# 703308 VO High-Performance Computing WS2022/2023
# Expanding Horizons: Additional Programming Models

Philipp Gschwandtner

# Overview

- general discussion on programming models
  - characteristics
  - categorization
- examples
  - Pthreads, C++ STL threads, TBB, HPX, AllScale, OpenCL/CUDA, SYCL, Chapel, X10, Matlab

# Motivation

- functional portability
  - write once, run anywhere
  - guaranteed by MPI, OpenMP and alike – but is there more?
- performance portability
  - move optimization away from the program into the toolchain
  - enable automatic optimization
- "separation of concerns" principle states that
  - domain science experts should focus on their domain, not deal with HPC specifics
    - e.g. I should add memory padding for optimal cache usage?
    - What's the cache line size? And by the way, what's a cache?
  - computer science experts should know computer science

# A simple loop

```
// increment every element of
// an array with values 0-9
for(int i = 0; i < 10; i++) {
  A[i] += i;
}
```

# A simple parallel loop (MPI vs. AllScale)

```
int rank, size;
MPI_Com_rank(MPI_COMM_WORLD, &rank)
MPI_Com_size(MPI_COMM_WORLD, &size)
int p = 10/size;
MPI_Scatter(A,...)
for(int i = p*rank; i < p*rank+p; i++) {
  A[i-p*rank] += i;
}
MPI_Gather(A,...)
```

end of iterator range

start of iterator range

body function
(C++ lambda)

pfor operator
(generic function)

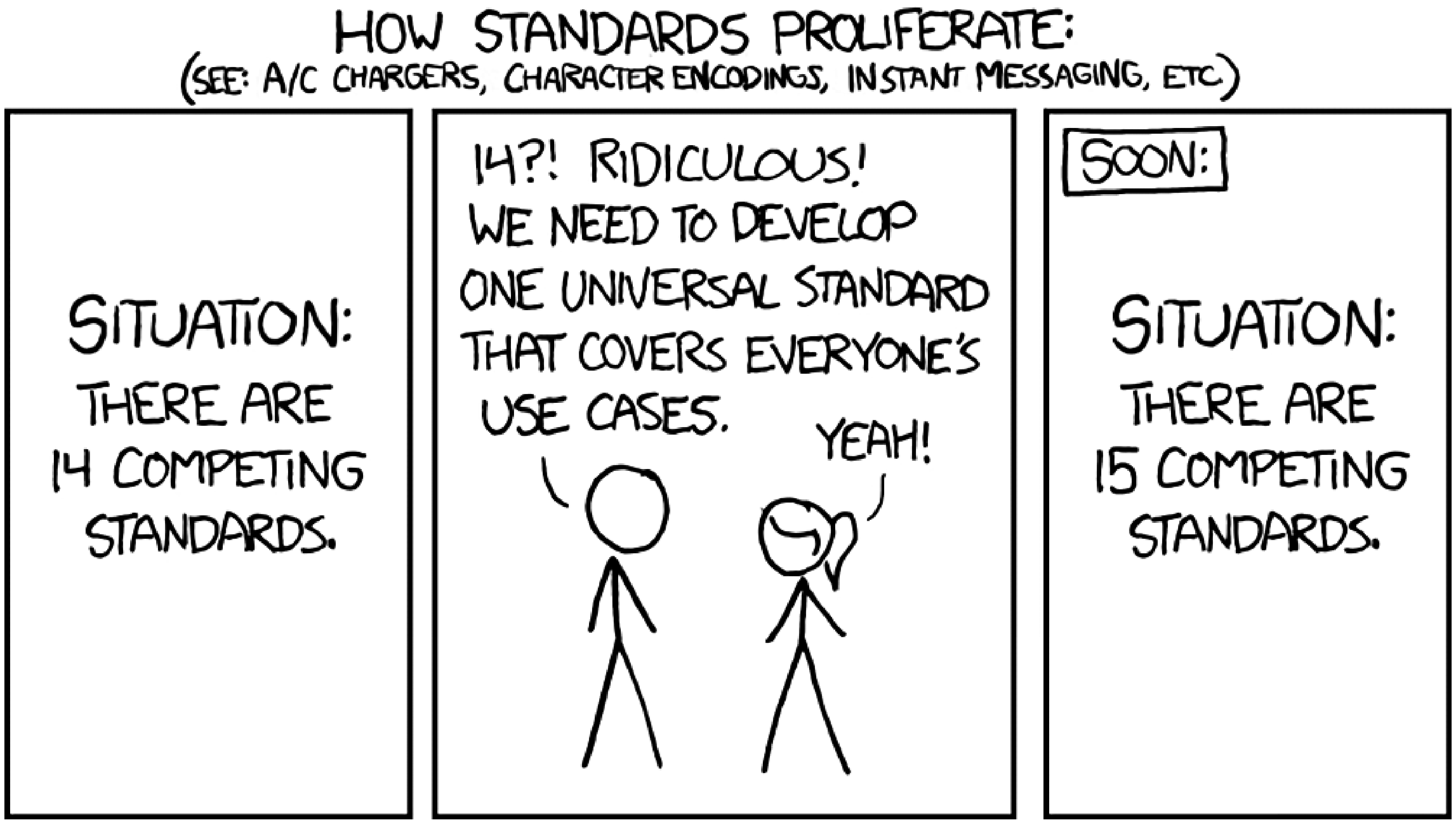
```
pfor(0, 10, [&](int i) {
    A[i] += i;
});
```

