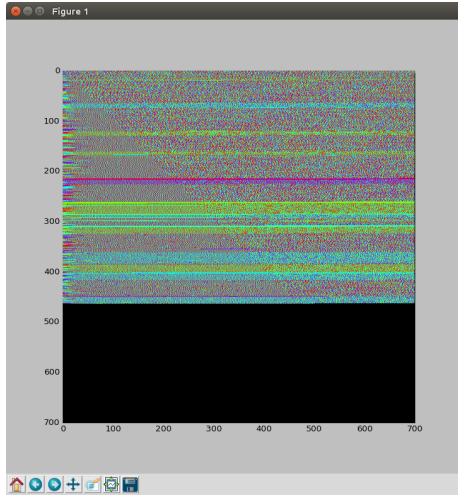


load ≈ 0

Visualizing schedulers



static



dynamic

Contents

- 1 Overview
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Task parallelism in OpenMP

- OpenMP's initial focus was simple parallel loops
- since 3.0, it supports task parallelism
- but why it's necessary?
- aren't `parallel` and `for` all we need?

Limitation of parallel for

- what if you have a parallel loop inside another

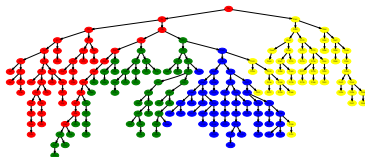
```
1 for ( ... ) {  
2     ...  
3     for ( ... ) ...  
4 }
```

- perhaps inside a separate function?

```
1 main() {  
2     for ( ... ) {  
3         ...  
4         g();  
5     }  
6 }  
7 g() {  
8     for ( ... ) ...  
9 }
```

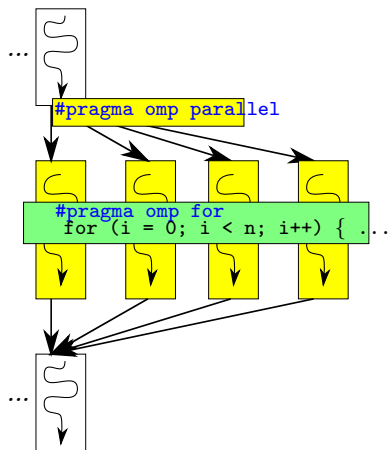
- what for parallel recursions?

```
1 qs() {  
2     if (...) { ... }  
3     else {  
4         qs();  
5         qs();  
6     }  
7 }
```



parallel for can't handle nested parallelism

- OpenMP generally ignores nested parallel pragma when enough threads have been created by the outer parallel pragma, for good reasons
- the fundamental limitation is its simplistic work-sharing mechanism
- tasks address these issues, by allowing tasks to be created at arbitrary points of execution (and a mechanism to distribute them across cores)



Task parallelism in OpenMP

- syntax:

- create a task \approx TBB's `task_group::run`

```
1  #pragma omp task  
2  S
```

- wait for tasks \approx TBB's `task_group::wait`

```
1  #pragma omp taskwait
```


OpenMP task parallelism template

- don't forget to create a **parallel** region
- don't also forget to enter a **master** region, which says only the master executes the following statement and others “stand-by”

```
1 int main() {  
2     #pragma omp parallel  
3     #pragma omp master  
4     // or #pragma omp single  
5     ms(a, a + n, t, 0);  
6 }
```

- and create tasks in the master region

```
1 void ms(a, a_end, t, dest) {  
2     if (n == 1) {  
3         ...  
4     } else {  
5         ...  
6         #pragma omp task  
7         ms(a, c, t, 1 - dest);  
8         #pragma omp task  
9         ms(c, a_end, t + nh, 1 - dest);  
10        #pragma omp taskwait  
11        ...  
12    }
```

Visualizing task parallel schedulers

- the workload is exactly the same as before

```
1 #pragma omp for collapse(2) schedule(runtime)
2 for (i = 0; i < 1000; i++)
3     for (j = 0; j < 1000; j++)
4         unit_work(i, j);
```

