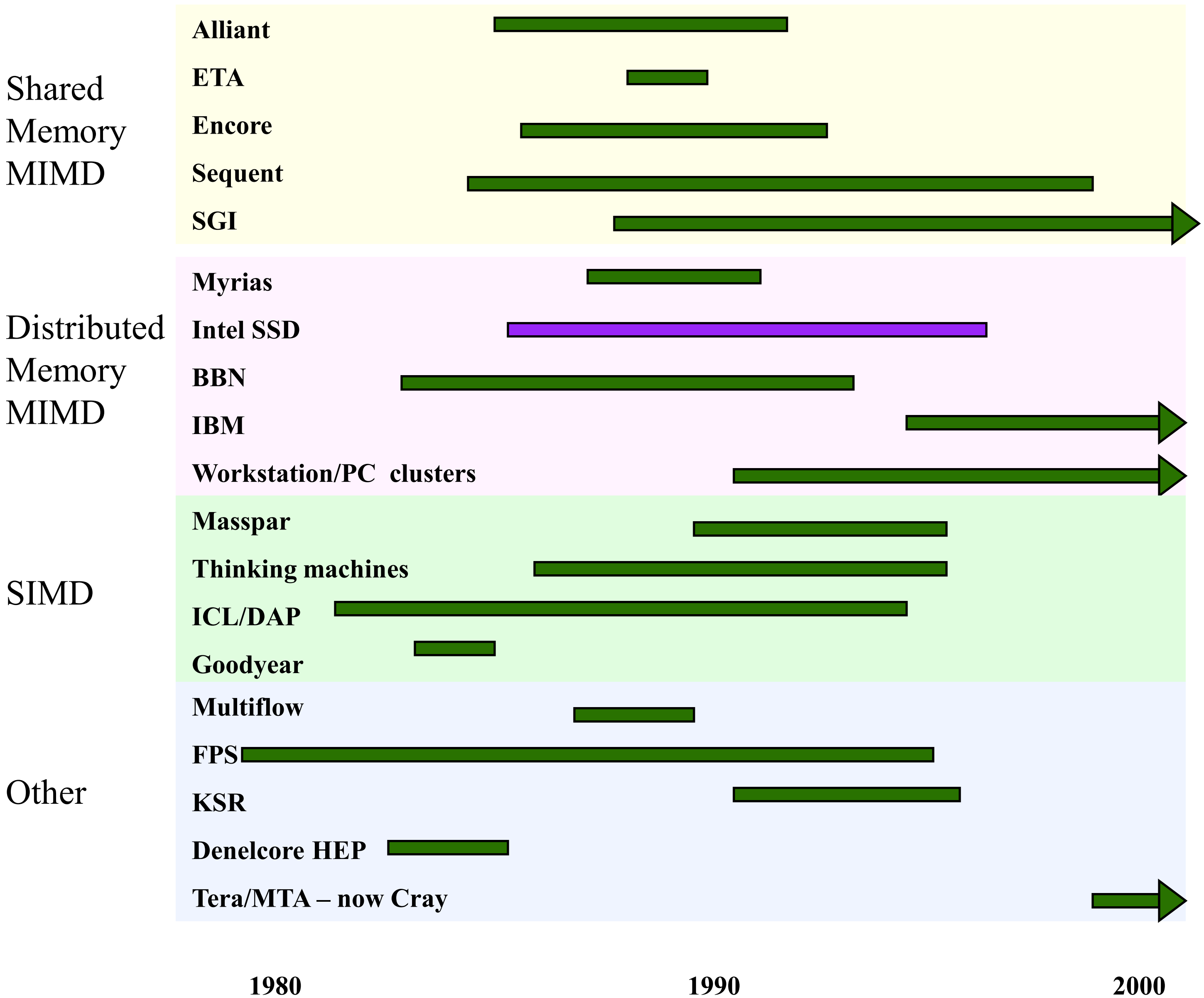
Recent developments in parallel programming: the good, the bad, and the ugly

Tim Mattson, Kayak bum and Intel Labs researcher

Disclaimer

- The views expressed in this talk are those of the speaker and not his employer.
- I work in a research lab and know very little about Intel Products that you couldn’t learn online.

the “Dead Architecture Society”



What went wrong? Automatic parallelism will never work in real applications … so you have to write parallel code … and Programming these systems were akin to herding cats

Source: EDS Super bowl 2005 commercial

Only a small number of super computing aficionados took up the challenge of programming these systems

Application software is all that matters!

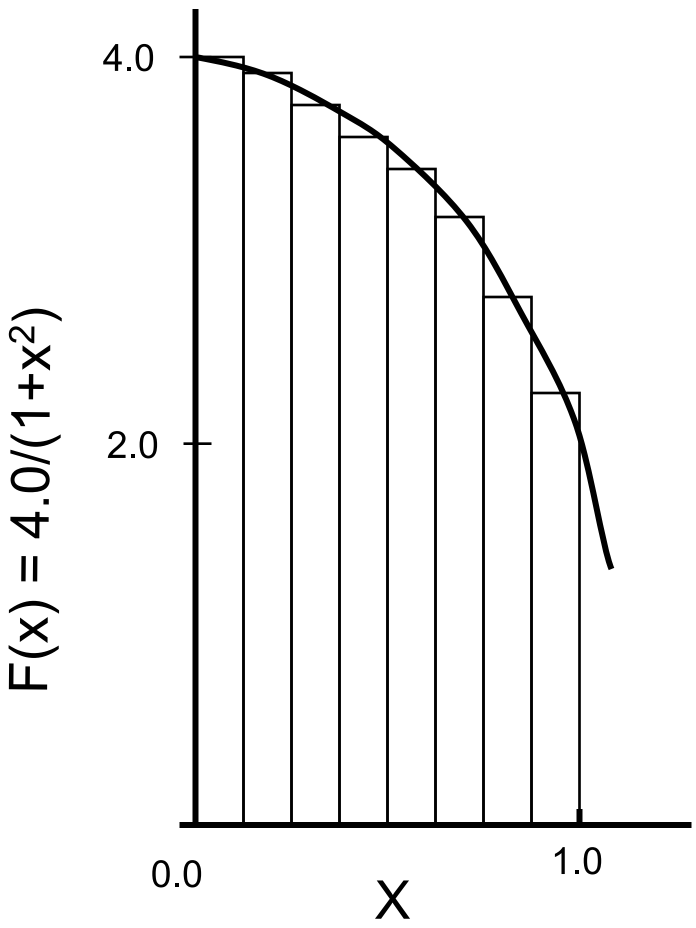
- If we don't want to add many-core chips to the dead architecture society, we had better take the needs of our applications programmers VERY seriously

… so let's take a look at some of the important recent trends in parallel programming.

The Good the Bad and the Ugly

- Threading like its 2011
- Next generation heterogeneous programming
- Parallel languages/tools will never get it right. I give up.

Example Problem: Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

PI Program: an example

```
#define NUMSTEPS = 100000;
double step;
void main ()
{           int i;    double x, pi, sum = 0.0;

            step = 1.0/(double) NUMSTEPS;
            x = 0.5 * step;
            for (i=0;i<= NUMSTEPS; i++){
                      x+=step;
                      sum += 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

PI Program: an example

Let's turn this into a parallel program using the Pthreads API.

```
#define NUMSTEPS = 100000;
double step;
void main ()
{         int i;    double x, pi, sum = 0.0;

          step = 1.0/(double) NUMSTEPS;
          x = 0.5 * step;
          for (i=0;i<= NUMSTEPS; i++){
                   x+=step;
                   sum += 4.0/(1.0+x*x);
          }
          pi = step * sum;
}
```

Variable to accumulate thread results must be shared

Assign loop iterations to threads

Package this into a function

Assure safe update to sum … correct for any thread schedule

Numerical Integration: PThreads (1 of 2)

```
#include <stdio.h>
#include <pthread.h>
#define NUMSTEPS 10000000
#define NUMTHREADS 4
double gStep = 0.0, gPi = 0.0;        pthread_mutex_t gLock;
void *Func(void *pArg)
{
   int myRank = *((int *)pArg);
   double partialSum = 0.0, x;
   for (int i = myRank; i < NUMSTEPS; i += NUMTHREADS)
   {
      x = (i + 0.5f) * gStep;
      partialSum += 4.0f / (1.0f + x*x);
   }
   pthread_mutex_lock(&gLock);
     gPi += partialSum * gStep;
   pthread_mutex_unlock(&gLock);

   return 0;
}
```

Global variables … on the heap

Cyclic loop distribution … deal out loop iterations as you would a deck of cards

Put any code you want inbetweeen the Mutex_lock and unlock. This is called a **Critical section** … only one thread at a time can execute this code

Source: Michael Wrinn of Intel

Numerical Integration: PThreads (2 of 2)

```
int main()
{
  pthread_t thrds[NUMTHREADS];
  int tNum[NUMTHREADS], i;
  pthread_mutex_init(&gLock, NULL);
  gStep = 1.0 / NUMSTEPS;
  for ( i = 0; i < NUMTHREADS; ++i )
  {
    tRank[i] = i;
    pthread_create(&thrds[i], NULL,Func,(void)&tRank[i]);
  }
  for ( i = 0; i < NUMTHREADS; ++i )
  {
    pthread_join(thrds[i], NULL);
  }
  pthread_mutex_destroy(&gLock);
  printf("Computed value of Pi: %12.9f\n", gPi );
  return 0;
}
```

Source: Michael Wrinn of Intel

Windows API (Win32): Same algorithm, different API

```
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
    int i, start;
   double x, sum = 0.0;


   start = *(int *) arg;
   step = 1.0/(double) num_steps;

   for (i=start;i<= num_steps;i=i+NUM_THREADS){
        x = (i-0.5)*step;
       sum = sum + 4.0/(1.0+x*x);
   }
   EnterCriticalSection(&hUpdateMutex);
       global_sum += sum;
   LeaveCriticalSection(&hUpdateMutex);
}
```

```
void main ()
{
   double pi; int i;
   DWORD threadID;
   int threadArg[NUM_THREADS];

   for(i=0; i<NUM_THREADS; i++)   threadArg[i] = i+1;

   InitializeCriticalSection(&hUpdateMutex);

   for (i=0; i<NUM_THREADS; i++){
        thread_handles[i] = CreateThread(0, 0,
             (LPTHREAD_START_ROUTINE) Pi,
             &threadArg[i], 0, &threadID);
}

   WaitForMultipleObjects(NUM_THREADS,
             thread_handles, TRUE,INFINITE);

   pi = global_sum * step;

   printf(" pi is %f \n",pi);
}
```

C++'11provides a portable (and cleaner) way to write my "pi program"

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
std::mutex m;
static long nsteps = 100000000;
double step;
double pi=0.0;
void pi_func(int id, int nthrds)
{
  double x, sum=0.0;
  double step =1.0/(double) nsteps;
  for (int i=id;i<=nsteps; i+=nthrs){
     x = (i-0.5)*step;
     sum = sum + 4.0/(1.0+x*x);
  }
  m.lock();
     pi += step * sum;
  m.unlock();
}
```

```
int main ()
{
  int i;
  unsigned long hwthrds =
        std::thread::hardware_concurrency();

  std::vector<std::thread>thrds(hwthrds-1);

  for(int i=0; i<hw_thrds-1;i++)
    thrds[i]=std::thread(pi_func,i,hwthrds);
  pi_func(hw_thrds-1,hw_thrds);

  for(int i=0; i<hw_thrds-1;i++)
        thrds[i].join();

  std::cout << "\n pi =" << pi <<"\n";
}
```

History of C++

1979	BjarneStroustrup developed “C with classes” inspired by his work with the early OOP language Simula
Early 80’s	CFront is the first tool to generate C from “C with Classes”
1983	C++ is born based on C with Classes
1985	Bjarne Stroustrup publishes “the C++ programming language”
1998	First formal standard from ISO, C++98
2003	Second ISO C++ standard .. Patched up issues in C++98
Late 2011	The current ISO C++ standard released …. C++11. Many changes including multi-threading support!!!
2013	CPLEX group formed to explore high level parallel constructs in future ISO C++ standards

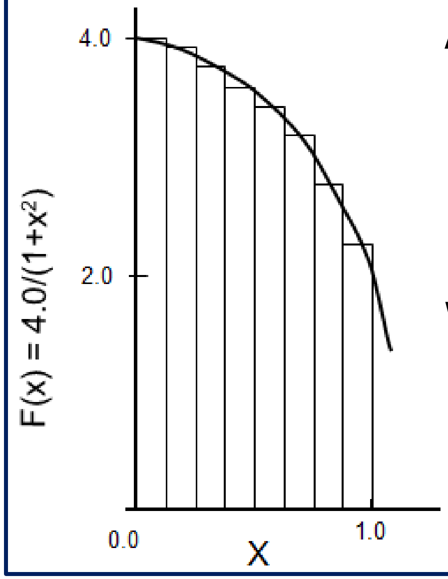
C++'11 and multithreaded programming

```
#include <iostream>

static long num_steps = 100000000;
double step, pi=0.0;
int main ()
{
        double x, sum=0.0, double step = 1.0/(double) num_steps;
        for (int i=1;i<= num_steps; i++){
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }

        pi += step * sum;

   std::cout << "\n pi with " << num_steps << " is " << pi <<"\n";
}
```



Approximate this integral

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

With this summation

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

C++'11 and multithreaded programming

Step 1 of 3

```
#include <iostream>
#include <thread>
#include <vector>


static long num_steps = 100000000;
double step, pi=0.0;
int main ()
{  unsigned long nthrds = std::thread::hardware_concurrency();
   std::vector<std::thread>threads(nthrds-1);


        double x, sum=0.0, double step = 1.0/(double) num_steps;
        for (int i=1;i<= num_steps; i++){
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }

        pi += step * sum;


   std::cout << "\n pi with " << num_steps << " is " << pi <<"\n";
}
```

Fetch how many concurrent threads the hardware can support

Setup a vector of thread objects

C++’11 and multithreaded programming

Step 2 of 3

```
#include <iostream>
#include <thread>
#include <vector>


static long num_steps = 100000000;
double step, pi=0.0;
int main ()
{  unsigned long nthrds = std::thread::hardware_concurrency();
   std::vector<std::thread>threads(nthrds-1);
   for(int id=0; id<nthrds;id++) {
       threads[id]=std::thread([id,nthrds]{
          double x, sum=0.0, double step = 1.0/(double) num_steps;
          for (int i=1;i<= num_steps; i++){
             x = (i-0.5)*step;
             sum = sum + 4.0/(1.0+x*x);
          }

          pi += step * sum;
       });
   }
  for(int id=0; id<nthrds;id++) threads[id].join();
  std::cout << "\n pi with " << num_steps << " is " << pi <<"\n";
}
```

Call constructor for each thread with “pi loop” packaged into a lambda expression with capture (copy) of id and nthrds.

Wait for each thread to finish

C++'11 and multithreaded programming

Step 3 of 3

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
std::mutex m;
static long num_steps = 100000000;
double step, pi=0.0;
int main ()
{  unsigned long nthrds = std::thread::hardware_concurrency();
   std::vector<std::thread>threads(nthrds-1);
   for(int id=0; id<nthrds;id++) {
       threads[id]=std::thread([id,nthrds]{
          double x, sum=0.0, double step = 1.0/(double) num_steps;
          for (int i=id;i<= num_steps; i+=nthrds){
              x = (i-0.5)*step;
              sum = sum + 4.0/(1.0+x*x);
          }
         std::lock_guard<std::mutex> guard(m);
          pi += step * sum;
       });
   }
   for(int id=0; id<nthrds;id++) threads[id].join();
   std::cout << "\n pi with " << num_steps << " is " << pi <<"\n";
}
```

Declare a mutex to support safe accumulation of each threads partial sum

Cyclic distribution of loop iterations

Protect update of our accumulator with a mutex (release in thread destructor)

C++'11 and multithreaded programming

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
std::mutex m;
static long num_steps = 100000000;
double step, pi=0.0;
int main ()
{  unsigned long nthrds = std::thread::hardware_concurrency();
   std::vector<std::thread>threads(nthrds-1);
   for(int id=0; id<nthrds;id++) {
      threads[id]=std::thread([id,nthrds]{
         double x, sum=0.0, double step = 1.0/(double) num_steps;
         for (int i=id;i<= num_steps; i+=nthrds){
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
         }
         m.lock();   pi += step * sum;   m.unlock();

      });
   }
   for(int id=0; id<nthrds;id++) threads[id].join();
   std::cout << "\n pi with " << num_steps << " is " << pi <<"\n";
}
```

Alternate use of the mutex … might perform better?