erator variable

A captured by reference

loop body

# The sad truth (xkcd 927)



# How to categorize programming models

- type of API/user interface
  - language? language extension? library?
- domain specificity
  - single use-case only? generic?
- target platform
  - distributed memory? shared memory? accelerators?
- features
  - intra-node scheduling? inter-node scheduling? data decomposition? data distribution? work decomposition? work distribution? fault tolerance? nested parallelism? making coffee?

# Underlying communication models

- shared memory
  - single memory address space
  - direct data access, synchronization via shared memory (e.g. locks, semaphores)
- message passing
  - multiple, distinct memory address spaces
  - data transfer and synchronization via message exchange (e.g. send/recv)
- Global Address Space (GAS)
  - single, global memory address space spanning multiple nodes
  - remote data access, synchronization similar to shared memory or implicit
  - often support active messages: “at location x, execute function y” (similar to remote procedure calls)

# Type of API (effort-benefit tradeoff)


- applications require less code
  - expressions hold more semantics
- optimizations reside in toolchain
  - application developer has less control
  - automatically apply to every program
- toolchain maintenance is large effort

- applications require more code
  - expressions hold less semantics
- optimizations (partially) in application
  - application developer has more control
  - need to be repeated per application
- toolchain maintenance is minimal

# Languages

- upside: new/own syntax
  - tailored towards the specific use-case (e.g. parallel task execution) or domain science (e.g. linear algebra) – Domain Specific Language (DSL)
  - stripped down to the required minimum
  - learn once, use intuitively without requiring a host language such as C++
    - note there's also embedded DSL (eDSL), which is technically a library but repurposes so much of the host language syntax, we might start calling it a language – funny example based on operator overloading: https://github.com/Quuxplusone/analog-literals/blob/master/readme.cpp

- downside: toolchain support
  - mandatory: compiler, runtime system, debugger, profiler, etc.
  - maintainer's view: any language optimizations have to be implemented separately per programming model (e.g. auto-vectorization)
  - no room for manual low-level optimization (e.g. vectorization, inline assembler)

