The perfectly parallel program

Design philosophy for parallel programming

Zenny Wettersten (CERN, TU Wien)

Recap

Vectorised hardware and heterogeneity

- Vectorised (SIMD/SIMT) instructions
  - Performs same operation on many data simultaneously
  - Simultaneity requires "non-divergent"[1] program flow
  - If code divergence depends on data; should factorise such branches
  - Vectorised $\neq$ heterogeneous; Local SIMD CPU, or offload to a device
- Heterogeneous computing
  - Offload part of program to external device
  - Communication becomes limiting factor in possible speedup

[1] Code may branch, but all data in the same batch should take the same data path.

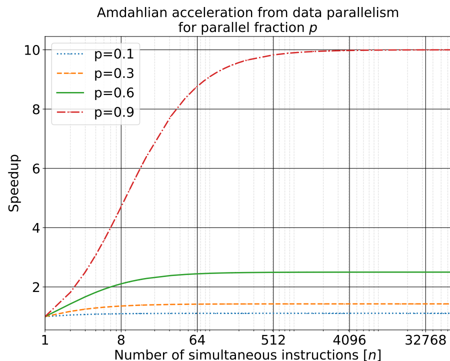
General rules for parallel and heterogeneous programs

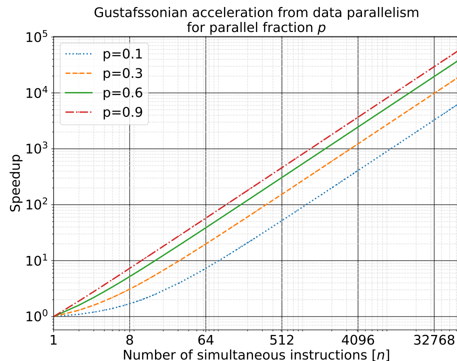
- Maximise parallelism, minimise sequential code
  - The vectorised sections should be *as long as possible*
  - Factorise/remove conditionals from vectorised code
- Localise communication, avoid synchronisation
  - Data transfer between devices should happen as few times as possible
  - Better to send all data as once than passing back and forth
  - Larger sequential sections might be preferable if it avoids communication
- Work with your available hardware
  - Multi-core CPUs allow multithreading, does little[2] for data-parallelism
  - SIMD CPUs give free speedup, but conditionals are destructive
  - Most GPUs are very bad at double precision arithmetic

[2] Multithreaded data parallelism can be preferable to task parallelism when singular jobs are bottlenecks.

Defining parallelism

Reminder: Amdahl's and Gustafsson's laws





$$\text{Speedup} = \left.\frac{t_{par}}{t_{seq}}\right|_{\text{data}}$$

$$\text{Speedup} = \left.\frac{\text{data}_{par}}{\text{data}_{seq}}\right|_{t}$$

Examples of data parallel methods in HEP

- Monte Carlo event generation
  - Both embarrassingly parallel and (mostly) lockstep
  - Involves same amplitude evaluation across many phase space points
- Simulations
  - Detector simulations highly branching, but still embarrassingly parallel
  - GPUs have huge throughput — alright to throw away some results
  - GPUs fundamentally image renderers: Made for optical particles
- On- and offline signal analysis
  - (Triggering and tagging suited for ML[3])[4]
  - Vertex reconstruction involves interpolating thousands of tracks
  - Fitting data means minimising NLL over parameter space

---

[3] Complex problems with large data reduction and an a priori unknown solution.

[4] Additionally, FPGAs are perfect for the fast throughput needed for triggers.

tl;dr:

Parallel algorithms have independent steps and can be performed in lockstep, which a lot (but not all!) HEP code fulfills

Writing parallel programs

Checking the fairness of a coin

Problem formulation: We have a coin simulator `coinFlip()`, which when called returns either heads (+1) or tails (-1). We do not know a priori that `coinFlip()` is fair (i.e. has an exactly 50/50 chance of returning either heads or tails), and while we do not know the inner mechanism of the simulator we have been assured that it is thread-safe and branch-free.
Now we want to statistically test whether `coinFlip()` is fair.

Question 4

What type of problem is this?
Is data-parallelism a good solution?