Parallelism in C++'11

- The core constructs expected on a shared memory machine were added:
  - Threads
  - Synchronization
  - Futures and promises
  - Async tasks
- Plus features to make things easier for programmers
  - Lambda functions
  - Auto
- And they did the responsible thing … they defined a memory model.
  - Constrains consistency of memory operations between threads to define the semantics of shared variables. Defines the set of values that can be returned from a read

The C++'11 Memory model

- For most programmers …
    - If your program is free of data races
        - i.e. loads and stores to the same location form different threads don't conflict.
    - If you use the default mode on synchronization constructs
  - Then your program will appear to be sequentially consistent … that is:
    - Each thread sees loads and stores in "**program order**"
    - All threads see Loads and stores in a single "**total order**" defined as a semantically allowed interleaving of ops from each thread.

But there will be pain …

- Original Code

x = y = 0	
Thread 1	Thread 2
r1 = X;	r3 = y;
r2 = X;	x = r3;
If (r1==r2) y = 1;	

- Sequential consistency allows results:

r1	r2	r3
0	0	1
0	0	0

- Redundant read elimination means r1 always equals r2 so y always equals 1 and can be moved ahead of load(x)

Thread 1	Thread 2
y = 1;	r3 = y;
r1 = x;	x = r3;
r2 = r1;	
if(true);	

r1	r2	r3
0	0	1
1	1	1

From S. Adve and H. Boehm, Comm ACM vol. 53, No. 8, pp. 90-101

And even more pain… (1 of 3)

```
#include <iostream>
#include <omp.h>
int val1 = 0; flag= 0
#pragma omp parallel sections num_threads(2) shared (val1, flag)
{
   #pragma omp section
   {    val1 = 1;
        #pragma omp flush
         flag = 1;
         #pragma omp flush
   }
   #pragma omp section
    {
        #pragma omp flush
         if (flag == 1)
           printf("if this prints, it can only print 1, %d",val1);
   }
}
```

We've been teaching people to write code like this for years.

According to the rules for flush in OpenMP 2.5 and earlier, it is correct.

And it's worked every where I"ve tested it

And even more pain… (2 of 3)

```
#include <iostream>
int val1 = 0; flag= 0
void func1()  {
        val1 = 1;
        std::atomic_thread_fence();
         flag = 1;
        std::atomic_thread_fence();
}
void func2()   {
          std::atomic_thread_fence();
          if (flag==1)
                std::cout << "val 1 = " << val1 <<  "better equal 1 \n";
}
Int main() {
    std::thread t1 (func1);
    std::thread t2 (func2);
    t1.join();    t2.join();
}
```

Let's do this with C++'11

This won't work. Why?

Because fences only order atomic operations. Normal loads and stores can move around them!

And even more pain… (3 of 3)

```
#include <iostream>
int val1 = 0; std::atomic<int>flag=0;
void func1()  {
        val1 = 1;
        ~~std::atomic_thread_fence();~~
        flag.store(1,std::memory_order_relaxed);
        std::atomic_thread_fence();
}
void func2()   {
         std::atomic_thread_fence();
         if (flag.load(std::memory_order_relaxed)==1))
              std::cout << “val 1 = “ << val1 <<  “better equal 1 \n”;
}
Int main() {
    std::thread t1 (func1);
    std::thread t2 (func2);
    t1.join();    t2.join();
}
```

Make flag an atomic and this works

Experienced multithreaded programmers will find this surprising and obnoxious.

And even more pain

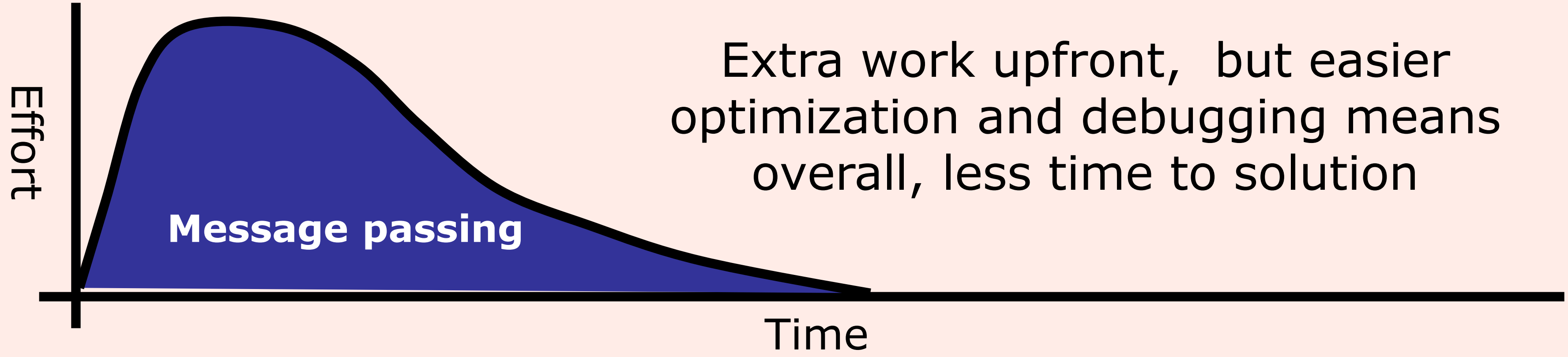
- Relaxed consistency is supported with atomics and fences with the following memory orders:
  - Relaxed
  - Acquire
  - Consume
  - Release
  - Acquire and release
  - Sequentially consistent

```
std::atomic<bool> x,y; x = y = false;
void spin_lock_release(){
      x.store(true,std::memory_order_release);
}
void spin_lock_wait(){
      whiile(!y.load(std::memory_order_acquire));
}
```

- Using these correctly is painfully difficult and well beyond the abilities of most (all?) of us.
- Do we really know what we are doing by foisting such things onto the world's programmers?

> Remember the famous warning attributed to A. Einstein
>
> "If you can't explain it to a six year old, you don't understand it yourself."

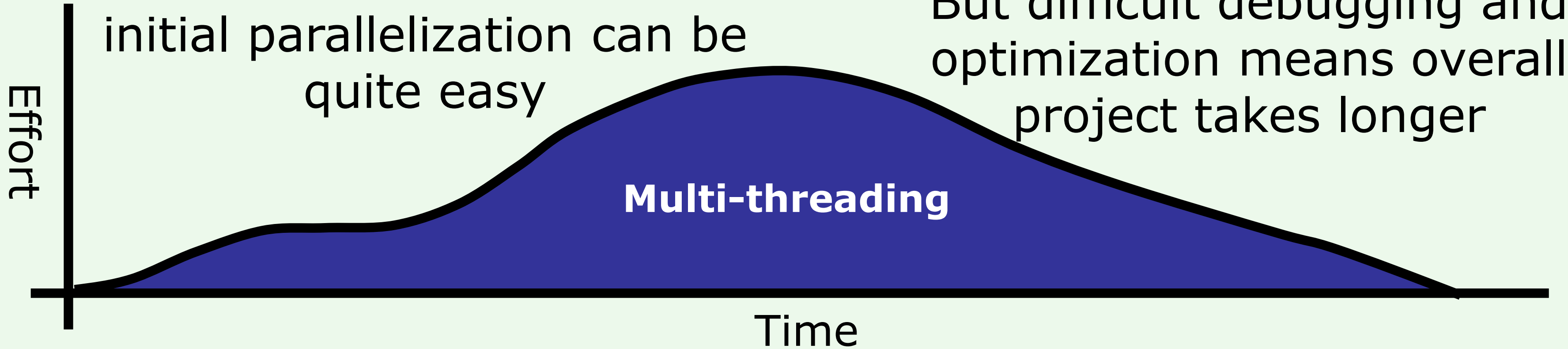
We should just abandon threading



Extra work upfront, but easier optimization and debugging means overall, less time to solution

initial parallelization can be quite easy

But difficult debugging and optimization means overall project takes longer



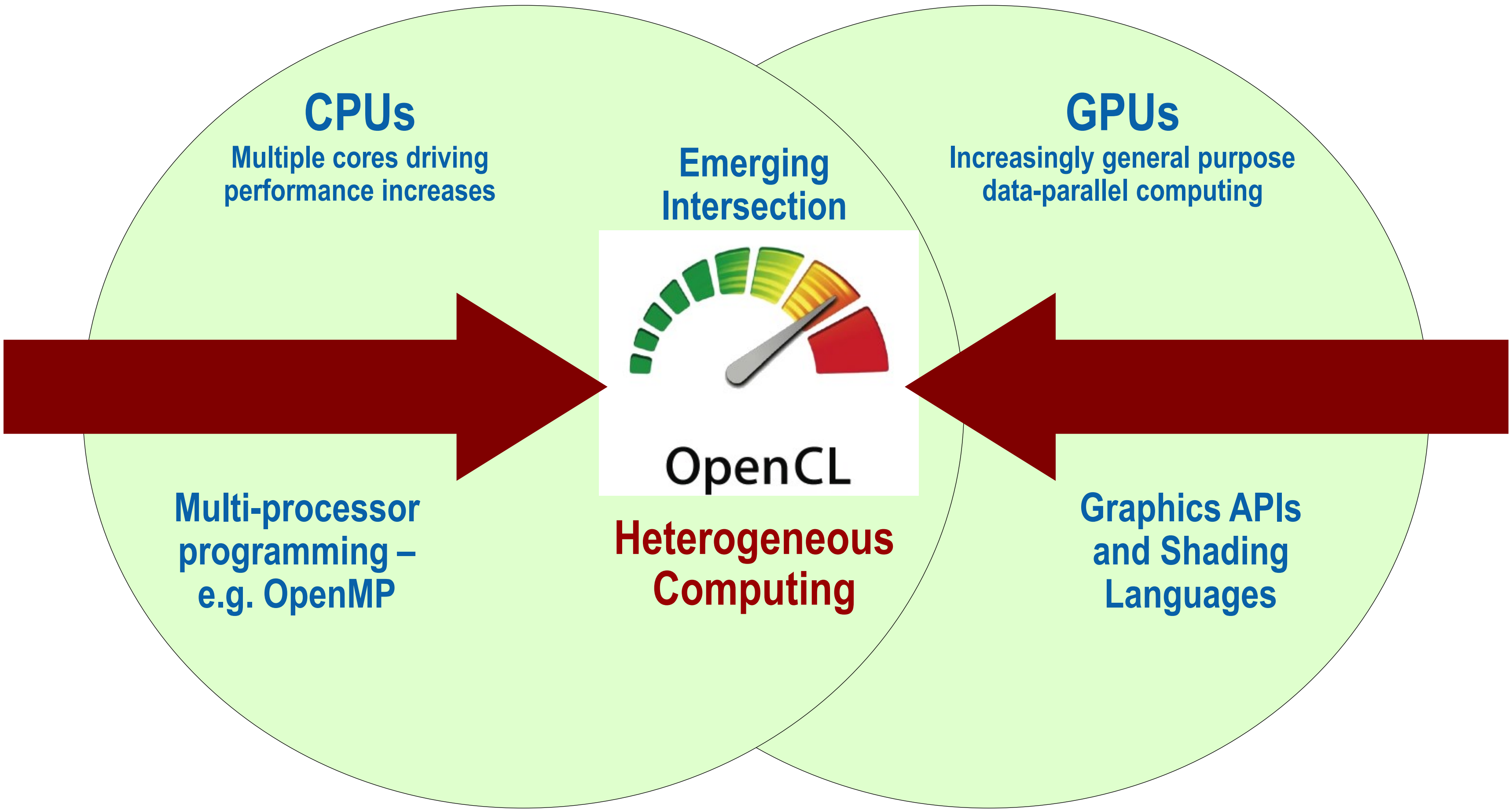
Proving that a shared address space program using semaphores is race free is an NP-complete problem*

*P. N. Klein, H. Lu, and R. H. B. Netzer, Detecting Race Conditions in Parallel Programs that Use Semaphores, Algorithmica, vol. 35 pp. 321–345, 2003

The Good the Bad and the Ugly

- Threading like its 2011
- Next generation heterogeneous programming
- Parallel languages/tools will never get it right. I give up

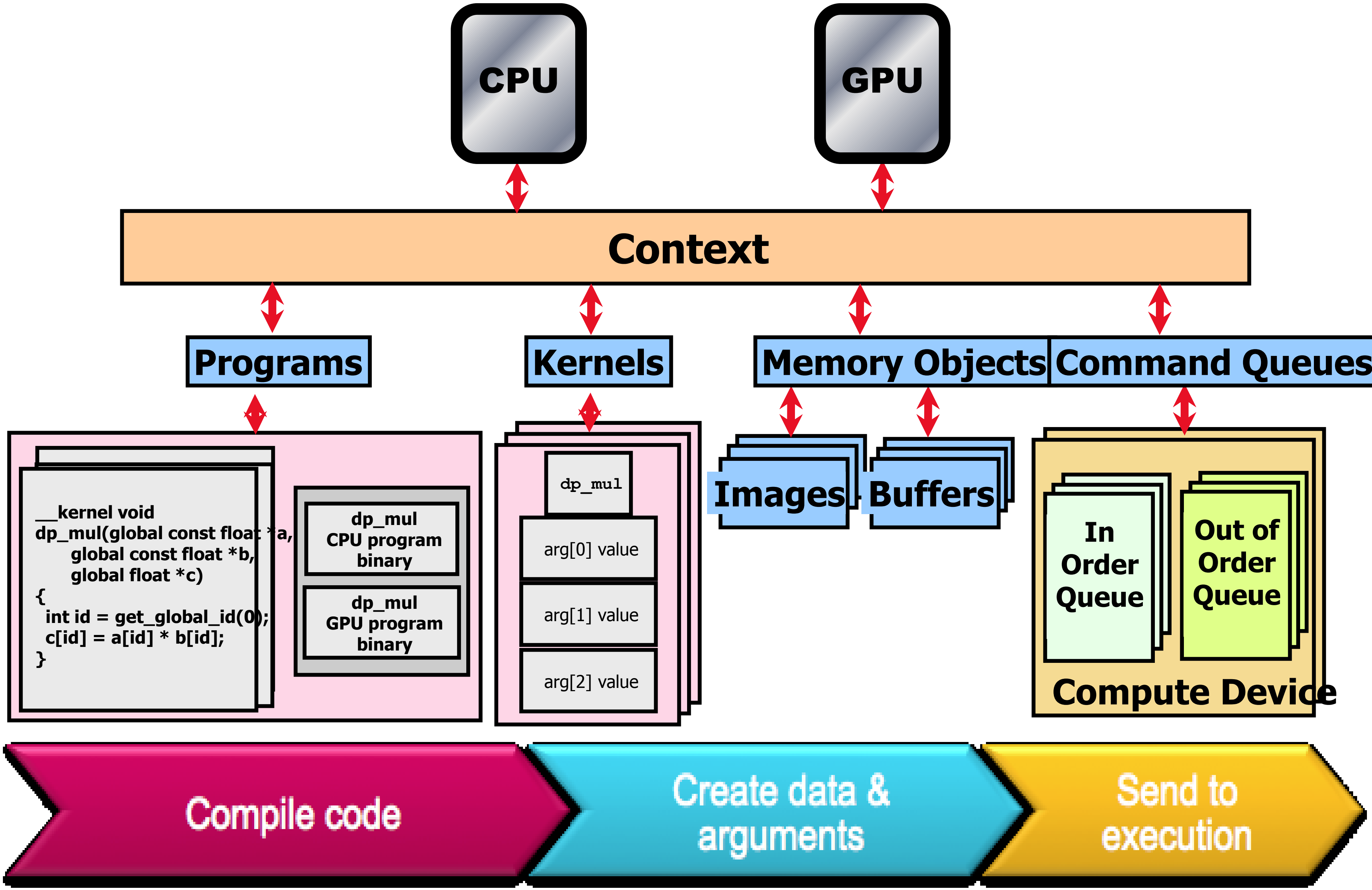
Industry Standards for Programming Heterogeneous Platforms