- but we rewrite it into recursions

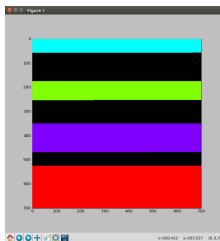
```
1 void work_rec(rectangle b) {
2     if (small(b)) {
3         ...
4     } else {
5         rectangle c[2][2];
6         split(b, c); // split b into 2x2 sub-rectangles
7         for (i = 0; i < 2; i++) {
8             for (j = 0; j < 2; j++) {
9                 #pragma omp task
10                    work_rec(b[i][j]);
11            }
12        }
13        #pragma omp taskwait
14    }
15 }
```

load $\sim [100, 10000]$ clock

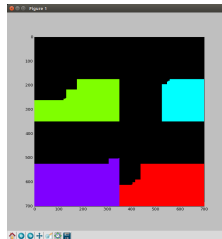


load ≈ 0

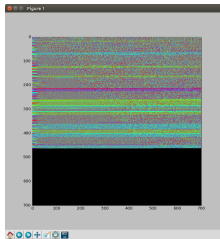
Visualizing schedulers



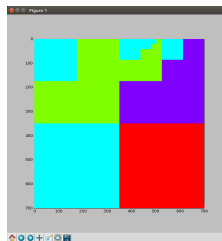
static



2D recursive (midway)



dynamic



2D recursive (end)

A note on current GCC task implementation

- the overhead seems high and the scalability seems low, so it's not very useful now
- TBB, Cilk, and MassiveThreads-backed TBB are all much better
- Intel implementation of OpenMP tasks is good too
- we'll come back to the topic of efficient implementation of task parallelism later

Pros/cons of schedulers

- **static:**

- partitioning iterations is simple and does not require communication
- may cause load imbalance (leave some threads idle, even when other threads have many work to do)
- mapping between work \leftrightarrow thread is deterministic and predictable (why it's important?)

- **dynamic:**

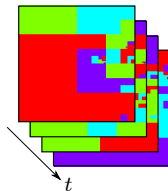
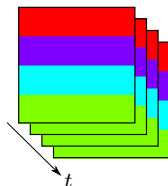
- no worry about load imbalance, if chunks are sufficiently small
- partitioning iterations needs communication (no two threads execute the same iteration) and may become a bottleneck
- mapping between iterations and threads is non-deterministic
- OpenMP's dynamic scheduler is inflexible in partitioning loop nests

Pros/cons of schedulers

- recursive (divide and conquer + tasks):
 - no worry about load imbalance, as in dynamic
 - distributing tasks needs communication, but efficient implementation techniques are known
 - mapping between work and thread is non-deterministic, as in dynamic
 - you can flexibly partition loop nests in various ways (e.g., keep the space to square-like)
 - need boilerplate coding efforts (easily circumvented by additional libraries; e.g., TBB's `blocked_range2d` and `parallel_for`)

Deterministic and predictable schedulers

- programs often execute the same for loops many times, with the same trip counts, and with the same iteration touching a similar region
- such *iterative* applications may benefit from reusing data brought into cache in the previous execution of the same loop
- a deterministic scheduler achieves this benefit



Contents

- 1 Overview
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Data sharing

- `parallel`, `for`, `task` pragma accept clauses specifying which variables should be shared among threads or between the parent/child tasks
- 2.14 “Data Environments”
 - `private`
 - `firstprivate`
 - `shared`
 - `reduction` (only for `parallel` and `for`)
 - `copyin`

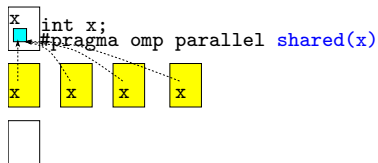
Data sharing example

```
1 int main() {
2     int S; /* shared */
3     int P; /* made private below */
4     #pragma omp parallel private(P) shared(S)
5     {
6         int L; /* automatically private */
7         printf("S at %p, P at %p, L at %p\n",
8             &S, &P, &L);
9     }
10    return 0;
11 }
```

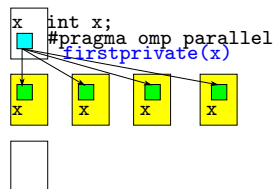
```
1 $ OMP_NUM_THREADS=2 ./a.out
2 S at 0x..777f494, P at 0x..80d0e28, L at 0x..80d0e2c
3 S at 0x..777f494, P at 0x..777f468, L at 0x..777f46c
```

Data sharing behavior

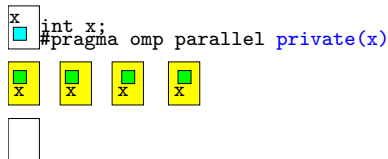
shared



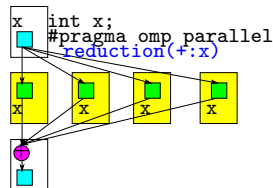
firstprivate



private



reduction



Reduction

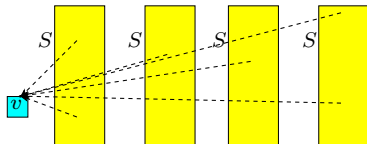
- in general, “reduction” refers to an operation to combine many values into a single value. e.g.,

- $v = v_1 + \dots + v_n$
- $v = \max(v_1, \dots, v_n)$
- ...