# Language extensions

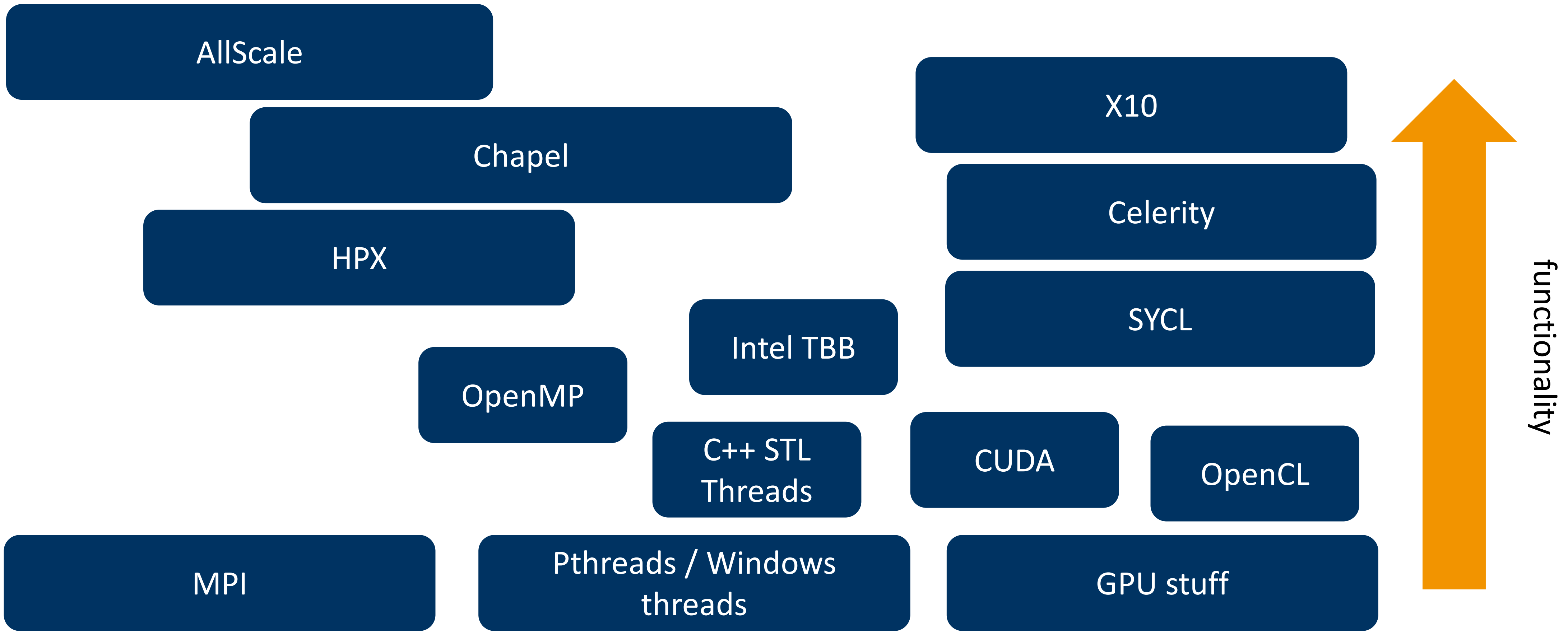
- upside: enrich an existing language with new features
  - builds upon a hopefully mature programming language
  - add use-case- or domain-specific features (e.g. OpenMP)
- downside: toolchain support
  - compiler and other tools need to be extended (albeit not written from scratch)
  - still needs to partially adhere to host language specification for compatibility

# Libraries

- upside: build completely on mature languages
  - toolchain support is already present
  - benefit from new optimizations in e.g. compiler research often without manual effort

- downside: encapsulation required
  - need to wrap the underlying, generic programming language (e.g. prevent accidental `double`-to-`int` conversions)
  - offer a (semi-)fixed-syntax API to the programmer, less flexibility
  - often requires measures to decrease debugging complexity
    - I'm looking at you, C++ templates!

# Programming model tetris



# Disclaimer

I have not previously worked with all of the following programming models.

Hence, although (most of) the following code examples are functional, there are very likely faster/prettier solutions.

More sophisticated models offer multiple ways of constructing a solution for a given problem – the implementations shown here are selected to facilitate a quick glance on the model and are not necessarily optimal.