OpenCL – Open Computing Language

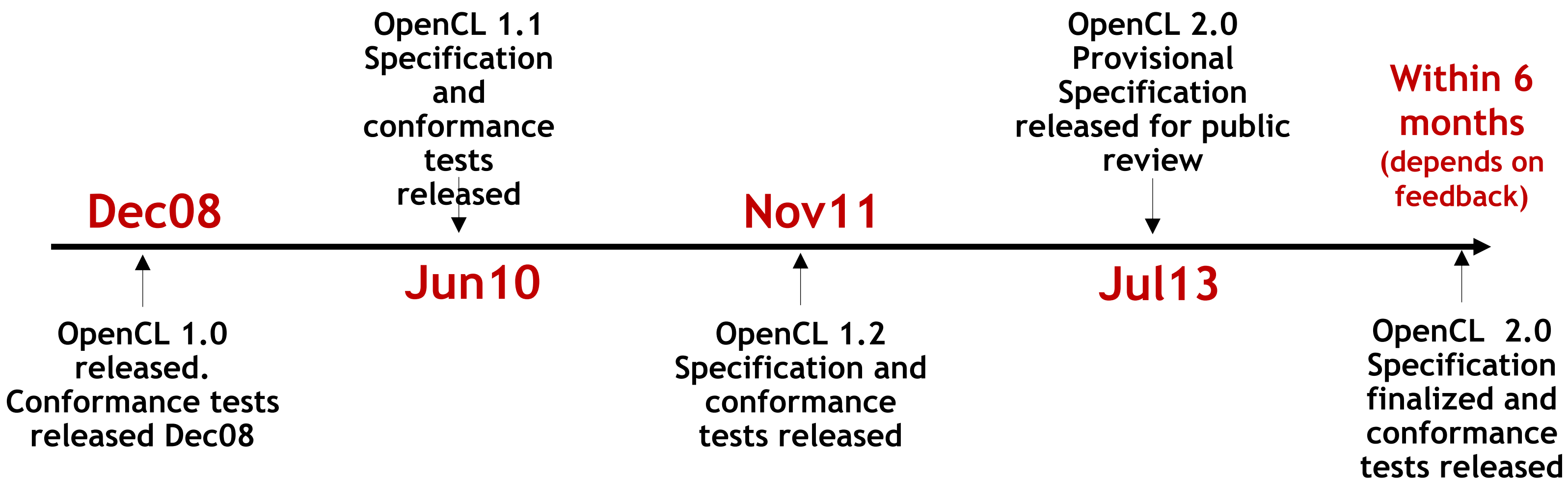
Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

OpenCL Summary



OpenCL Milestones

- Multiple conformant 1.X implementations shipping on desktop and mobile
  - For CPUs and GPUs on multiple OS



OpenCL as Parallel Compute Foundation

C++ AMP
Shevlin Park
Uses Clang and LLVM

OpenCL HLM
C++ syntax/compiler extensions

WebCL
JavaScript binding to OpenCL for initiation of OpenCL C kernels

Aparapi
Java language extensions for parallelism

River Trail
Language extensions to JavaScript

PyOpenCL
Python wrapper around OpenCL

Harlan
High level language for GPU programming

OpenCL

OpenCL provides vendor optimized, cross-platform, cross-vendor access to heterogeneous compute resources

OpenCL: Major developments in 2013

OpenCL 2.0 public review draft

OpenCL 2.0
Significant enhancements to memory and execution models to expose emerging hardware capabilities and provide increased flexibility, functionality and performance to developers

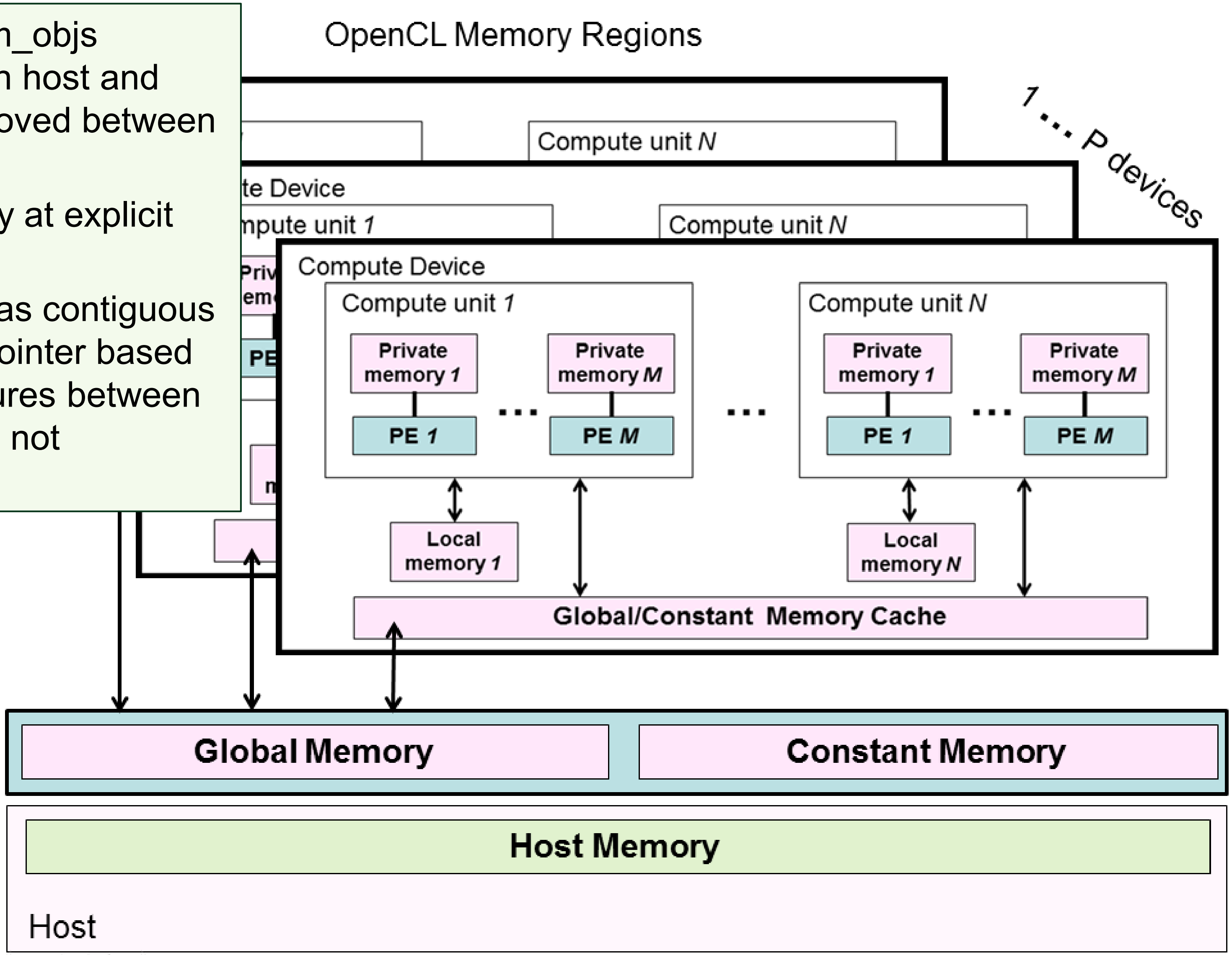
OpenCL

OpenCL SPIR 1.2 public review draft

OpenCL-SPIR (Standard Parallel Intermediate Representation)
LLVM-based, low-level Intermediate Representation as target back-end for alternative high-level languages. Provides enhanced IP protection for software vendors.

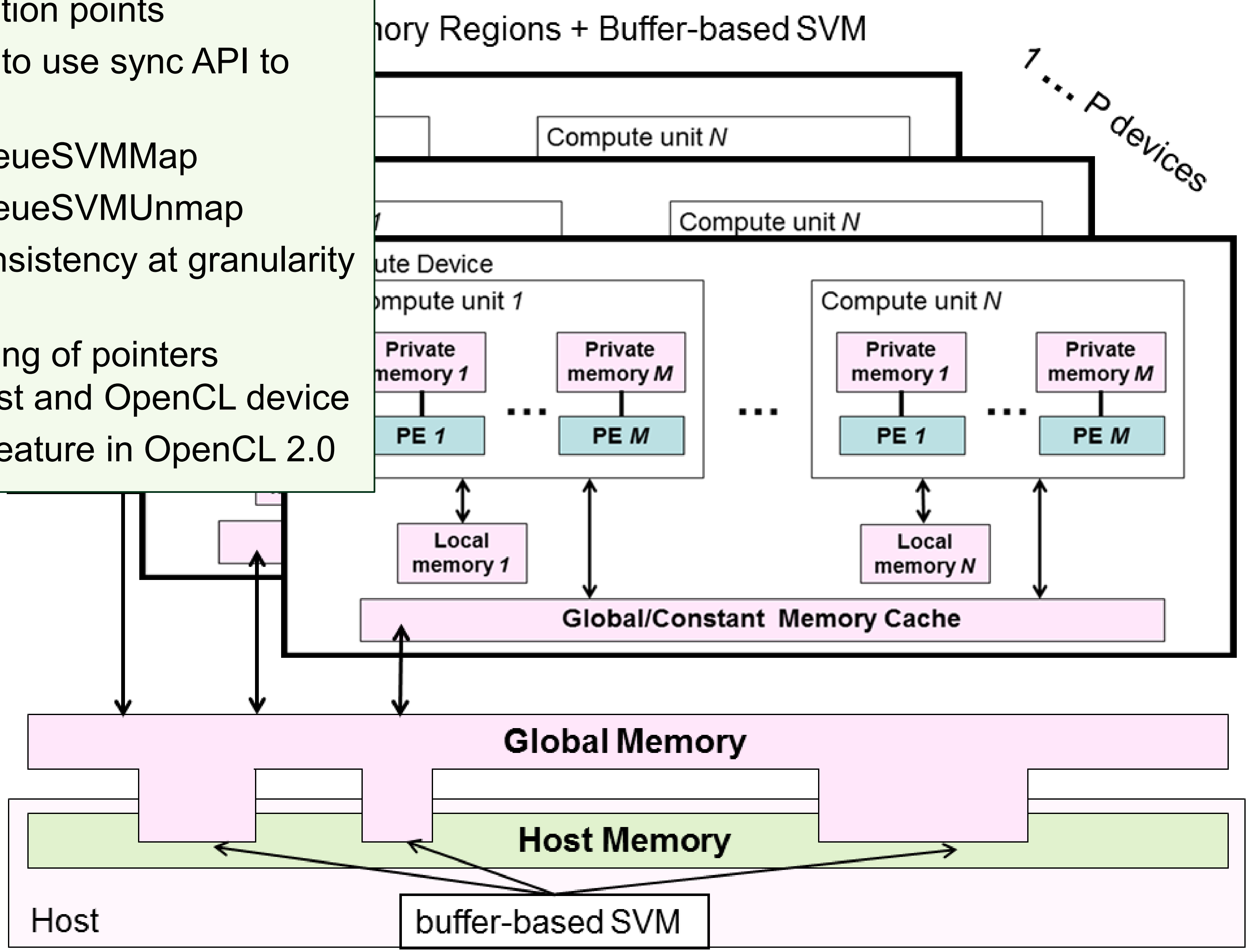
OpenCL 1.X memory Regions

- Global Mem_objs allocated on host and explicitly moved between regions.
- Consistency at explicit sync points
- Mem_objs as contiguous blocks … pointer based data structures between host/device not supported.



OpenCL 2.0: coarse grained SVM

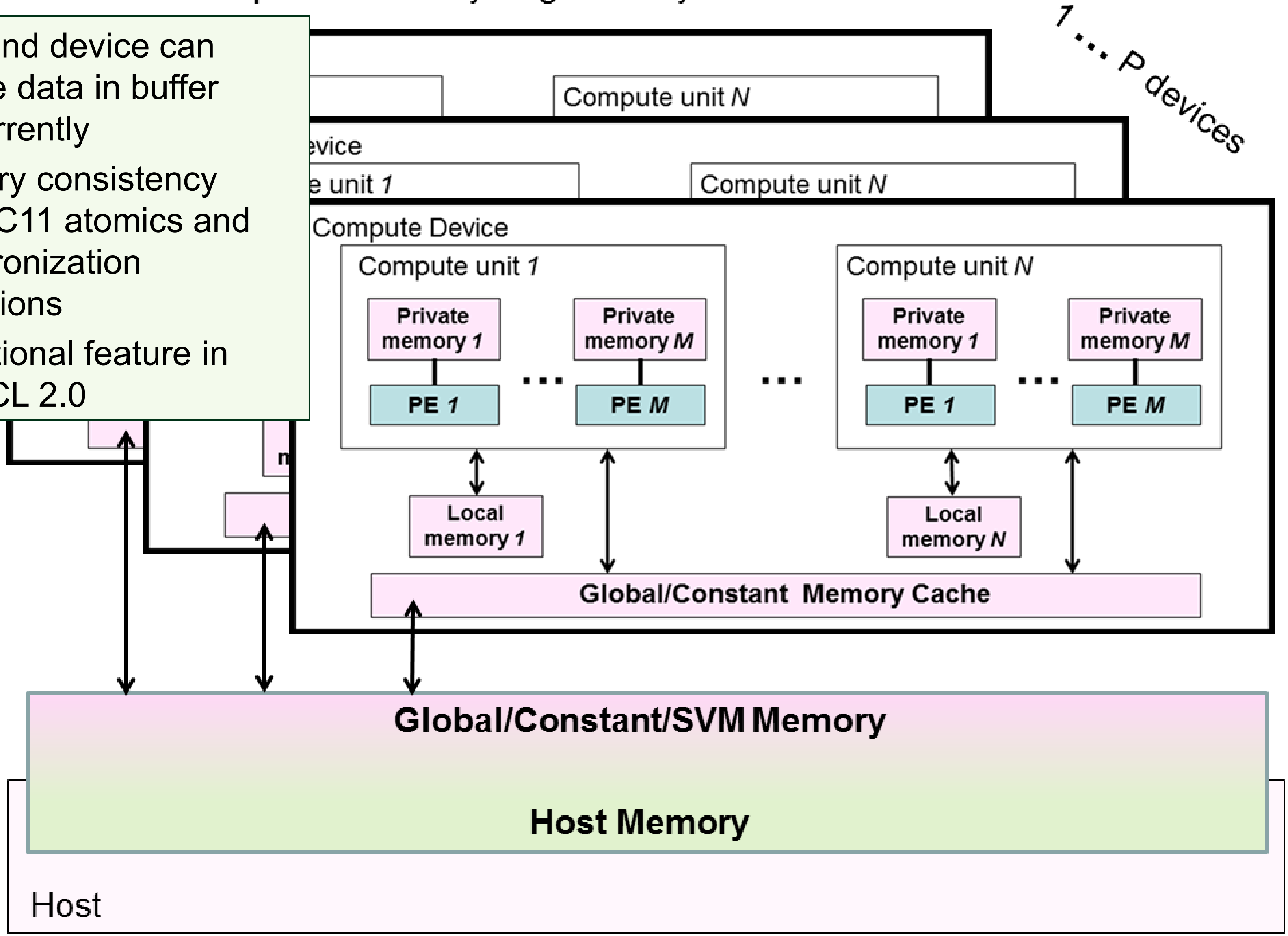
- Memory consistency at synchronization points
- Host needs to use sync API to update data
  - clEnqueueSVMMap
  - clEnqueueSVMUnmap
- Memory consistency at granularity of a buffer
- Allows sharing of pointers between host and OpenCL device
- A required feature in OpenCL 2.0