Answer

Example program solution:

```
1:  procedure COINISFAIR(n)
2:      flips ← ZEROS(size = n)
3:      mcDev ← 1/SQRT(n)
4:      for i = 1, . . . , n do            ▷ This loop is trivially data parallel.
5:          flips(i) ← COINFLIP( )
6:      avg ← SUM(flips)
7:      if |avg| < mcDev then
8:          WRITE(The coin seems fair.)
9:      else
10:         WRITE(The coin seems unfair.)
```

More complicated example: Finding an unfair coin?

Problem formulation: We now have a multiple coin simulator `coinsFlip(k)` which takes an integer $0 \leq k \leq 9$ to decide which of ten coins to flip. Like before, each single coin is thread-safe and branch-free, but `coinsFlip(k)` itself branches based on the argument. We have been told that at least one of the coins is unbalanced, but we do not know which or how unbalanced it is. We want to determine which coin(s) is/are unfair.

Question 5

How do we adapt our previous method to this branching case?

Answer

Example program solution:

```
1: function COINISFAIR(n, k)
2:     flips ← ZEROS(size = n)
3:     mcDev ← 1/SQRT(n)
4:     for i = 1, ..., n do
5:         ▷ This loop can trivially be made data parallel. ◁
6:         flips(i) ← COINSFLIP(k)
7:     avg ← SUM(flips)/n
8:     return (|avg| < mcDev)
```

```
1: procedure FAIRCHECKER(n)
2:     isFair ← TRUE(size = 10)
3:     for i = 0, ..., 9 do
4:         ▷ This loop can be thread parallel. ◁
5:         isFair[i] ← COINISFAIR(n, i)
6:     for i = 0, ..., 9 do
7:         if isFair(i) then
8:             WRITE(Coin i seems fair.)
9:         else
10:            WRITE(Coin i seems unfair.)
```

Related example: *How* unfair are our coins?

Problem formulation: Our new simulator `coinsFlip(p,k)` takes as input a real number $p \in [0, 1]$ and a positive integer $k > 0$. `coinsFlip` now contains an unknown number of coins, and p is the "seed" determining which coin is used. Furthermore, k states how many times the given coin will be flipped, and `coinsFlip` returns the sum of the results of k consecutive coin flips.
Since we no longer know how many coins there are, we now simply want to determine how likely it is that a given coin is unfair, and how unfair such a coin is.

Question 6

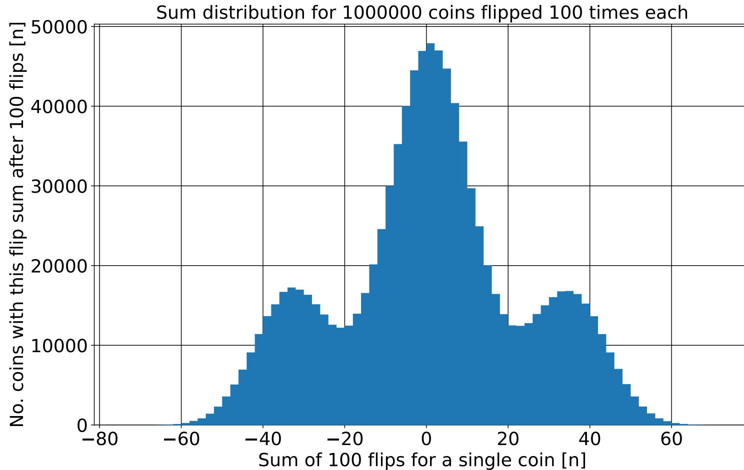
How should we test our possibly infinite amount of coins?

Answer

Example program solution:

```
1: function COINFAIR(n, p)
2:     flips ← ZEROS(size = n)
3:     for i = 1, ..., n do
4:         ▷ This loop can trivially be made data parallel. ◁
5:         flips(i) ← COINSFLIP(p)
6:     return SUM(flips)
```

```
1: procedure FAIRTESTER(n,k)
2:     seeds ← RNG(size = k)
3:     for i = 1, ..., k do
4:         ▷ This loop can be thread parallel. ◁
5:         sum[i] ← COINFAIR(n, seeds[i])
6:     HISTOGRAM(RANGE(−n, n), sum)
```



Histogram showing the results from an implementation of the previous slide's pseudocode. Notice the three slightly overlapping Gaussian distributions, similar to Breit-Wigner peaks in a momentum distribution. As an estimate, we can normalise the peaks to the sums of the peaks to get a 58% probability of a fair coin and 21% to get an unfair coin in either direction.