# Libraries: Pthreads

- C library for thread-based parallelism
  - API for creating and destroying threads, synchronization and cancellation handling
  - not domain-specific, can be used for any type of program
  - is usually the underlying machinery for e.g. OpenMP

- targets shared memory only
  - no support for distributed memory or accelerators
  - no support for automatic work or data decomposition/distribution, scheduling, etc.
  - no support for fault tolerance

# Libraries: Pthreads example (optimized sum)

```c
#define NTHREADS 4
#define SIZE 1000000
#define SUBSIZE (SIZE/NTHREADS)
double sum = 0.0;
pthread_mutex_t sum_mutex;
void *partialSum(void *tid) {
  double mySum = 0.0;
  int start = (*((int*)tid)) * SUBSIZE);
  int end = start + SUBSIZE;
  for(int i = start; i < end ; i++) {
    mySum++; // local sum without lock
  }
  // global sum requires lock
  pthread_mutex_lock(&sum_mutex);
  sum += mySum;
  pthread_mutex_unlock(&sum_mutex);
  pthread_exit(NULL);
}
```

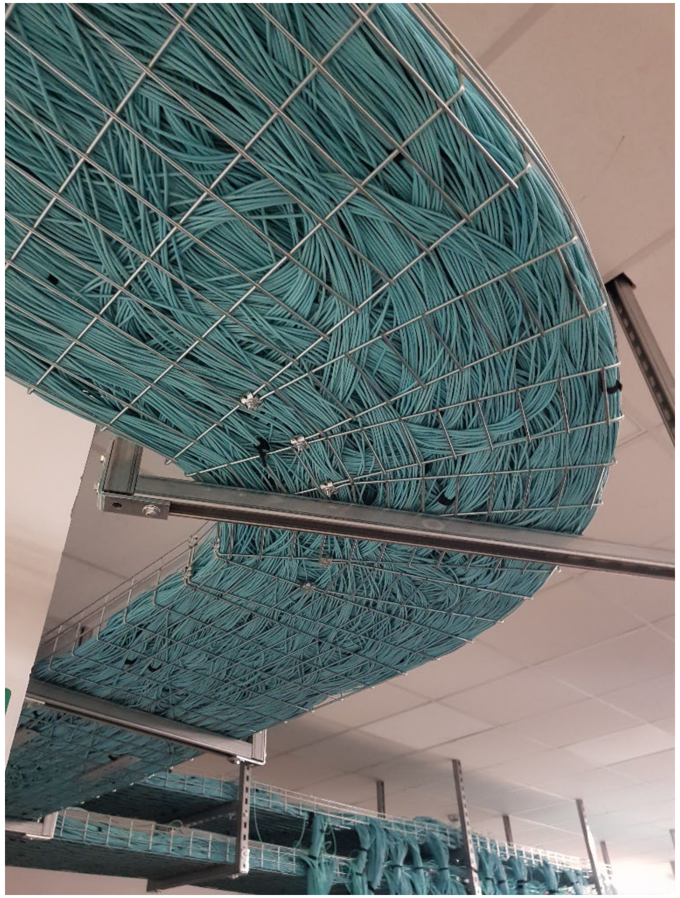
```c
int main(int argc, char *argv[]) {
  int tids[NTHREADS];
  pthread_t threads[NTHREADS];
  pthread_attr_t attr;
  pthread_mutex_init(&sum_mutex, NULL);
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,
    PTHREAD_CREATE_JOINABLE);
  for(int i = 0; i < NTHREADS; i++) {
    tids[i] = i;
    pthread_create(&threads[i], &attr, partialSum,
      (void *)&tids[i]); // start threads
  }
  for(int i = 0; i<NTHREADS; i++) {
    pthread_join(threads[i], NULL); // wait
  }
  printf("Done. Sum= %e \n", sum);
  /* cleanup ... */
}
```