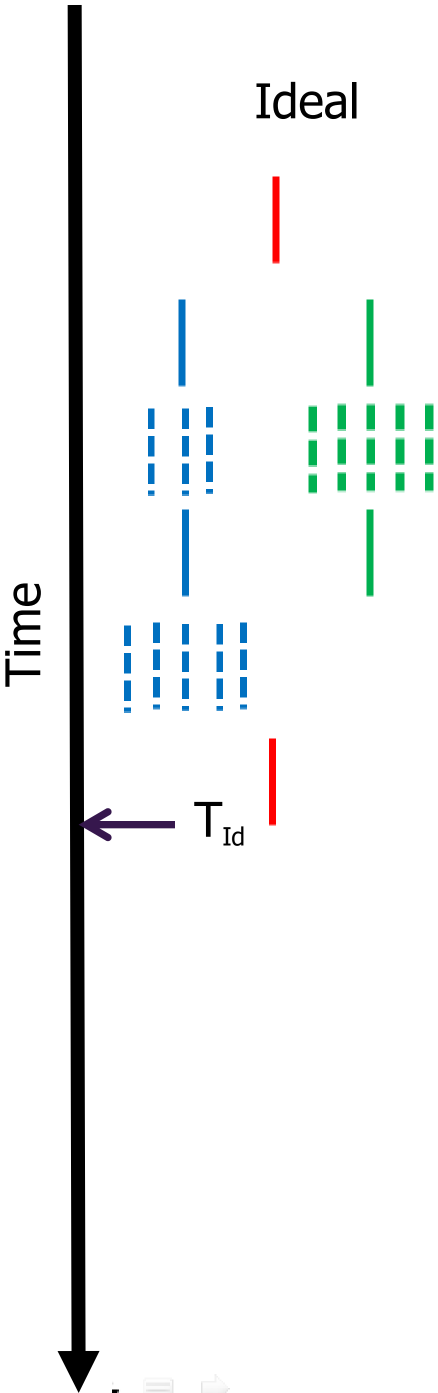
OpenCL 2.0: fine grained/System SVM

- Host and device can update data in buffer concurrently
- Memory consistency using C11 atomics and synchronization operations
- An optional feature in OpenCL 2.0

OpenCL Memory Regions + system SVM

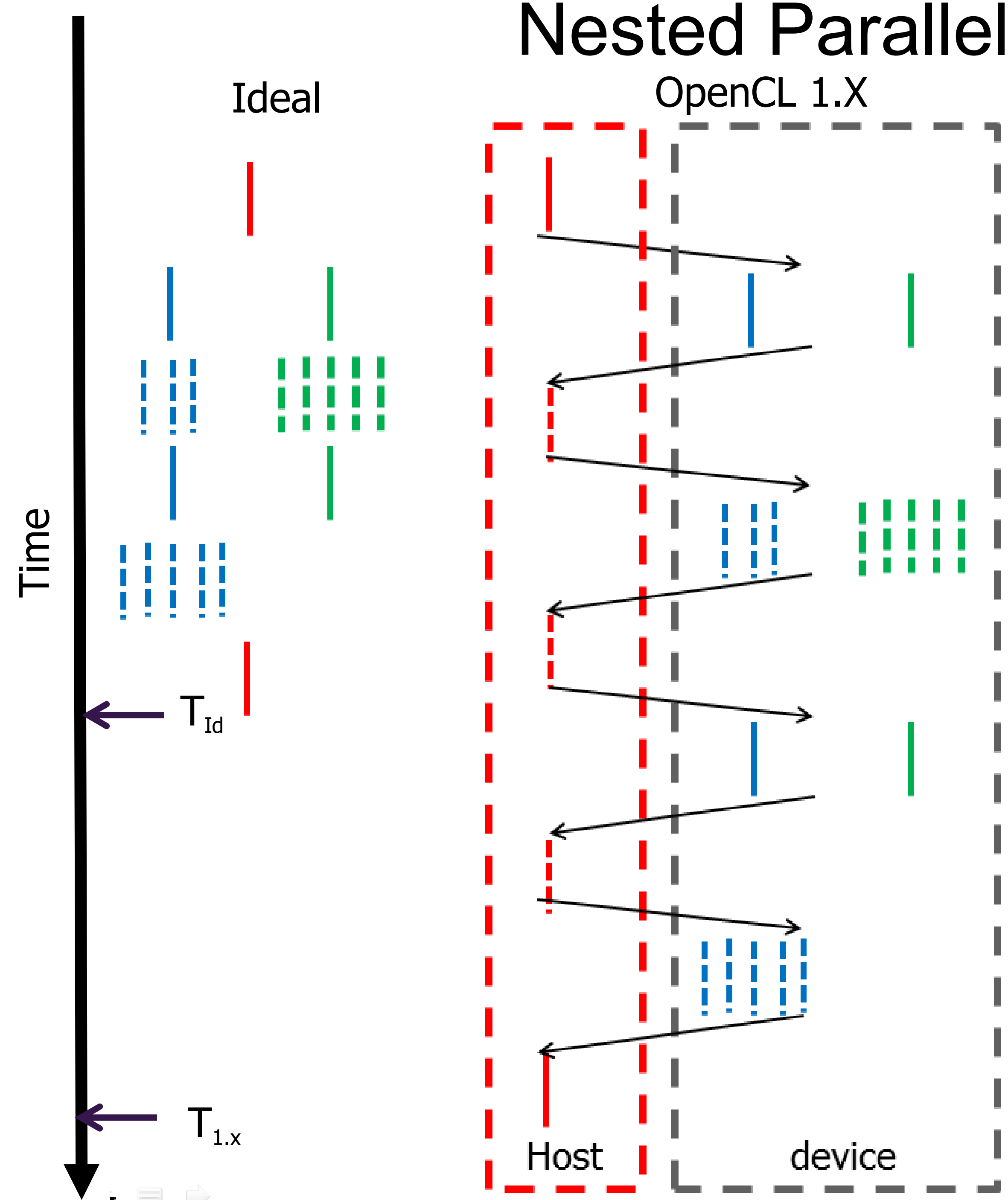


Nested Parallelism



Consider an algorithm as a task graph where the task structure is determined at runtime based on the input data.

Nested Parallelism

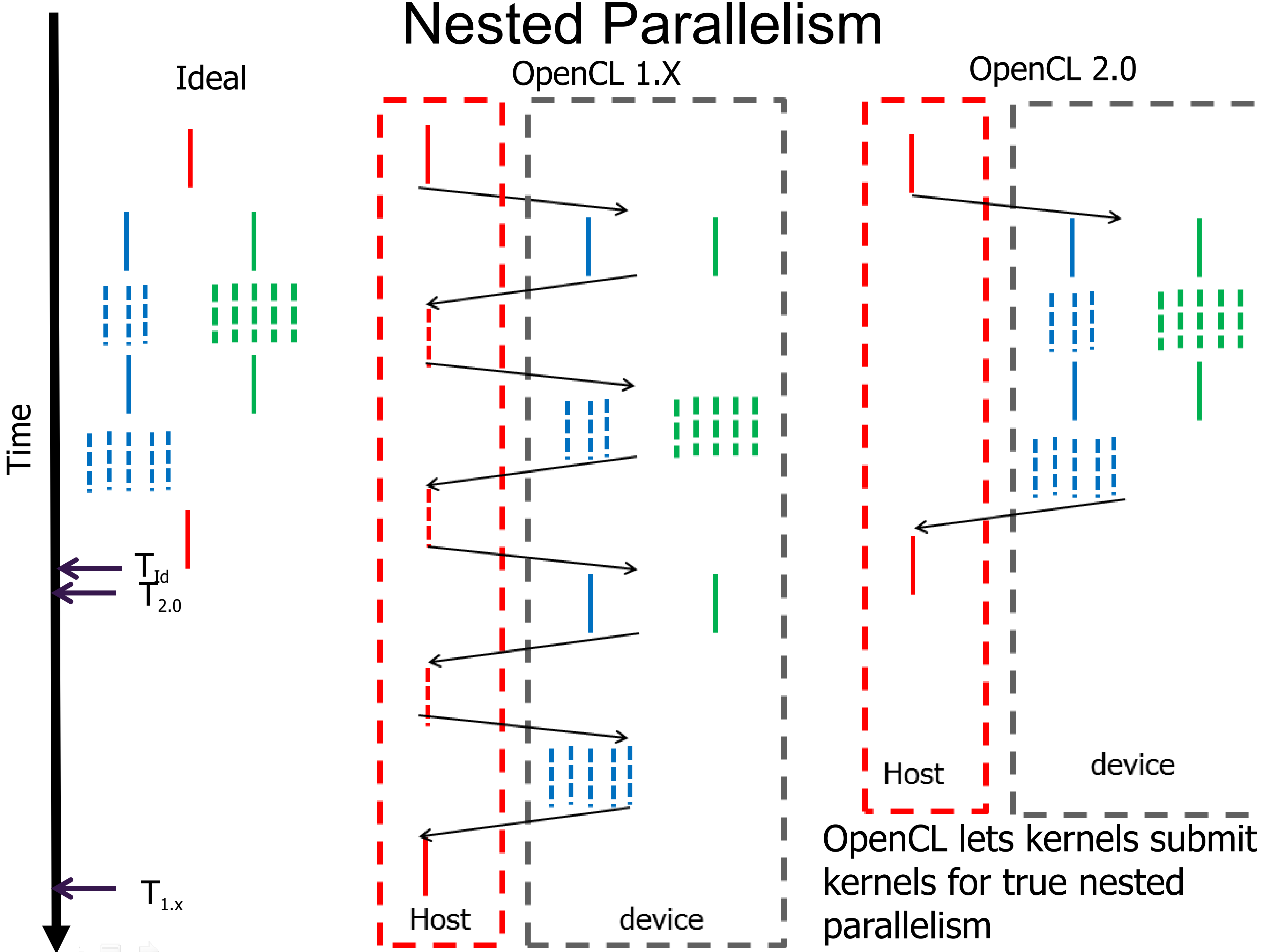


With OpenCL 1.X only the host can submit kernels for execution.

So after each task ends, it must copy data back to the host so the host knows which kernels to submit in the next phase.

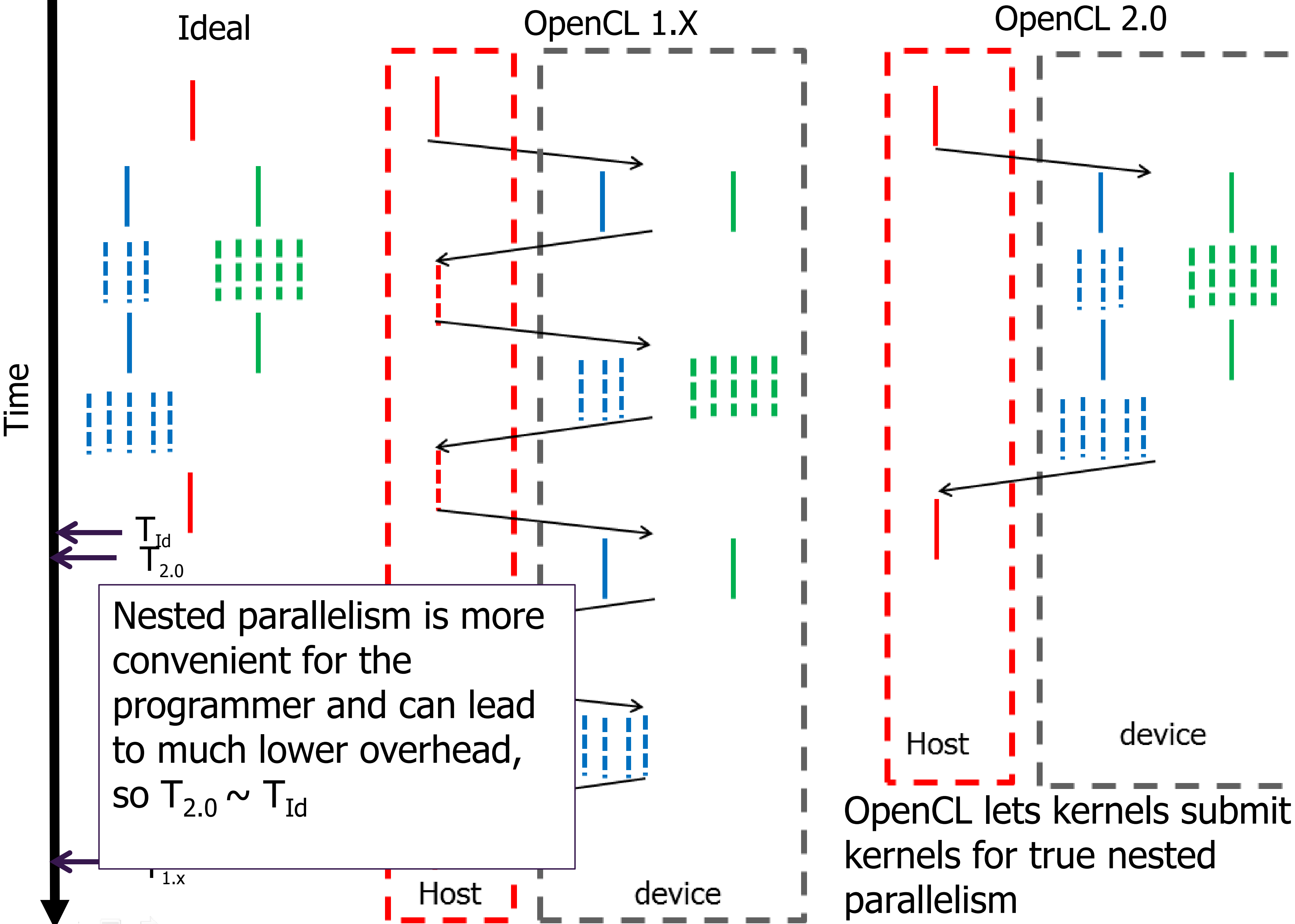
This requires extra code (the red dotted lines) and overhead resulting in $T_{1.x} >> T_{Id}$

Nested Parallelism



OpenCL lets kernels submit kernels for true nested parallelism

Nested Parallelism



Nested parallelism is more convenient for the programmer and can lead to much lower overhead, so $T_{2.0} \sim T_{Id}$

OpenCL lets kernels submit kernels for true nested parallelism

Nested Parallelism

- Use clang Blocks to describe kernel to queue

```
kernel void my_func(global int *a, global int *b)
{
    …
    void (^my_block_A)(void) =
             ^{
                    size_t id = get_global_id(0);
                    b[id] += a[id];
              };

    enqueue_kernel(get_default_queue(),
                      CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                        ndrange_1D(…),
                        my_block_A);
}
```

Generic Address Space

- OpenCL 2.0 no longer requires an address space qualifier for arguments to a function that are a pointer to a type
  - Except for kernel functions
- Generic address space assumed if no address space is specified
- Makes it really easy to write functions without having to worry about which address space arguments point to

```
void
my_func (int *ptr, …)
{
      …
      foo(ptr, …);
      …
}

kernel void
foo(global int *g_ptr,
      local int *l_ptr, …)
{
      …
      my_func(g_ptr, …);
      my_func(l_ptr, …);
}
```

Other OpenCL 2.0 Features

- What made it in
  - Memory model based on C'11 … includes atomics, and memory orders
  - Pipe memory objects to support pipeline algorithms.
  - Flexible work-group sizes
  - Expanded set of work-group functions (collective operations across work-items in a single work-group).
    - `broadcast, reduction, vote (any & all), prefix sum`
  - … and much more
- But we still lack …
  - Support for a C++ kernel programming language.
  - Ability to write a wide range of algorithms that require concurrency guarantees (e.g. try writing a spin lock in OpenCL).