Frameworks and APIs for parallelism

Non-exhaustive list of options

- Multithreading with OpenMP
- Vectorised SIMD CPU instructions
- Explicit heterogeneous APIs
- Portability frameworks/abstraction layers

Multithreading loops (with e.g. OpenMP) 101

- Wrap `for`/`do` loops in parallel imperative
- Informs compiler that iterations of a block are independent
- OpenMP: Uses comments/pragmas (example from Wikipedia):

```
int main(int argc, char **argv){
    int a[100000];
    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }
    return 0;
}
```

Vector instructions (e.g. AVX) 101

- Most modern CPUs have some SIMD instructions implemented
- E.g. `gcc` has implemented auto-vectorisation since `4.0`
  - Compiler flag `-ftree-vectorize` (with some implied sub-flags)
  - Implied by `-O3` optimisation flag since `GCC 4.3`
  - Implied by `-O2` optimisation flag since `GCC 12.1`
  - You might have used auto-vectorisation without knowing it!
- Relies on compiler properly identifying vectorisable blocks
  - Make parallel blocks clear (e.g. indexed arrays, public variables `const`)
  - Clarify ownership of variables (`restrict` keyword, `unique_ptr`)
- Experimental `std::simd` (C++) for explicit SIMD vectorisation
- (OpenMP: `#pragma omp simd` directive for explicit SIMD)

CUDA and HIP 101

- Explicit APIs for specific hardware manufacturers[5]
  - CUDA/HIP (primarily) based on C++, can compile C++ code
  - Using generic macros translated to specific backends at compile time, can achieve some level of portability
- Makes distinct separation between host code and device code
- Explicit memory management — allocation, transfer, frees
- Device tasks run exactly as written
- Not portable; needs to be rewritten for each architecture
  - Due to historical NVidia dominance, most APIs share CUDA's structure: Porting *often* (but not always) simple

[5] HIP (AMD) can compile directly to NVidia devices as of recently.

Portability frameworks 101

- Code wrappers, translates to relevant device behind the scenes
- Examples: alpaka, Kokkos, RAJA, SYCL
- Same code can be compiled for different architectures
- Memory management (partially) handled by the framework
- Framework "translates" code to the relevant form
  - The exact form of the compiled code may differ from written
- By construction, portable: Compiles to any supported hardware
  - Aims for performance portability; depends on the framework

	CUDA	HIP	OpenMP Offload	Kokkos	dpc++ / SYCL	alpaka	std::par
NVidia GPU					*codeplay and intel/llvm*		*nvc++*
AMD GPU				*feature complete for select GPUs*	*via openSYCL and intel/llvm*	*hip 4.0.1 / clang*	
Intel GPU		*CHIP-SPV early prototype*		*native and via OpenMP target offload*		*prototype*	*oneAPI::dpl*
multicore CPU							*g++ & tbb*
FPGA						*via SYCL*	

Figure taken from arXiv:2306.15869 without modification (probably outdated). Dark green means full support, light green partial support or current development, and red no support.

Question 7

When programming for heterogeneous parallel computing, which framework should I use?

Answer

The one your team is using.

No dominant standard has been established, so use whatever you're comfortable with within your restrictions.
Just make sure to keep up to date with the options.

Legacy code

Rewriting existing software with data parallelism

- In reality, we rarely write software from scratch
- Working on existing programs, parallelisation has several stages
- We need to identify which parts are suited for porting
- Then, need to refactor the program and extract that part
- Once extracted, rewrite section for parallel architectures
- ...that's it, right?

Reminder:

Code is never slow until it has been profiled!