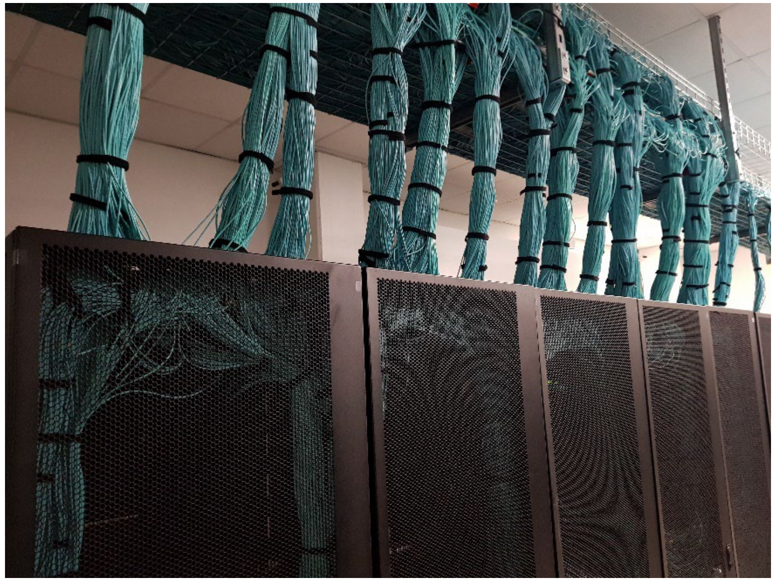
# Side-note: Why care about fault tolerance?

- MARCONI @ CINECA, Bologna
  - 250k cores
  - 20 Petaflop/s

- approx. 10.000 optical fibers connecting their HPC infrastructure

- approx. 1 cable failure per day!





# Libraries: C++ STL threads

- thread-based parallelism part of the STL since C++11
  - API similar to Pthreads with convenience functions and type safety
  - e.g. C++17 hint for avoiding false sharing
    `constexpr std::size_t hardware_destructive_interference_size`
  - also features mutex, semaphores, condition variables, futures, etc.

- targets shared memory only
  - no support for distributed memory or accelerators
  - no support for automatic work or data decomposition/distribution, scheduling, etc.
  - no support for fault tolerance

# Libraries: C++ STL threads example (optimized sum)

```cpp
using DataType = unsigned long long;
using indexType = int;
constexpr DataType size = 100000;
constexpr int numThreads = 4;
std::mutex myMutex;

void partialSum(DataType& sum, int beg, int end) {
  DataType mySum{};
  for (auto i = beg; i < end; ++i) {
    mySum++;
  }
  std::lock_guard<std::mutex> lockGuard(myMutex);
  sum += mySum;
} // end of scope, lockGuard releases mutex
```

```cpp
int main() {
  DataType sum{};
  std::vector<std::thread> threads;

  for (int i = 0; i < numThreads; ++i) {
    threads.push_back(std::thread(partialSum,
      std::ref(sum), size/numThreads*i,
      size/numThreads * (i+1)));
  }

  for (auto& t : threads) {
    t.join();
  }

  std::cout << "Result: " << sum << std::endl;
}
```

# Libraries: Intel TBB

- thread-based parallelism library
  - higher-level API that offers frequently-used programming patterns `parallel_for, parallel_reduce, parallel_scan, parallel_sort`
  - concurrent data structures for vectors, queues, maps, etc.
  - advantage over OpenMP: data structures & no compiler support required

- targets shared memory only
  - no support for distributed memory or accelerators
  - supports automatic work decomposition & distribution
  - no support for fault tolerance

# Libraries: Intel TBB example (optimized sum)

```
using namespace tbb;
constexpr int size = 100000;

struct SumFunctor {
  float mySum;
  void operator()(
      const blocked_range<size_t>& r) {
    for(auto i = r.begin(); i != r.end(); ++i)
      mySum++;
  }
  SumFunctor(SumFunctor&, split) : mySum(0) {}
  void join(const SumFunctor& y) {
    mySum += y.mySum;
  }
  SumFunctor() : mySum(0) {}
};
```

```
int main() {
  SumFunctor sf;
  parallel_reduce(
    blocked_range<size_t>(0, size), sf);
  std::cout << sf.mySum << std::endl;
}
```

# Libraries: HPX

- GAS-based library for distributed memory
  - shared memory parallelism programming style
  - supports data and task parallelism
  - offers control over data distribution, scheduling, etc.