The Good the Bad and the Ugly

- Threading like its 2011
- Next generation heterogeneous programming
- Parallel languages/tools will never get it right. I give up

My optimistic view from 2005 …

We've learned our lesson … we emphasize a small number of industry standards

Parallel Programming API's today

- Thread Libraries
  - Win32 API
  - POSIX threads.
- Compiler Directives
  - OpenMP - portable shared memory parallelism.
- Message Passing Libraries
  - MPI - message passing
- Coming soon … a parallel language for managed runtimes? Java or X10?

We don't want to scare away the programmers … Only add a new API/language if we can't get the job done by fixing an existing approach.

Third party names are the property of their owners.

But we didn’t learn our lesson

History is repeating itself!

A small sampling of Programming environments from the NEW golden age of parallel programming **(from the literature 2010-2012)**

AM++
ArBB
BSP
C++11
C++AMP
Charm++
Chapel
Cilk++
CnC
coArray Fortran
Codelets

Copperhead
CUDA
DryadOpt
Erlang
Fortress
GA
GO
Gossamer
GPars
GRAMPS
Hadoop
HMMP

ISPC
Java
Liszt
MapReduce
MATE-CG
MCAPI
MPI
NESL
OoOJava
OpenMP
OpenCL
OpenSHMEM

OpenACC
PAMI
Parallel Haskell
ParalleX
PATUS
PLINQ
PPL
Pthreads
PXIF
PyPar
Plan42
RCCE

Scala
SIAL
STAPL
STM
SWARM
TBB
UPC
Win32
threads
X10
XMT
ZPL

We’ve slipped back into the “just create a new language” mentality.

Note: I’m not criticizing these technologies. I’m criticizing our collective urge to create so many of them.

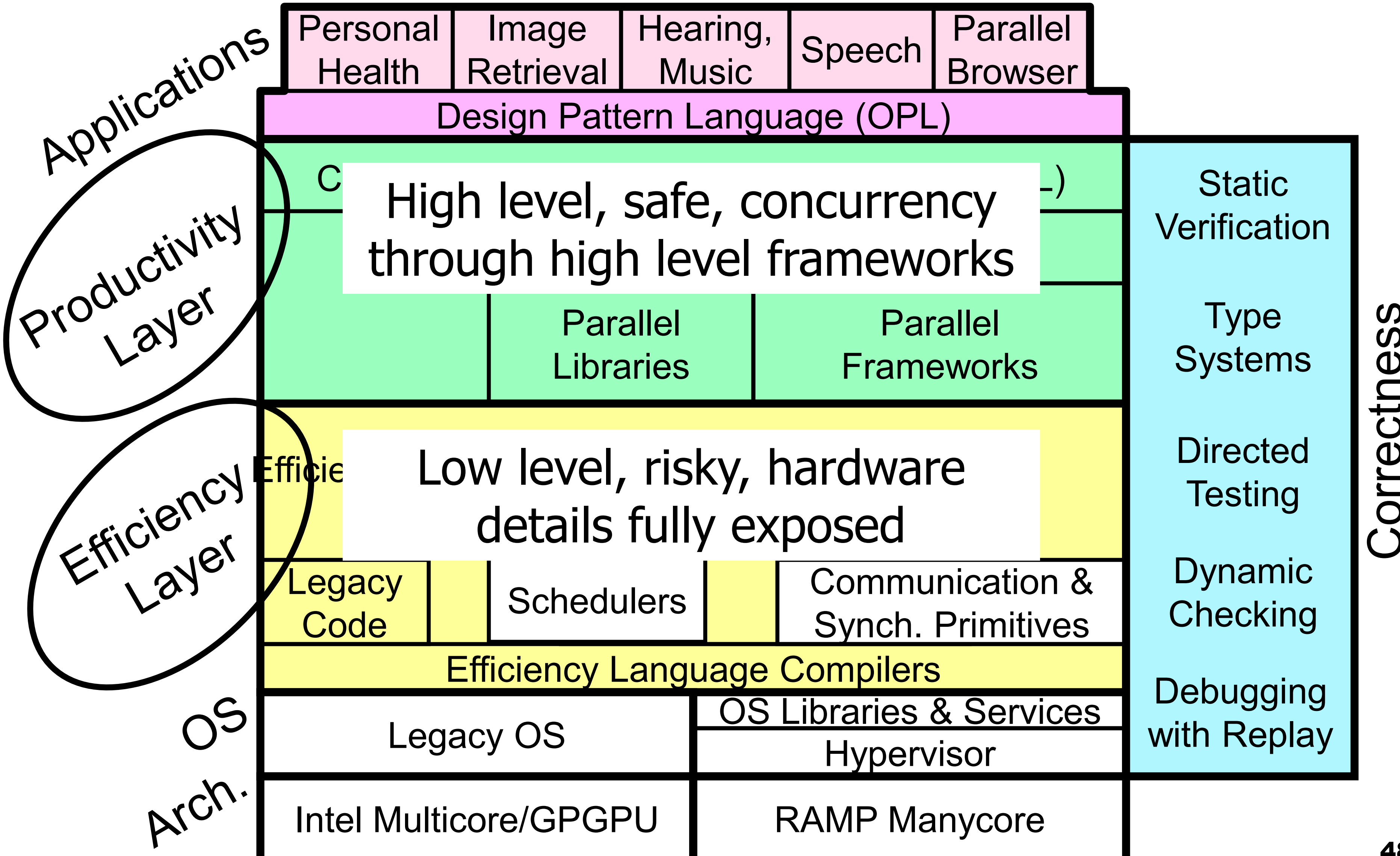
I give up ….

- Computer Scientists are just going to make things worse … creating new languages instead of making the ones we have work well (with the tools we need).
- We application developers must take charge of our own destiny.
- We need to:
  - Raise the level of abstraction so our programming model matches the mathematics of our domain.
  - Build frameworks we can maintain that hide the computer Science mess from our desire to do real work.
- Examples:
  - Trilliois, Cactus, PETsc …
  - The Combinatorial BLAS and KDT

Discussed in the next batch of slides

Par Lab Research Overview

Easy to write correct software that runs efficiently on manycore



A Design Pattern Language for Engineering Parallel applications

Applications

Structural Patterns

- Pipe-and-Filter
- Agent-and-Repository
- Process-Control
- Event-Based/Implicit-Invocation
- Puppeteer
- Model-View-Controller
- Iterative-Refinement
- Map-Reduce
- Layered-Systems
- Arbitrary-Static-Task-Graph

Computational Patterns

- Graph-Algorithms
- Dynamic-Programming
- Dense-Linear-Algebra
- Sparse-Linear-Algebra
- Unstructured-Grids
- Structured-Grids
- Graphical-Models
- Finite-State-Machines
- Backtrack-Branch-and-Bound
- N-Body-Methods
- Circuits
- Spectral-Methods
- Monte-Carlo

Concurrent Algorithm Strategy Patterns

- Task-Parallelism
- Recursive-splitting
- Data-Parallelism
- Pipeline
- Discrete-Event
- Geometric-Decomposition
- Speculation

Implementation Strategy Patterns

Program structure

- SPMD
- Strict-Data-Par.
- Fork/Join
- Actors
- Master/Worker
- Graph-Partitioning
- Loop-Par.
- BSP
- Task-Queue

Data structure

- Shared-Queue
- Shared-Hash-Table
- Distributed-Array
- Shared-Data

Parallel Execution Patterns

Advancing "program counters"

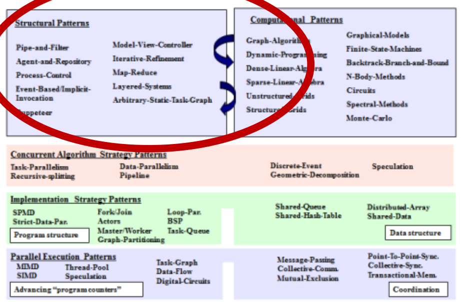
- MIMD
- SIMD
- Thread-Pool
- Speculation
- Task-Graph
- Data-Flow
- Digital-Circuits

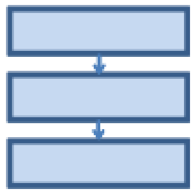
Coordination

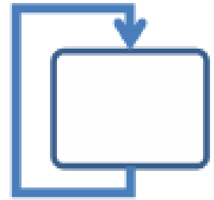
- Message-Passing
- Collective-Comm.
- Mutual-Exclusion
- Point-To-Point-Sync.
- Collective-Sync.
- Transactional-Mem.

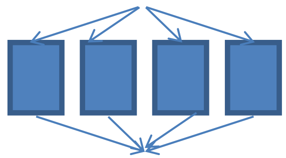
Source: Keutzer and Mattson, Intel Technology Journal, Vol. 13, Issue 4, 2010.

Pattern examples



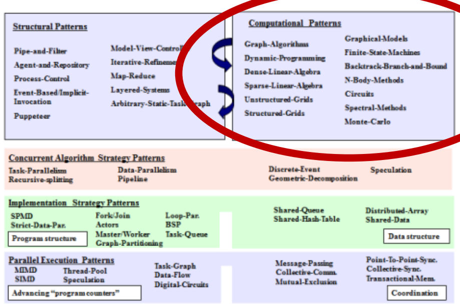


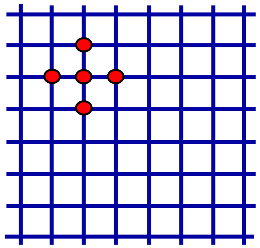


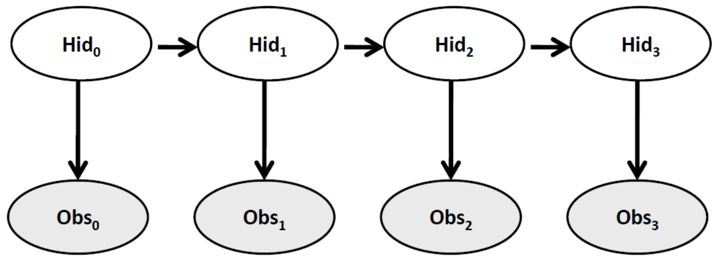


- Pipe-and-Filter
- Iterative refinement
- MapReduce

Structural Patterns: Define the software structure .. *Not* what is computed

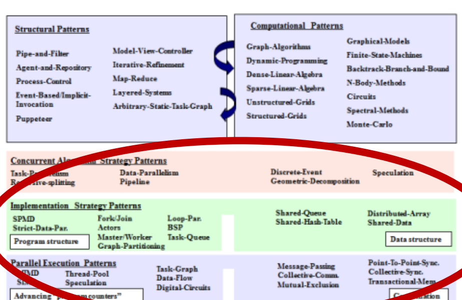






- Structured mesh
- Graphical Models

Computational Patterns: Define the computations "inside the boxes"

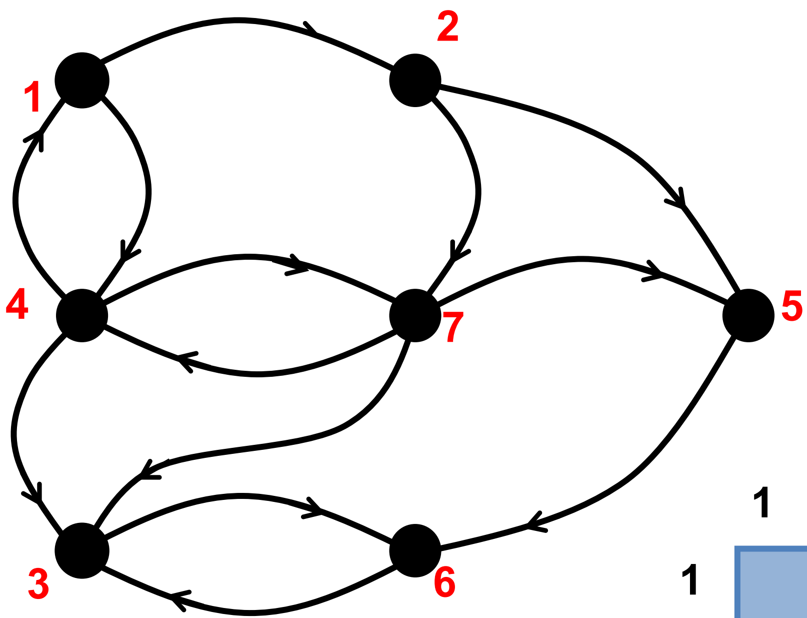


- Fork-join
- SPMD
- Data parallel

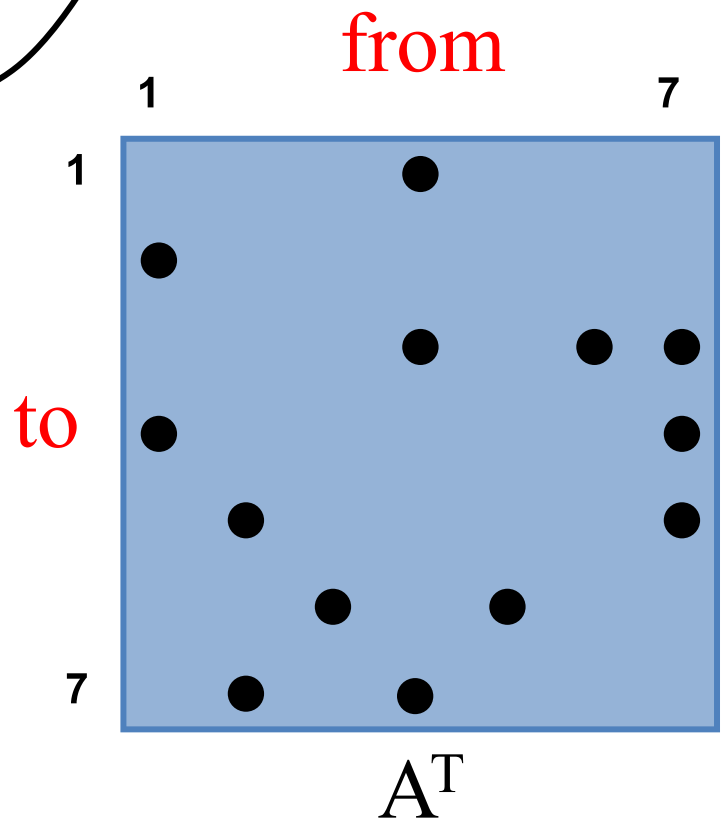
Parallel Patterns: Defines parallel algorithms

Acknowledgements

- I am a cheerleader and funder of this work, but I didn't do any of it myself
  - Content on the Combinatorial BLAS and KDT come from Aydin Buluc of LBNL and John Gilbert of UC Santa Barbara.
  - The integration of SEJITS with KDT was carried out by Aydin Buluc with Shoaib Kamil of MIT, Armando Fox of UCB, Adam Lugowski of UCSB, Lenny Oliker of LBNL, and Sam Williams of LBNL

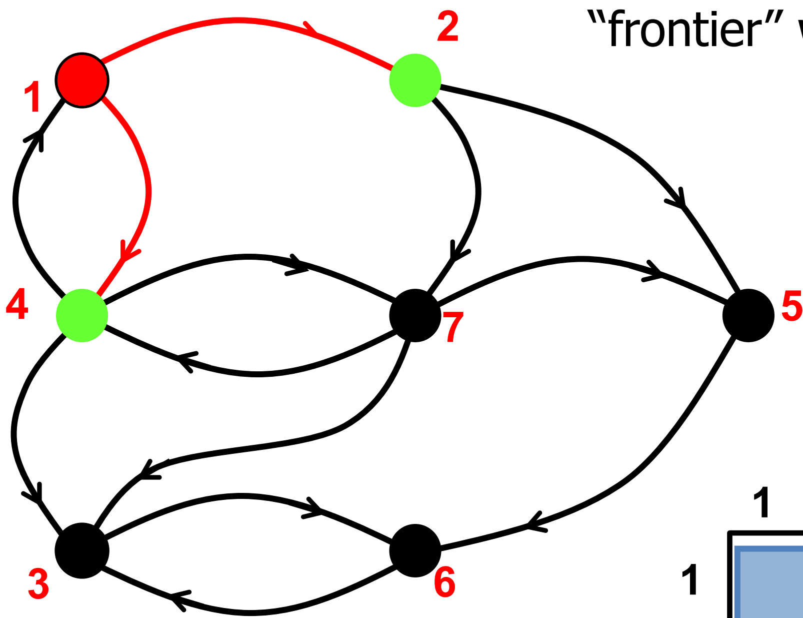


Consider the Breadth First Search Problem … I want to know for each node in the graph, which node is its parent.



