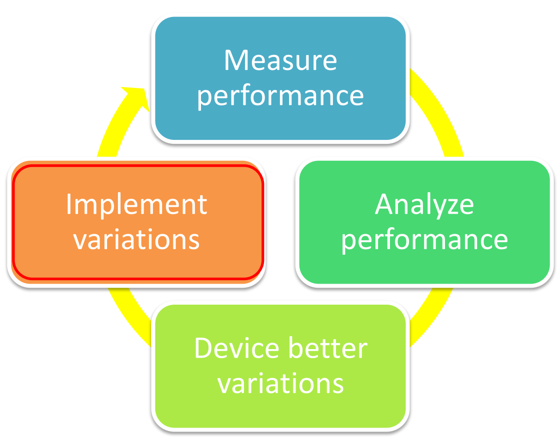
Reiji Suda

HPC, PARALLEL, AT

HPC people's cycle



- There are deterministic improvements and nondeterministic tunings
- We want to apply deterministic improvements in all the cases they are applicable

Dead code

- Commercial / multi-purpose codes…
  - implement many functionalities (e.g. fluid, structure, heat, magnetics, various FEM elements)
- In-house codes…
  - have many experimental functionalities (e.g. trying a new model, algorithm, or discretization)
- There are many unused fragments of codes
  - Which part is used is chosen by a part of input data
- Unused fragments may hinder optimizations

Typical cases

```
for each grid point
    compute this
if (model A is used)
    do a very complex thing
for each grid point
    compute that
```

↓ if (model A is not used)

```
for each grid point
    compute this
for each grid point
    compute that
```

↓ just a loop fusion

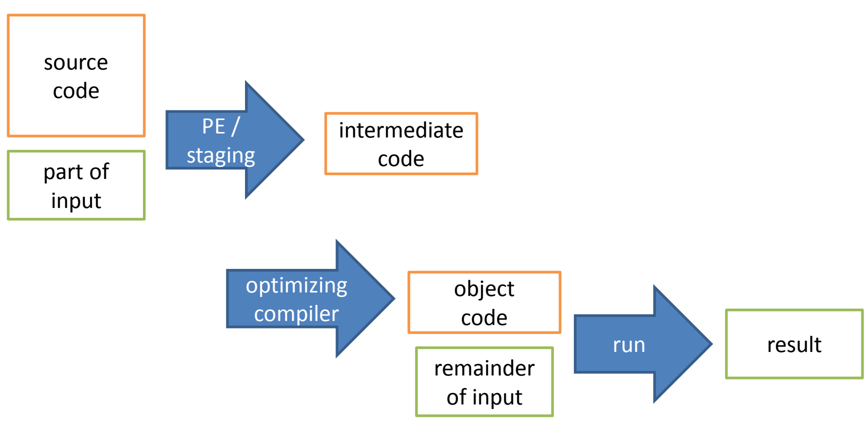
```
for each grid point
    compute this and that
```

```
if (model is A)
    compute this way
else if (model is B)
    compute that way
compute something
if (model is A)
    compute this way
else if (model is B)
    compute that way
```

↓ if (model A is used)

```
compute this way
compute something
compute this way
```

What I want



* HPC people would prefer explicit annotation, to declare "I want this place to be optimized"

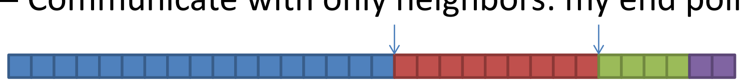
Unused code in libraries

- Libraries are destined to be general
  - Many parts are unused
- Sometimes we know special optimal method for special cases

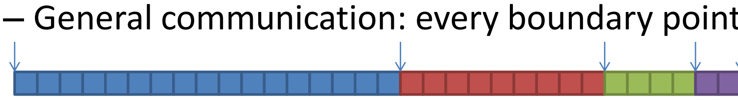
Ex. domain decomposition

- Fixed-size decomposition: block size

- Variable-size decomposition
  - Communicate with only neighbors: my end points



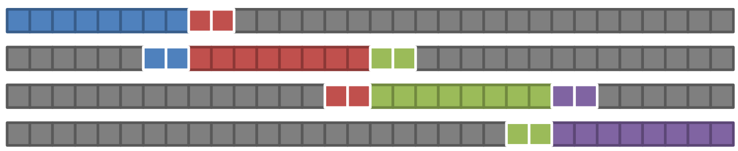
  - General communication: every boundary points