Profiling

- Even if a program section is easily parallelised, *there's no point in rewriting it if it's not a bottleneck*
- Identifying relevant code sections is primarily about profiling
- Only once hotspots are found should you consider how the code can be accelerated
- Non-exhaustive list of profiling tools I use:
  - perf: Sampling, results (proportionally) representative to real runtimes
  - Callgrind (Valgrind): Tracing, gives exact instruction counts
  - GDB: Debugger, allows for exact program runtime debugging

Flame Graph

ffv1_0_ f.. f.. vvv..

bo.. b.. sb.. sloopma.. matrix_1_

sb.. b.. sb.. sloopma.. born_ smatrix1_splitorders_ ffv.. ffv1.. ffv1.. v..

sb.. sb.. ge.. sloopma.. sbo.. smatrix1_ sborn_sf_ born_

sb.. sb.. xm.. binothlha_ sbo.. sborn_sf_ smatrix_real_ sbornsoft_ sborn_splitorders_

sre.. sr.. co.. bornsoftvirtual_ sreal_ sr.. sreal_ sborn_

compute.. co.. compute_nbody_noborn_ compute_real_emission_ compu.. compute_.. include_multichannel_enhance_

sigintf_

__mint_module_MOD_compute_integrand

[f.. __mint_module_MOD_mint n..

top.. MAIN__ n..

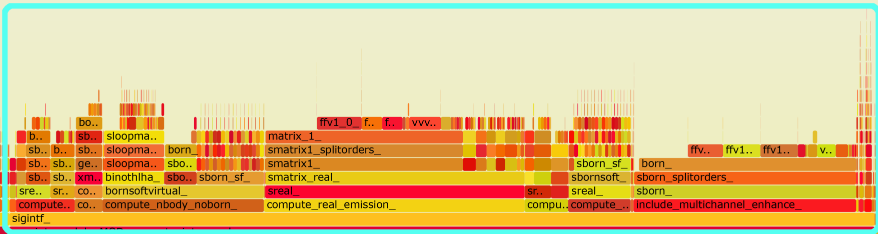
main main n..

__l.. __libc_start_main l..

_st.. _start loop_ct..

f951 madevent_mintMC

Flame Graph



Refactoring

- Once you've identified hotspots, time to factor it out
- Difficulty of this task entirely program dependent:
  Can go from trivial to a complete code overhaul
- Essentially, your goal is to put the code you aim to parallelise in a singular block/loop that can easily be extracted and replaced with calls to external functions/libraries
- Here you also want to remove dependencies on global variables; this block should only rely on data you have full knowledge of
- Once *this* is done, you can actually start porting

Running in parallel

- With parallel block separated from the rest of the program, it's time to write your parallel code
- For directive-based frameworks (e.g. OpenMP), this is trivial
  - Although you will certainly find more errors once you start compiling
- For language extensions, this means rewriting your block/function using whatever framework you've decided on
- Depending on the API, the final code likely looks similar to what you started with, except you can now run it on your device!