- targets shared and distributed memory systems
  - limited automatic support for distributed memory (explicit user code changes)
  - support for automatic work/data decomposition and distribution
  - no support for fault tolerance

# Libraries: HPX example (squaring distributed vector elements)

```
char const* const vec_name = "vector";
char const* const latch_name = "latch";
hpx::partitioned_vector<int> v; // create vec
hpx::lcos::latch l; // synchronization object
if(0 == hpx::get_locality_id()) { // root
  std::vector<hpx::id_type> localities =
    hpx::find_all_localities();
  v = hpx::partitioned_vector<int>(10000,
    hpx::container_layout(localities));
  v.register_as(vec_name);
  l = hpx::lcos::latch(localities.size());
  l.register_as(latch_name);
}
```

```
else { // connect to data at root
  hpx::future<void> f1 =
    v.connect_to(vec_name);
  l.connect_to(latch_name);
  f1.get();
}
partitioned_vector_view<int> view(v);
hpx::parallel::for_each( // square array data
    hpx::parallel::execution::par,
    view.begin(), view.end(),
    [](int& val) { val *= val; });
l.count_down_and_wait(); // wait
return hpx::finalize();
```

# Extension: AllScale

- GAS-based language (embedded DSL)
  - shared memory parallelism programming style
  - supports data, task and nested parallelism
  - offers control over data distribution, scheduling, etc.
- targets shared and distributed memory systems
  - limited automatic support for distributed memory (explicit user code changes)
  - support for process-level fault tolerance

# Extension: AllScale example (2D stencil)

```
const int N = 200;
const int T = 100;
using Grid = data::StaticGrid<double, N, N>;
using Point = allscale::utils::Vector<int, 2>;
Grid temp;

allscale::api::user::algorithm::stencil(temp, T,
  [k,T,N](time_t, const Point& p, const Grid& temp)->double { // inner elements
    return temp[p] + k*(temp[p+Point{-1,0}] + temp[p+Point{+1,0}] +
                        temp[p+Point{0,-1}] + temp[p+Point{0,+1}] + (-4)*temp[p]);
  },
  [k,T,N](time_t, const Point&, const Grid&)->double { // boundary cells
    return 0; // boundaries are constants
  });
```

# Languages: OpenCL

- C-like language + library for accelerator programming
  - use library to communicate with OpenCL driver on the host
  - use OpenCL-language to run code on the accelerator
    - is compiled by OpenCL compiler at execution
  - requires OpenCL toolchain (compiler, debugger, etc…)
  - similar to CUDA

- targets specifically accelerators
  - no support for distributed memory
  - no support for automatic work/data decomposition & distribution
  - no support for fault tolerance

# Languages: OpenCL example (matrix multiplication)

```
// this code runs on the host computer
clGetPlatformIDs(...); // setup environment
clGetDeviceIDs(...); // setup env
clCreateContext(...); // setup env
clCreateCommandQueue(...); // setup env
clCreateBuffer(...); // allocate gpu memory
clEnqueueWriteBuffer(...); // copy data to GPU
clCreateProgramWithSource(...); // build exe
clGetProgramBuildInfo(...); // build exe
clCreateKernel(...); // build exe
clEnqueueNDRangeKernel(...); // run kernel
clEnqueueReadBuffer(...); // get data from GPU
/* cleanup ... */
```

```
// this code runs on the GPU
__kernel void matrix_mul(
    __global float *matrix_A,
    __global float *matrix_B,
    __global float *matrix_C) {
  int i = get_global_id(0);
  int j = get_global_id(1);
  float sum = 0.0;
  for(int k = 0; k < M; ++k) {
    sum += matrix_A[N*i+k] * matrix_B[M*k+j];
  }
  matrix_C[N*i+j] = sum;
}
```