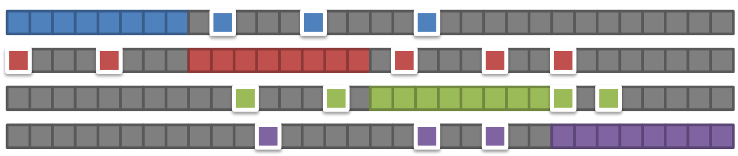
  - It could be big difference for 1,000,000 cores

Ex. shadow region

- If the shadow is adjacent to the body region: allocate a little bigger array



- Otherwise 1) compaction, 2) full shadow



Dead code in generated codes

- Assume that we have a general method A and a specialized method B on condition P
- Which is easier?
  - Static check – code generator checks whether P always holds or not and generate either A or B
  - Runtime check – code generator outputs "if P then B else A" (and let PE choose one)
  - Maybe, it just outputs A, a general solution

- There are deterministic improvements and nondeterministic tunings
- We want to try nondeterministic tunings to find whether they are effective or not

No optimal solution in HPC

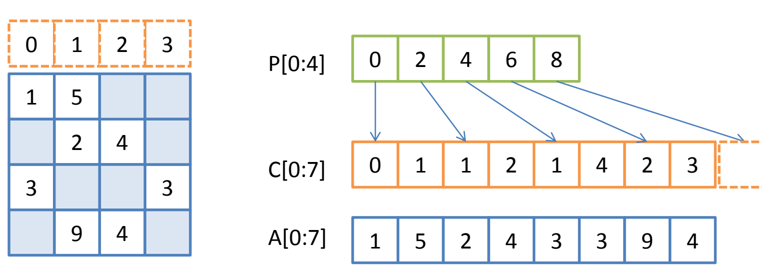
- Performance depends on conditions
  - HW conditions
    - Cache size, #cores, memory latency, network speed…
  - SW conditions
    - Library performance, working memory size…
  - Data conditions
    - Size of matrix, values, graph structures…
  - Environmental conditions
    - Other users, other processes…
- Tuning must be empirical

Ex. BLAS

- BLAS provides a high-performance implementation of basic linear algebra routines
- Can be 100x faster than naïve code
  - But usually tuned for very large matrices
- Could be 100x slower than naïve code for very small matrices (e.g. 3x3 or 4x4)

Ex. Sparse matrix

- There are data structures which stores only non-zero elements
- However if sparsity is not enough, it is more efficient to store it as a dense matrix

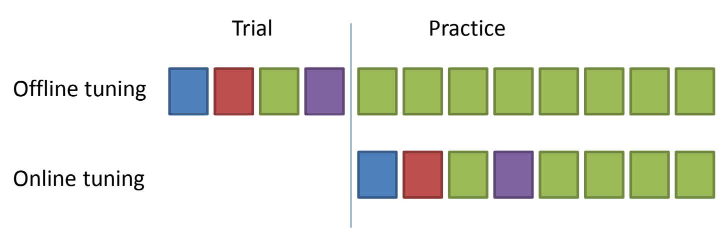


Automatic tuning

- Several variations of the same computation are programmed
- Performance is automatically measured, and a well performing one is automatically selected

- Problems
  - How to generate variations?
  - How to select a good variation?

Online automatic tuning



- Lower total cost
- Higher optimization if used many times
- Lower optimization is enough if used a few times

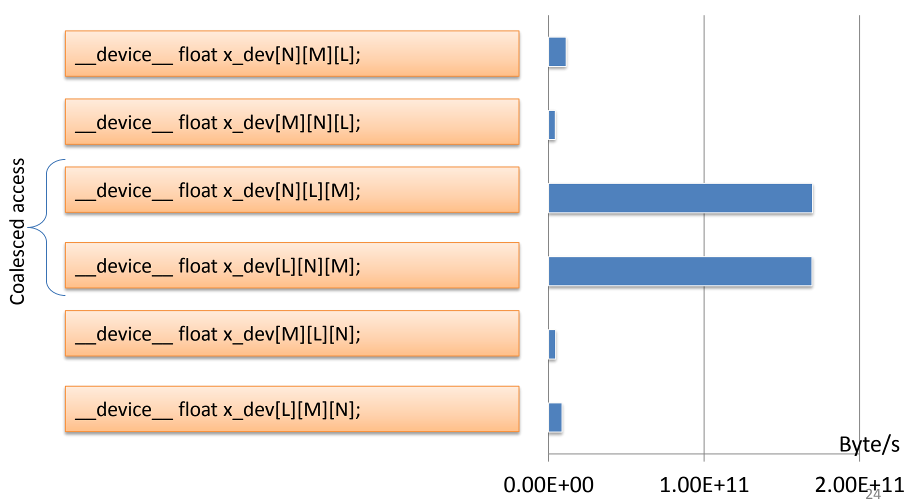
Dynamic generation?

What kind of variations?