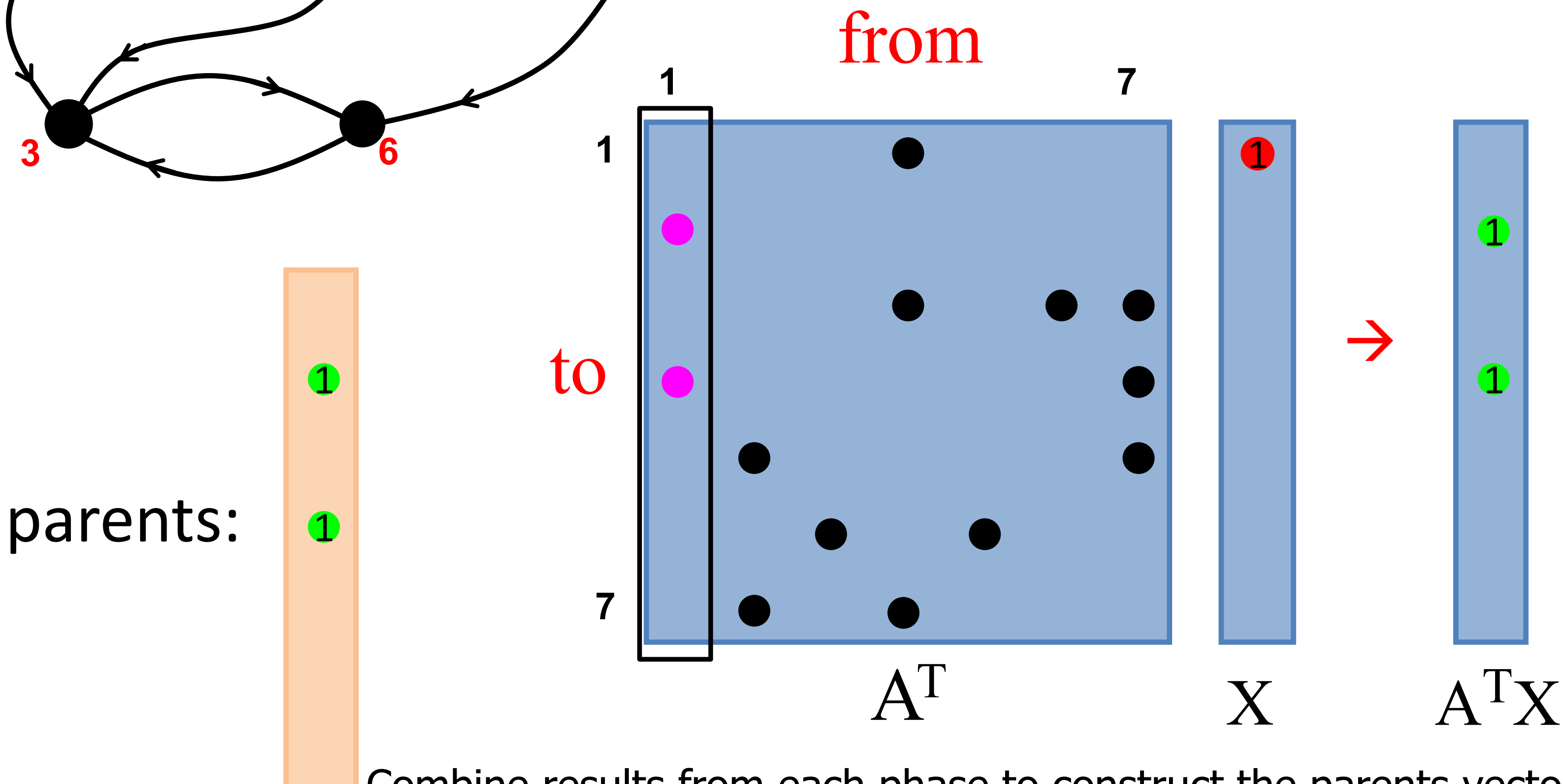
A = the adjacency matrix … Elements nonzero when vertices are adjacent

Start BFS from vertex 1. Each phase updates the "frontier" which is used in the next step

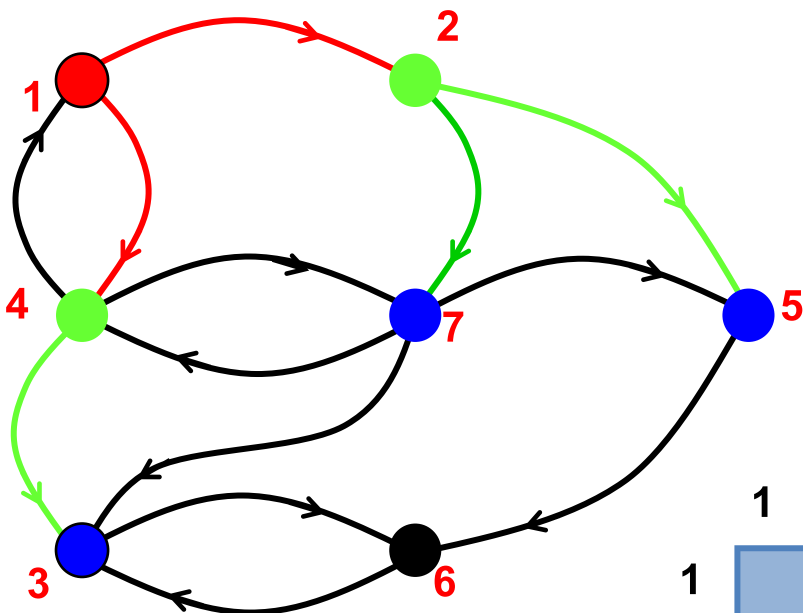


Use the sparse matrix vector multiplication pattern but replace the two traditional (*,+) operations with:

- **Multiply:** select
- **Add:** minimum

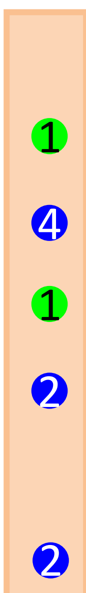


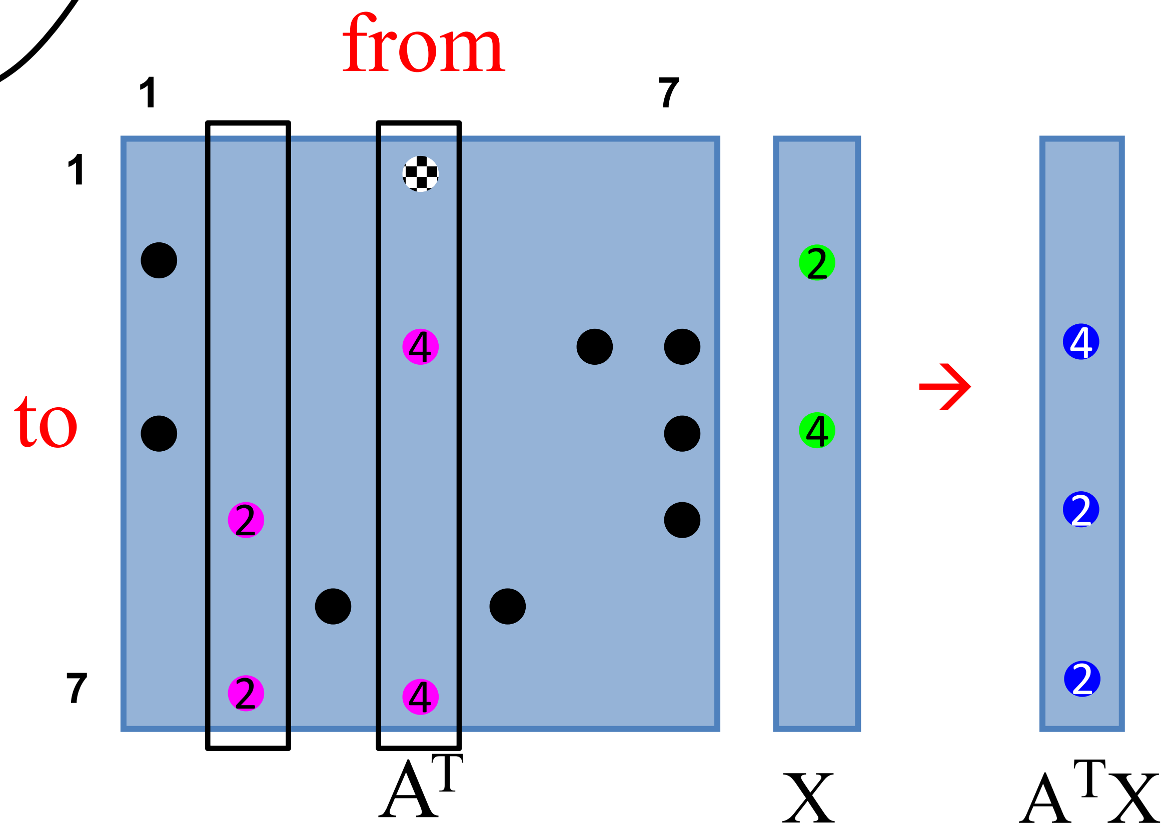
Combine results from each phase to construct the parents vector.

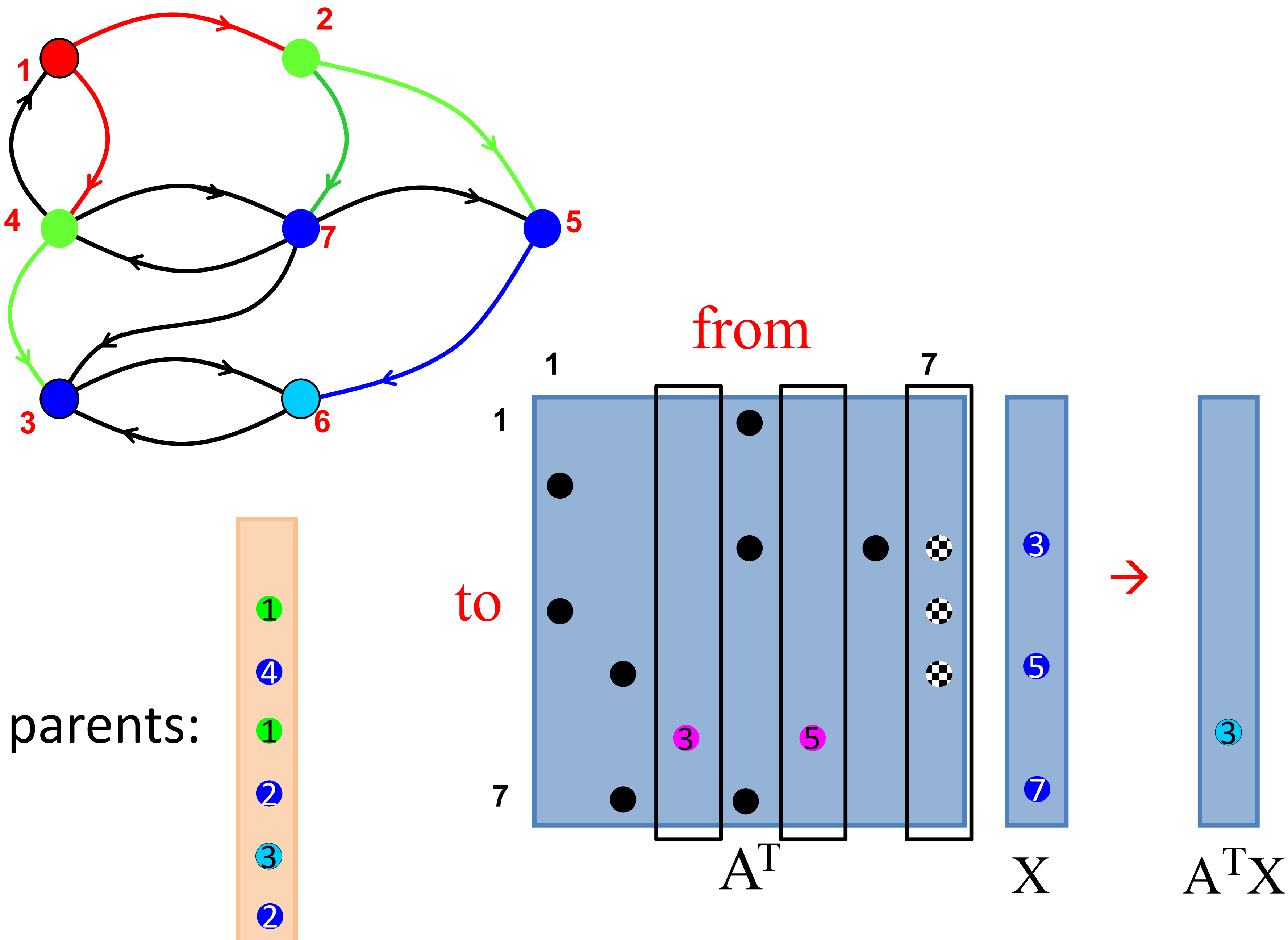


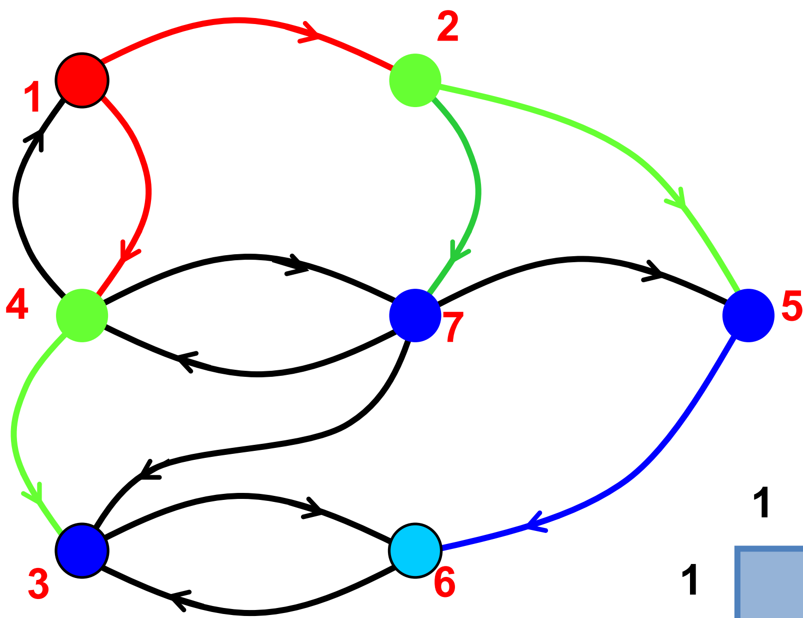
When multiple parents, Select vertex with <u>minimum</u> label as parent

parents:

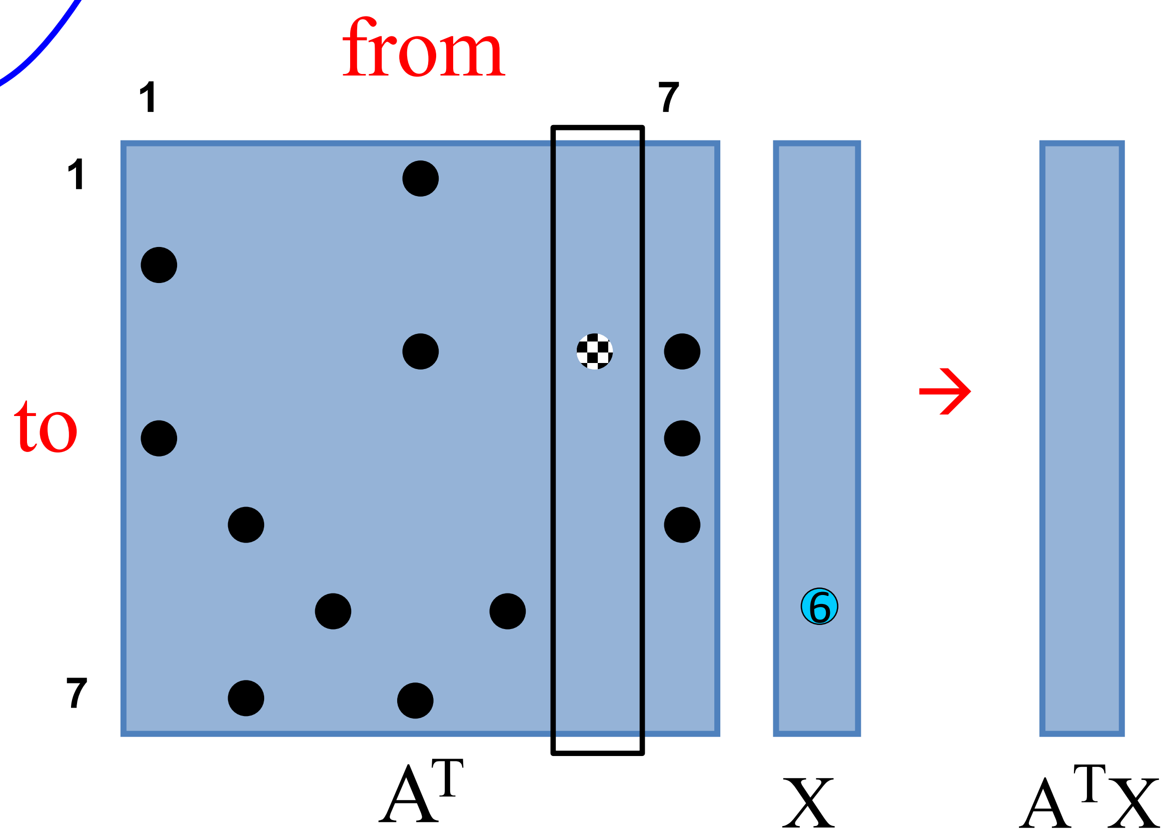




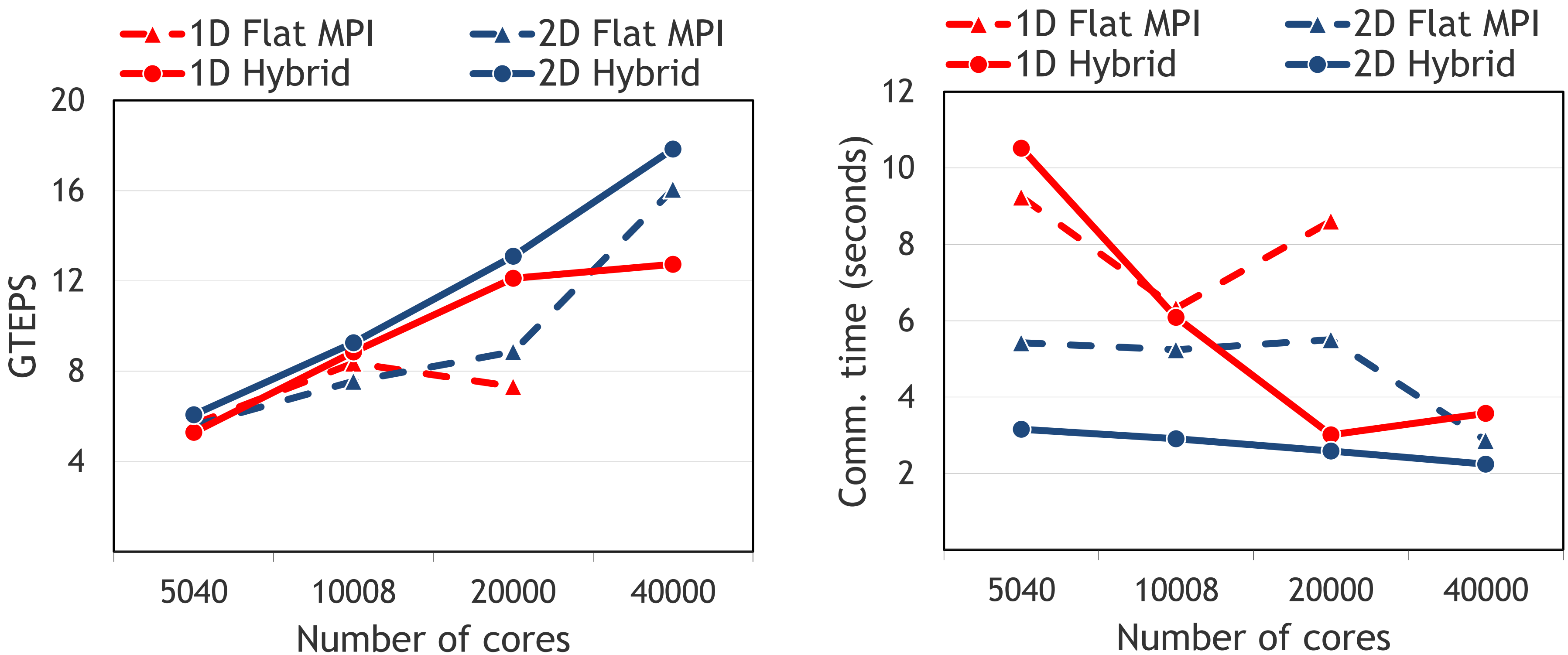

1
2
4
7
5
3
6
parents:
1
4
1
2
3
2
from
1
7
to
1
7
3
5
3
5
7
A^T
X
→
3
A^T X



Extended to matrix-matrix multiply, this primitive represents multi-source one-hop breadth-first search and combine … which is the foundation of many graph algorithms.



BFS strong scaling

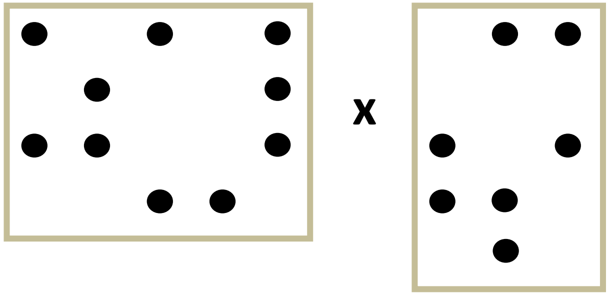


- NERSC Hopper (Cray XE6, Gemini interconnect AMD Magny-Cours)
- Hybrid: In-node 6-way OpenMP multithreading
- Graph500 (R-MAT): 4 billion vertices and 64 billion edges.

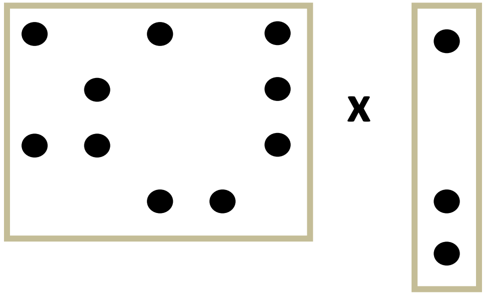
B., Madduri. Parallel breadth-first search on distributed memory systems. *Supercomputing*, 2011.

Linear-algebraic primitives

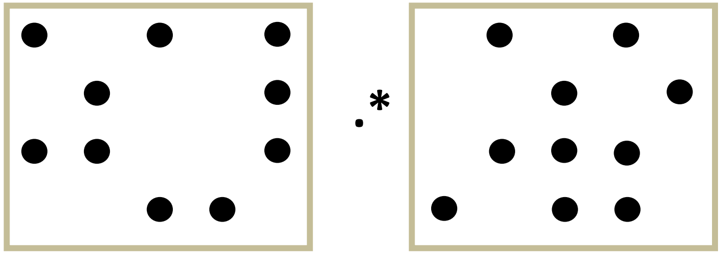
Sparse matrix-sparse matrix multiplication



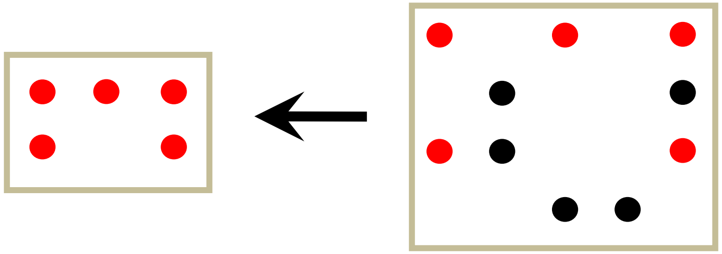
Sparse matrix-sparse vector multiplication



Element-wise operations



Sparse matrix indexing



The Combinatorial BLAS implements these, and more, on arbitrary semirings, e.g. (·, +), (and, or), (+, min)

Some Combinatorial BLAS functions

Function	Applies to	Parameters		Returns	Matlab Phrasing
SpGEMM	Sparse Matrix (as friend)	**A, B**: trA: trB:	sparse matrices transpose **A** if true transpose **B** if true	Sparse Matrix	$\mathbf{C} = \mathbf{A} * \mathbf{B}$
SpMV	Sparse Matrix (as friend)	**A**: **x**: trA:	sparse matrices sparse or dense vector(s) transpose **A** if true	Sparse or Dense Vector(s)	$\mathbf{y} = \mathbf{A} * \mathbf{x}$
SpEWiseX	Sparse Matrices (as friend)	**A, B**: notA: notB:	sparse matrices negate **A** if true negate **B** if true	Sparse Matrix	$\mathbf{C} = \mathbf{A} * \mathbf{B}$
Reduce	Any Matrix (as method)	dim: *binop*:	dimension to reduce reduction operator	Dense Vector	sum(**A**)
SpRef	Sparse Matrix (as method)	**p**: **q**:	row indices vector column indices vector	Sparse Matrix	$\mathbf{B} = \mathbf{A}(\mathbf{p}, \mathbf{q})$
SpAsgn	Sparse Matrix (as method)	**p**: **q**: **B**:	row indices vector column indices vector matrix to assign	none	$\mathbf{A}(\mathbf{p}, \mathbf{q}) = \mathbf{B}$
Scale	Any Matrix (as method)	**rhs**:	any object (except a sparse matrix)	none	Check guiding principles 3 and 4
Scale	Any Vector (as method)	**rhs**:	any vector	none	none
Apply	Any Object (as method)	*unop*:	unary operator (applied to non-zeros)	None	none

The case for graph primitives based on sparse matrices

Many irregular applications contain coarse-grained parallelism that can be exploited by abstractions at the proper level.

Traditional graph computations
Data driven, unpredictable communication.
Irregular and unstructured, poor locality of reference
Fine grained data accesses, dominated by latency

The case for graph primitives based on sparse matrices

Many irregular applications contain coarse-grained parallelism that can be exploited by abstractions at the proper level.

Traditional graph computations	Graphs in the language of linear algebra
Data driven, unpredictable communication.	Fixed communication patterns
Irregular and unstructured, poor locality of reference	Operations on matrix blocks exploit memory hierarchy
Fine grained data accesses, dominated by latency	Coarse grained parallelism, bandwidth limited

A new effort to define the BLAS of graphs-as-linear-algebra

- There are graph algorithms that require interaction between graph elements making a map-reduce style of computing impractical.
- Representing graphs in terms of linear algebra operations over semi-rings, is a well known technique.
- There is a great deal of variation in graph frameworks exposed to data-scientists … standardization at this high level makes no sense.
- The underlying primitives, however, are stable and ready to standardize.
  - Standardization enables vendor optimizations (e.g. the BLAS)
  - Standardization is efficient … keeps people from wasting time "reinventing the wheel".

To be presented at HPEC'13, Boston MA, Sept 11, 2013

A new effort to define the BLAS of graphs-as-linear-algebra

Standards for Graph Algorithm Primitives

Tim Mattson (Intel Corporation), David Bader (Georgia Institute of Technology), Jon Berry (Sandia National Laboratory), Aydin Buluc (Lawrence Berkeley National Laboratory), Jack Dongarra (University of Tennessee), Christos Faloutsos (Carnegie Melon University), John Feo (Pacific Northwest National Laboratory), John Gilbert (University of California at Santa Barbara), Joseph Gonzalez (University of California at Berkeley), Bruce Hendrickson (Sandia National Laboratory), Jeremy Kepner (Massachusetts Institute of Technology), Charles Leiserson (Massachusetts Institute of Technology), Andrew Lumsdaine (Indiana University), David Padua (University of Illinois at Urbana-Champaign), Stephen Poole (Oak Ridge National Laboratory), Steve Reinhardt (Cray Corporation), Mike Stonebraker (Massachusetts Institute of Technology), Steve Wallach (Convey Corporation), Andrew Yoo (Lawrence Livermore National Laboratory)

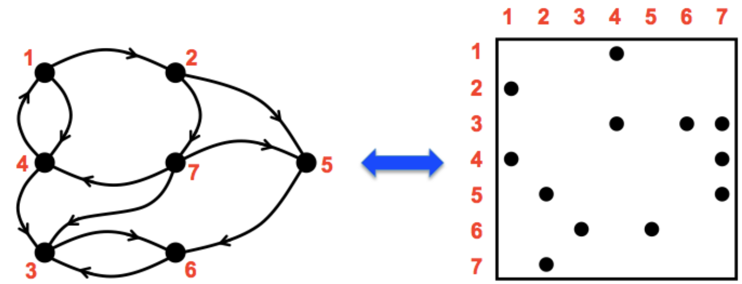
To be presented at HPEC'13, Boston MA, Sept 11, 2013