# Languages: SYCL

- C++-based DSL for accelerators with advantages over OpenCL/CUDA
  - type safety and more advantages of a C++ API
  - single source that runs on CPUs or accelerators (debugging!)
  - not vendor-specific
  - minimum amount of boilerplate code

- targets CPUs and accelerators (shared memory only)
  - no support for distributed memory (➔ CELERITY @ DPS group ☺)
  - limited support for automatic work/data decomposition and distribution
  - no support for fault tolerance

# Languages: SYCL example (vector add)

```
using namespace cl::sycl;
#define LENGTH 1024

std::vector host_a(LENGTH);
std::vector host_b(LENGTH);
std::vector host_r(LENGTH);

/* initialize a and b... */

buffer d_a(host_a); // input buffer
buffer d_b(host_b); // input buffer
buffer d_r(host_r); // output buffer
queue myQueue;
```

```
queue.submit([&](handler& cgh) {
  // Data accessors
  auto a = d_a.get_access<access::read>(cgh);
  auto b = d_b.get_access<access::read>(cgh);
  auto r = d_r.get_access<access::write>(cgh);

  // Kernel
  cgh.parallel_for<class VA>([=](id<> item) {
    int i = item.get_global(0);
    r[i] = a[i] + b[i];
  });
});
```

# Languages: Chapel

- GAS-based language developed by Cray
  - shared memory parallelism programming style
    - processes are called “*locales*”, and can have multiple “*threads*”
  - supports data, task and nested parallelism, offers distributed data structures
  - offers explicit control over data distribution, scheduling, etc.
  - advantage over OpenMP: supports distributed memory
  - advantage over MPI: no explicit message passing required

- targets shared and distributed memory systems
  - limited automatic support for distributed memory (explicit user code changes)
  - support for process-level fault tolerance

# Languages: Chapel examples (opt. sum, shared and distributed)

```
// automatically synchronized
var sum : sync int = 0;
// forall starts a thread per core
forall i in 1..1000000 {
  sum += 1; // no race condition!
}
// workaround due to a bug
var res = sum;
// print result
writeln(res);
```

```
// assign work in round-robin fashion
use CyclicDist;
// index space for distributed memory
const IndexSpace = {1..100000} dmapped
  Cyclic(startIdx=1);
var sum : sync int = 0;
// distribute iterations among localities
forall i in IndexSpace {
  sum += 1;
}
var res = sum;
writeln(res);
```

# Languages: X10

- GAS-based language developed by IBM
  - distributed memory
  - quite similar Chapel (but more object-orientation and Java-adherence, also offers garbage collection!)

- targets distributed memory incl. accelerators
  - limited automatic support for distributed memory (explicit user code changes)
  - support for process-level fault tolerance

# Languages: X10 example: optimized sum

```
import x10.array.*;

public class DistArraySum {
  static N = 10;

  static def sum(a:DistArray[Double]):Double {
    var sum:Double = 0;
    for(pt in a) sum += at(a.place(pt)) a(pt);
    return sum;
  }
  public static def main(Rail[String]) {
    val a = new DistArray_BlockBlock_2[Double](N, N, (i:Long,j:Long)=>(i+j) as Double);
    Console.OUT.println("Sum: " + sum(a));
  }
}
```

# Languages: Matlab

- domain-specific language for math applications
  - offers index-notation, syntax for matrix- and element-wise operations, etc.
  - offers built-in visualization tools

- offers various forms of parallelism
  - vectorization via specific functions implemented using the Intel MKL
  - distributed memory and accelerator parallelism via the Parallel Computing Toolbox and the Matlab Parallel Server
    - supports multi-process parallelism and GPUs via MPI and CUDA
    - offers interface to call MPI and CUDA from within Matlab code