Legacy code example, sigint from Flamegraph

```fortran
1      double precision function sigint(xx,vegas_wgt,ifl,f)
2      ...
3      if (abrv.eq.'real') goto 11
4      nbody=.true.
5      calculatedBorn=.false.
6      call get_born_nFKSprocess(nFKS_picked,nFKS_born)
7      call update_fks_dir(nFKS_born)
8      if (ini_fin_fks(ichan).eq.0) then
9          jac=1d0
10     else
11     jac=0.5d0
12     endif
13     call generate_momenta(nndim,iconfig,jac,x,p)
14     if (p_born(0,1).lt.0d0) goto 12
15     call compute_prefactors_nbody(vegas_wgt)
16     call set_cms_stuff(izero)
17     if (ickkw.eq.3) call set_FxFx_scale(1,p1_cnt(0,1,0))
18     passcuts_nbody=passcuts(p1_cnt(0,1,0),rwgt)
19     if (passcuts_nbody) then
20         pass_cuts_check=.true.
21         call set_alphaS(p1_cnt(0,1,0))
22         call include_multichannel_enhance(1)
23         if (abrv(1:2).ne.'vi') call compute_born
24         if (abrv.ne.'born') call compute_nbody_noborn
25     endif
26     11   continue
27     if (abrv(1:4).eq.'born'.or.abrv(1:4).eq.'bovi'
28         $.or.abrv(1:2).eq.'vi') goto 12
29     nbody=.false.
30     if (sum) then
31         nFKS_min=1
32         nFKS_max=ini_fin_fks_map(ini_fin_fks(ichan),0)
33         MC_int_wgt=1d0
34     else
35         nFKS_min=iran_picked
36         nFKS_max=iran_picked
37         MC_int_wgt=1d0/vol
38     endif
39     do i=nFKS_min,nFKS_max
40         iFKS=ini_fin_fks_map(ini_fin_fks(ichan),i)
41         ...
42         call update_fks_dir(iFKS)
43         call generate_momenta(nndim,iconfig,jac,x,p)
44         if (p_born(0,1).lt.0d0) cycle
45         call compute_prefactors_n1body(vegas_wgt,jac)
46         call set_cms_stuff(izero)
47         if (ickkw.eq.3) call set_FxFx_scale(2,p1_cnt(0,1,0))
48         passcuts_nbody =passcuts(p1_cnt(0,1,0),rwgt)
49         call set_cms_stuff(ione)
50         passcuts_coll=(use_evpr.and.passcuts_nbody).or.passcuts(p1_cnt(0,1,1),rwgt)
51         call set_cms_stuff(mohdr)
52         if (ickkw.eq.3) call set_FxFx_scale(3,p)
53         passcuts_n1body=passcuts(p,rwgt)
54         if (passcuts_nbody .and. abrv.ne.'real') then
55             pass_cuts_check=.true.
56             call set_cms_stuff(izero)
57             call set_alphaS(p1_cnt(0,1,0))
58             call include_multichannel_enhance(3)
59             call compute_soft_counter_term(0d0)
60             call set_cms_stuff(itwo)
61             call compute_soft_collinear_counter_term(0d0)
62         endif
63         if (passcuts_coll .and. abrv.ne.'real') then
64             call set_alphaS(p1_cnt(0,1,1))
65             call set_cms_stuff(ione)
66             call compute_collinear_counter_term(0d0)
67         endif
68         if (passcuts_n1body) then
69             pass_cuts_check=.true.
70             call set_cms_stuff(mohdr)
71             call set_alphaS(p)
72             call include_multichannel_enhance(2)
73             call compute_real_emission(p,1d0)
74         endif
75     enddo
76     ...
77     return
78     end
```

Exercise: Making code parallel

Problem background

The tabletop RPG system *Dungeons & Dragons 5e* (5e) allows player characters (PCs) to go from level 1 up to 20. When facing an appropriately levelled non-player character (NPC), 5e is balanced such that PCs' attacks should hit NPCs roughly 65% of the time, whereas NPCs' attacks should hit PCs roughly 45% of the time. Whether an attack hits or not is determined by the roll of a 20-sided die (d20), although the exact calculation varies between different PCs and NPCs, as well as their levels.

We've developed a game using 5e, and we want to test its balance.

Test suite

Before sending our game out, we run a test to check the balance.

Our test suite has four different pre-built PCs, each of which can vary from level 1 through 20. To avoid biasing our tests, we have a function to choose a random NPC at an input level, which we then pass to another function that either checks if a given PC hits the NPC or if the NPC hits a given PC. Due to the stochastic nature of dice, we use random number generation also here.

The test suite runs this test n times for each combination of specific PC, PC level, and NPC level, and then plots a heatmap of the results.

Simplified test suite structure

```
1:  procedure testSuite(n)
2:      PCs ← {barbarian,cleric,rogue,wizard}
3:      hits ← false(4, 20, 20, n)
4:      hitRate ← zeros(4, 20, 20)
5:      for all class ∈ PCs do
6:          for i = 1, . . . , 20 do
7:              for j = 1, . . . , 20 do
8:                  for k = 1, . . . , n do
9:                      NPC ← randomNPC(i)
10:                     hits[class, i, j, k] ← hitPC(class(i), NPC)
11:                 hitRate[class, i, j] ← sum(hits[class, i, j])/n
12:         heatMap(hitRate[class])
```

Question 8

How should we go about parallelising this program?

Answer

That's up to you to solve in the exercise session ;)

Exercise information

- A Github repo with implemented code will be posted on Indico
- The code is written in C++ and runs sequentially
- A makefile is included, just write `make` in the terminal
- (All prerequisites are in the exercise pdf, also on Indico)
- Try to restructure and modify the program to run as parallel as possible for whatever machine you have available!
- (It is sufficient to use OpenMP on your local machine, the primary goal is to restructure the code)
- The repo has pre-generated heatmaps for the sequential code: Check whether you get the same statistics
- (Does the game fulfill our balance criteria?)

Any questions?

CERN