- simply sharing the variable (v) does not work (race condition)
- even if you make updates atomic, it will be slow (by now you should know how slow it will be)

```
1 v = 0.0;
2 for (i = 0; i < n; i++) {
3     v += f(a + i * dt) * dt;
4 }
```

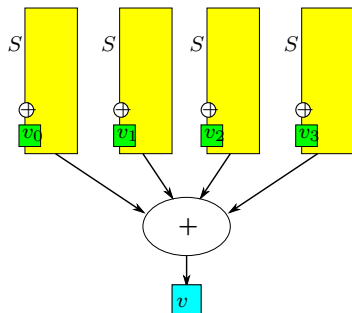
```
1 v = 0.0;
2 #pragma omp parallel shared(v)
3 #pragma omp for
4 for (i = 0; i < n; i++) {
5     #pragma omp atomic
6     v += f(a + i * dt) * dt;
7 }
```



Reduction clause in OpenMP

- a more efficient strategy is to
 - let each thread reduce on its private variable, and
 - when threads finish, combine their partial results into one
- reduction clause in OpenMP does just that

```
1  v = 0.0;  
2  #pragma omp parallel shared(v)  
3  #pragma omp for reduce(+:v)  
4  for (i = 0; i < n; i++) {  
5      v += f(a + i * dt) * dt;  
6  }
```



Simple reduction and user-defined reduction

(2.7.1)

- reduction syntax:

```
1 #pragma omp parallel reduction(op:var,var,...)  
2     S
```

- builtin reductions
 - *op* is one of +, *, -, &, ^, |, &&, and ||
 - (Since 3.1) min or max
- (Since 4.0) a user-defined reduction name

- user-defined reduction syntax:

```
1 #pragma omp declare reduction (name : type : combine statement)
```

User-defined reduction example

```
1  typedef struct {
2      int x; int y;
3  } point;
4  point add_point(point p, point q) {
5      point r = { p.x + q.x, p.y + q.y };
6      return r;
7  }
8  // declare reduction "ap" to add two points
9  #pragma omp declare reduction(ap: point: omp_out=add_point(omp_out,
10                                     omp_in))
11
12 int main(int argc, char ** argv) {
13     int n = atoi(argv[1]);
14     point p = { 0.0, 0.0 };
15     int i;
16     #pragma omp parallel for reduction(ap : p)
17     for (i = 0; i < n; i++) {
18         point q = { i, i };
19         p = add_point(p, q);
20     }
21     printf("%d %d\n", p.x, p.y);
22     return 0;
23 }
```

Contents

- 1 Overview
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

SIMD constructs

- `simd pragma`
 - allows an explicit vectorization of for loops
 - syntax restrictions similar to `omp for` pragma apply
- `declare simd pragma`
 - instructs the compiler to generate vectorized versions of a function
 - with it, loops with function calls can be vectorized

simd pragma

- basic syntax (similar to omp for):

```
1  #pragma omp simd clauses
2  for (i = ...; i < ...; i += ...)
3      S
```

- clauses
 - `aligned(var,var,... : align)`
 - `uniform(var,var,...) says variables are loop invariant`
 - `linear(var,var,... : stride) says variables have the specified stride between consecutive iterations`

declare simd pragma

- basic syntax (similar to omp for):

```
1 #pragma omp declare simd clauses  
2 function definition
```

- clauses
 - those for simd pragma
 - notinbranch
 - inbranch

SIMD pragmas, rationales

- most automatic vectorizers give up vectorization in many cases
 - ① conditionals (lanes may branch differently)
 - ② inner loops (lanes may have different trip counts)
 - ③ function calls (function bodies are not vectorized)
 - ④ iterations may not be independent
- `simd` and `declare simd` directives should eliminate obstacles 3 and 4 and significantly enhance vectorization opportunities

A note on current GCC OpenMP SIMD implementation

- as of now (version 4.9), GCC `simd` and `declare simd` \approx existing auto vectorizer – dependence analysis
- `declare simd` functions are first converted into a loop over all vector elements and then passed to the loop vectorizer

```
1 #pragma omp declare simd  
2 float f(float x, float y) {  
3     return x + y;  
4 }
```

→

```
1 float8 f(float8 vx, float8 vy) {  
2     float8 r;  
3     for (i = 0; i < 8; i++) {  
4         float x = vx[i], y = vy[i]  
5         r[i] = x + y;  
6     }  
7     return r;  
8 }
```

- the range of vectorizable loops in current GCC (4.9) implementation seems very limited
 - innermost loop with no conditionals
 - doubly nested loop with a very simple inner loop