# Languages: Matlab examples: SPMD & GPUs

```matlab
% start a pool with 3 workers
parpool(3);
% run program in parallel
spmd
  q = magic(labindex + 2);
end
% plot the data
figure
subplot(1,3,1), imagesc(q{1});
subplot(1,3,2), imagesc(q{2});
subplot(1,3,3), imagesc(q{3});
```

```matlab
% start a pool with all GPUs
parpool(gpuDeviceCount);
numSamples = 100;
% initialize data on the GPUs
X = zeros(numSamples,N,'gpuArray');
% distribute computation across GPUs
parfor i = 1:numSamples
    X(i,:) = rand(1,N,'gpuArray');
    for n=1:numIterations
        X(i,:) = r.*X(i,:).*(1-X(i,:));
    end
end
```

# Overview of state-of-the-art programming models

| | Architectural | | | Task System | | | | Management | | | | Eng. | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Communication Model | Distributed Memory | Heterogeneity | Graph Structure | Task Partitioning | Result Handling | Task Cancellation | Worker Management | Resilience Management | Work Mapping | Synchronization | Technological Readiness | Implementation Type |
| C++ STL | smem | × | × | dag | × | i/e | × | i | × | i/e | e | 9 | Library |
| TBB | smem | × | × | tree | × | i | ✓ | i | × | i | i | 8 | |
| HPX | gas | i | e | dag | ✓ | e | ✓ | i/e | × | i/e | e | 6 | |
| Legion | gas | i | e | tree | ✓ | e | × | i | × | i/e | e | 4 | |
| PaRSEC | msg | e | e | dag | × | e | ✓ | i | ✓ | i/e | i | 4 | |
| OpenMP | smem | × | i | dag | × | i | ✓ | e | × | i | i/e | 9 | Extension |
| Charm++ | gas | i | e | dag | ✓ | i/e | × | i | ✓ | i/e | e | 6 | |
| OmpSs | smem | × | i | dag | × | i | × | i | ✓ | i | i/e | 5 | |
| AllScale | gas | i | i | dag | ✓ | i/e | × | i | ✓ | i | i/e | 3 | |
| StarPU | msg | e | e | dag | ✓ | i | × | i | × | i/e | e | 5 | |
| Cilk Plus | smem | × | × | tree | × | i | × | i | × | i | e | 8 | Lang. |
| Chapel | gas | i | i | dag | ✓ | i | × | i | × | i/e | e | 5 | |
| X10 | gas | i | i | dag | ✓ | i | × | i | ✓ | i/e | e | 5 | |

# Where is the research going?

- everybody talks about increasing the “separation of concerns”
  - let computer science people master the computer science
  - let domain science people master their domain science
- everybody talks about “portability”
  - large efforts in creating DSLs that allow running on any architecture (e.g. Intel “One API”)
  - write code once, compile and run on CPU, GPU, FPGA, etc.
  - often blurry line of distinction between libraries and extensions
- everybody talks about “performance portability”
  - nobody knows how to define it accurately
  - nobody knows how to achieve it properly

# Considerations for application developers

- Which platform should I target?
  - shared memory, distributed memory, accelerators, ...
- Which features and degree of control do I need?
  - automatic work and data decomposition and distribution, scheduling, fault tolerance, ...
  - porting legacy codes might constrain you to using libraries (or possibly language extensions)
- How much support do I need?
  - consider maturity and long-term support of the language/extension/library
- Which programming language do I know?
  - check libraries or embedded DSLs of languages you already master

# Considerations for system developers

- Which features do I want to offer?
  - e.g. automatic data decomposition & distribution often requires data flow analysis and hence a compiler

- What do I require my application developers to master?
  - build upon widely-used programming languages
  - choosing a library-based model often decreases adoption barriers

- How much effort can I spend on this? How many people are in my workforce?
  - languages and their toolchains take an immense (!) amount of effort if done properly
  - only move away from library-only solutions if necessary
  - even then, consider language extensions first

# Summary

- choose your weapon wisely
  - feature set / target architecture / effort
- glimpse into the world of DSLs
  - Pthreads, C++ STL threads, TBB, HPX, AllScale, OpenCL/CUDA, SYCL, Chapel, X10, Matlab

# Image sources

- Standards: https://xkcd.com/927/
- State-of-the-Art Overview: https://link.springer.com/content/pdf/10.1007%2Fs11227-018-2238-4.pdf