- Scheduling variations
  - unrolling, code motion, loop transformation, software pipelining
  - scheduling in parallel processing
- Data structure variations
  - array dimension, padding, skewing, space-filling order, (recursive) block indexing, reordering
  - list vs array, array-of-struct vs struct-of-array, object inlining
  - distributed data structure, software cache
- Algorithmic variations
  - different algorithm, preprocessing, parallelization, mixed precision
- Platform specific coding
  - message passing, short vector instructions, GPU etc.
- Data structure vs code generation, storage vs recomputation

Array dimension

- Read L=16 block x M=1024 thread x N=16K word (GPU)



Array of struct / struct of array

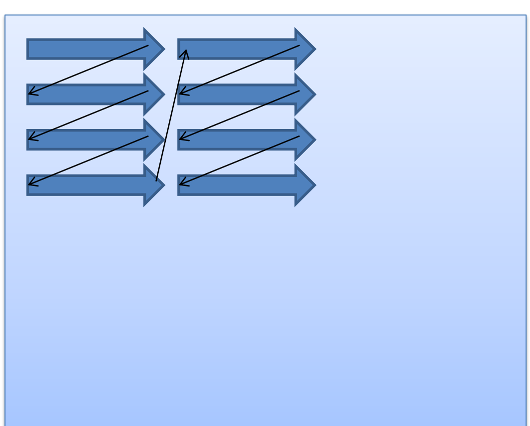
```
struct {
  double a, b, c, d;
} array[N];
```

Better spatial locality if all fields are accessed at once

```
struct {
  double a[N], b[N], c[N], d[N];
} array;
```

Better spatial locality if only small part of fields are accessed at once
Almost mandatory in GPUs (not coalesced)

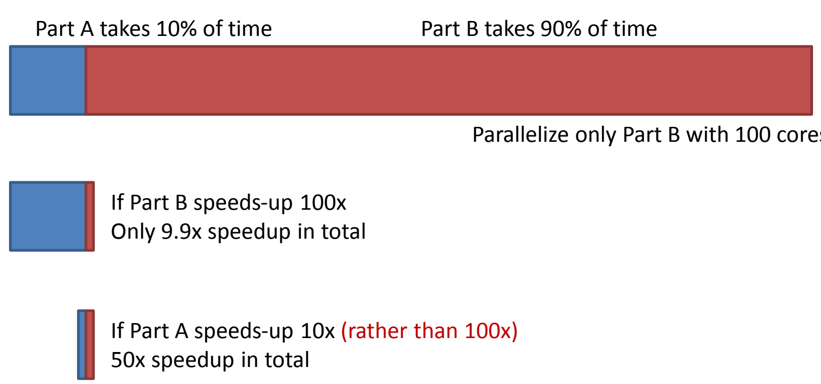
Block indexing



Parallel processing

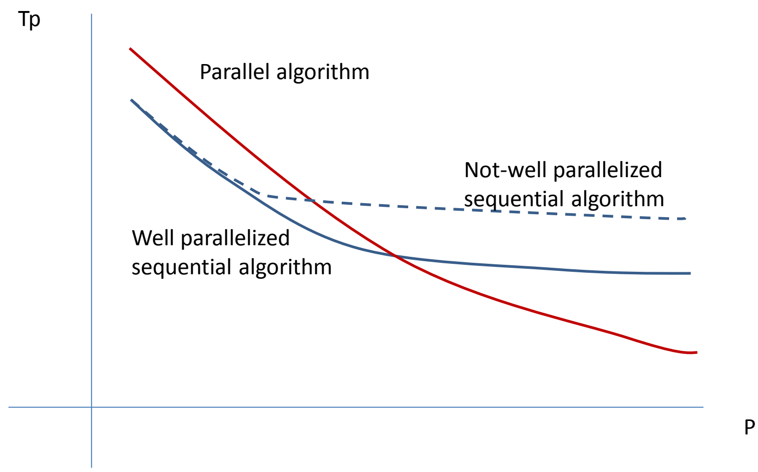
- Moore's law continues, but…
  - The clock frequency will not improve much (because of power & cooling limitation)
  - Performance only comes from parallelism
- Free lunch is over
- In 10 years, processors have 20 ~ 200 cores
  - Everyone needs to do parallel programming

Amdahl's law



Need to parallelize dirty 90%, low efficiency allowed

Reviving parallel algorithms?



What I want

- Formulating HPC methods in reusable components, parameterized
  - Program transform or program generation, and hand-written alternative
- Generating multiple specialized versions of functions / classes from one piece of code
  - And dynamic selection
- Enumerative / dynamic generation
- Support for debugging and testing
  - In case of error, select another