Knowledge

Discovery

Toolbox

http://kdt.sourceforge.net/

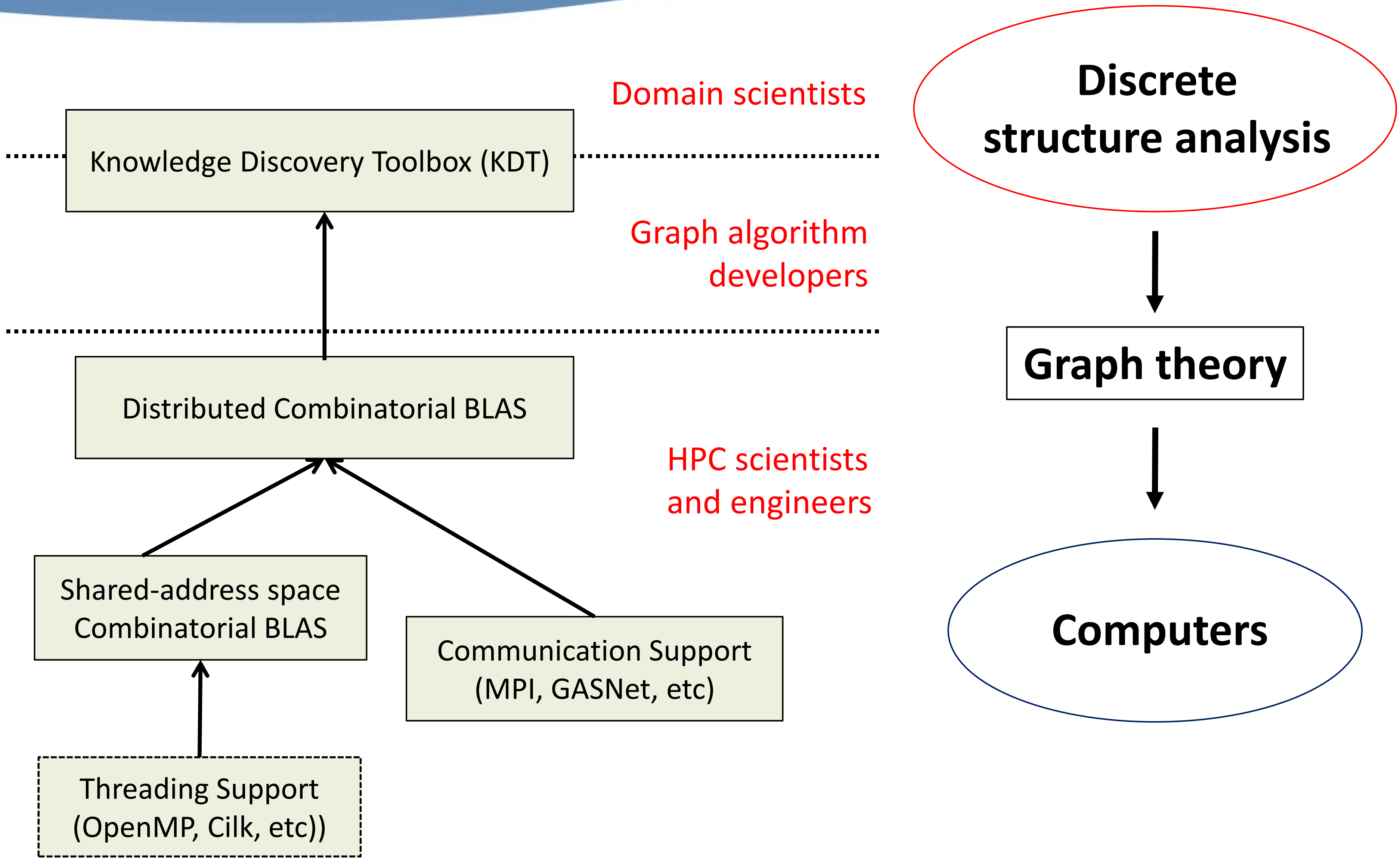


A general graph library with operations based on linear algebraic primitives

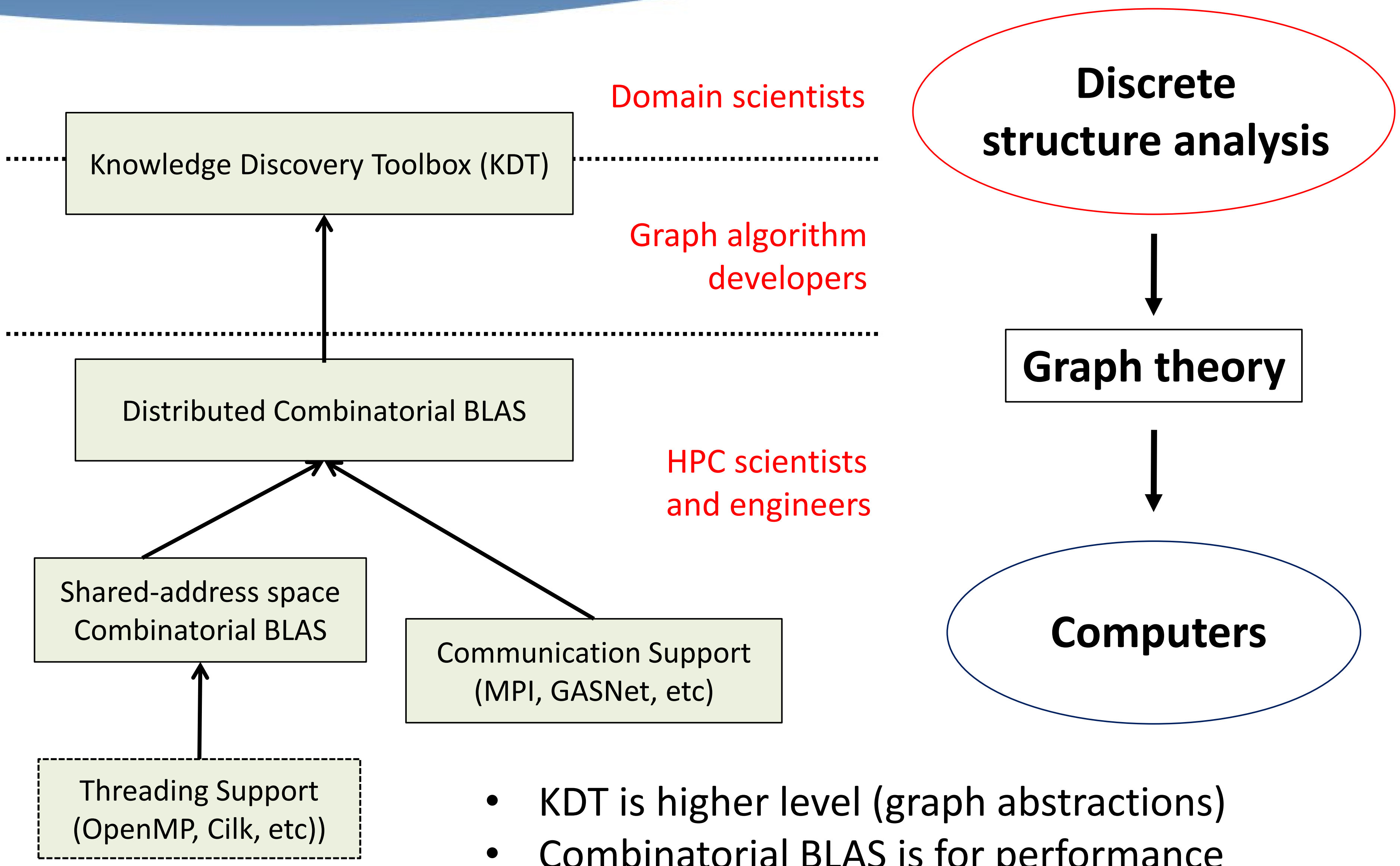
- Aimed at domain experts who know their problem well but don’t know how to program a supercomputer
- Easy-to-use Python interface
- Runs on a laptop as well as a cluster with 10,000 processors

- Open source software (New BSD license)
- V0.3 release April 2013

Parallel Graph Analysis Software



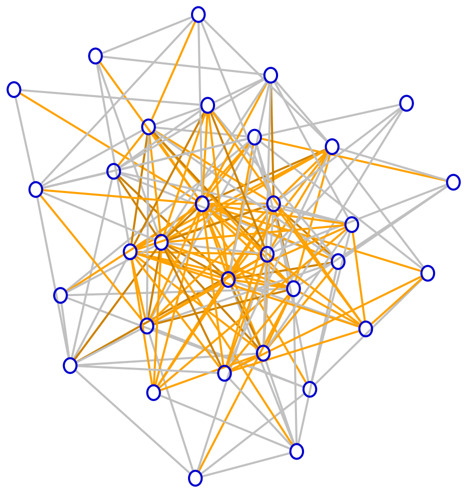
Parallel Graph Analysis Software



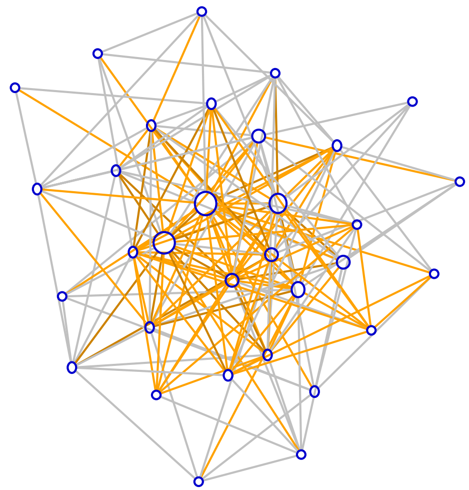
- KDT is higher level (graph abstractions)
- Combinatorial BLAS is for performance

The need for filters

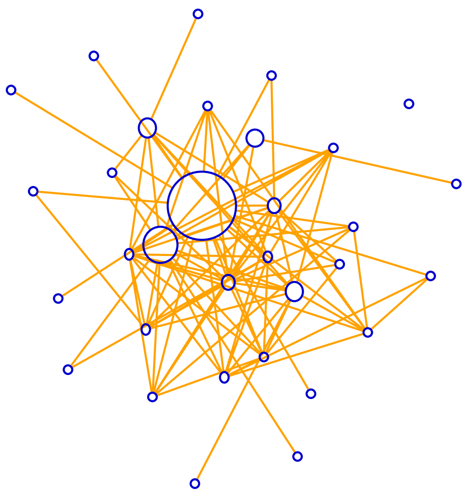
Graph of text & phone calls



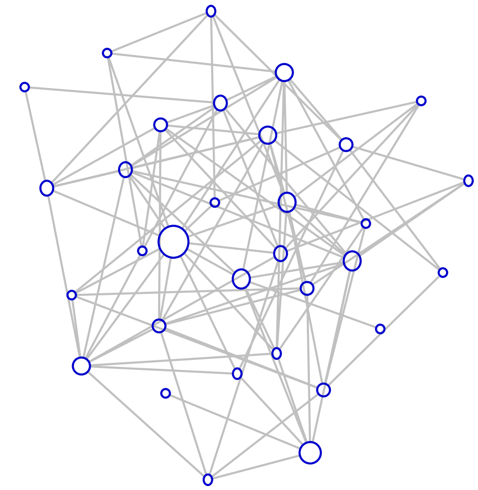
Betweenness centrality



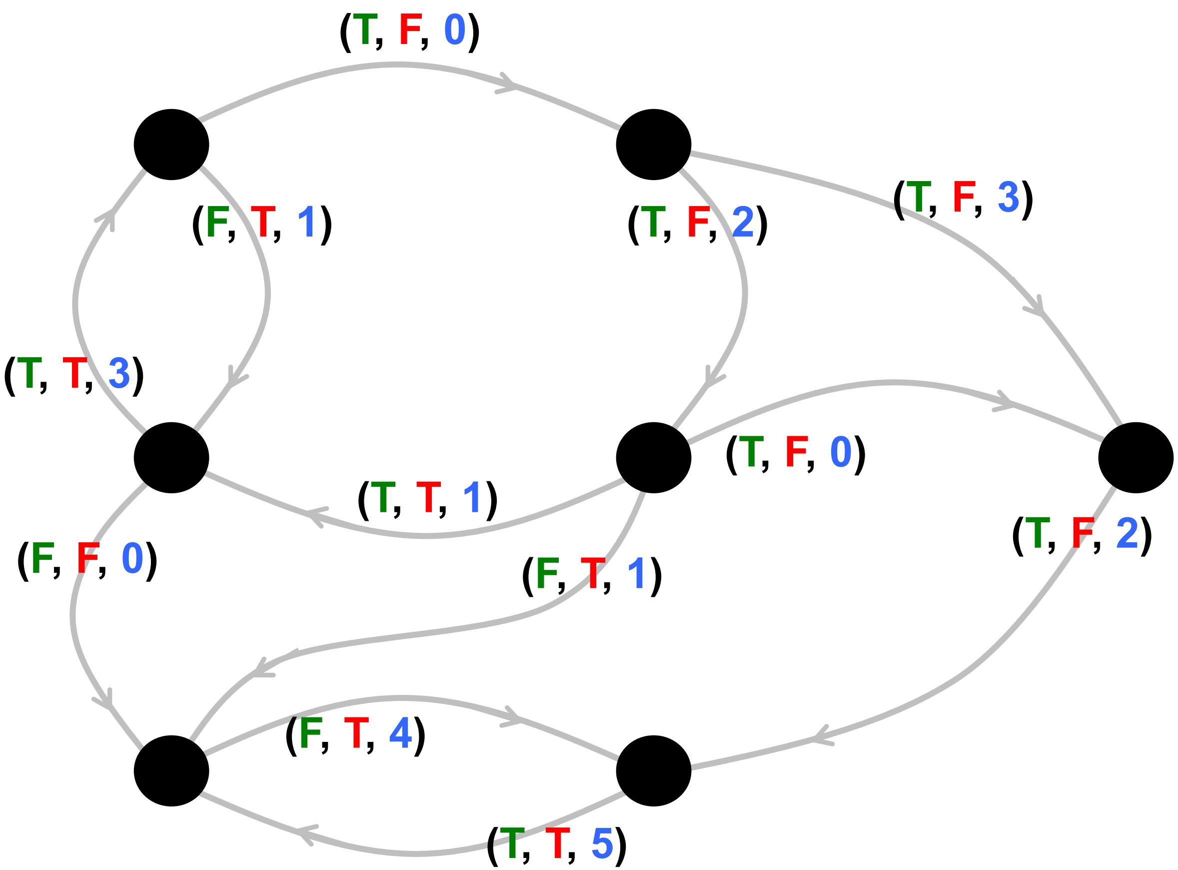
Betweenness centrality on text messages



Betweenness centrality on phone calls



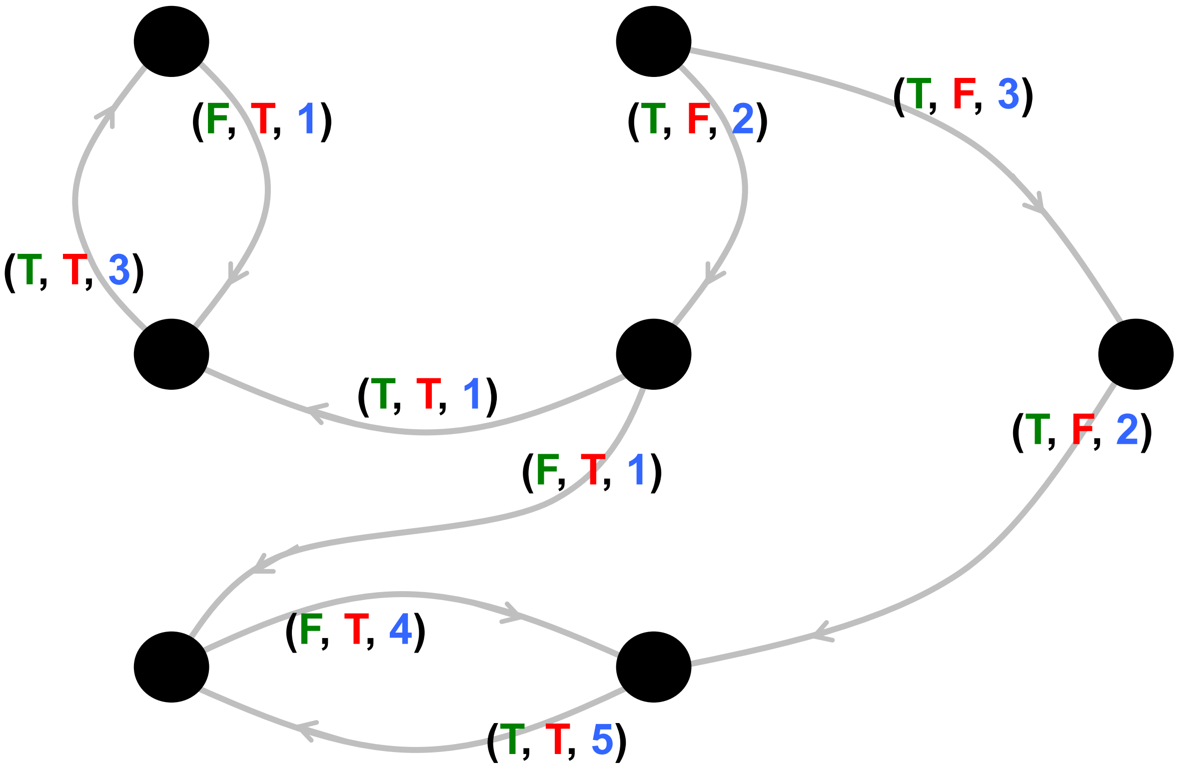
Edge filter illustration




```
class edge_attr:
    isText
    isPhoneCall
    weight
```

Edge filter illustration

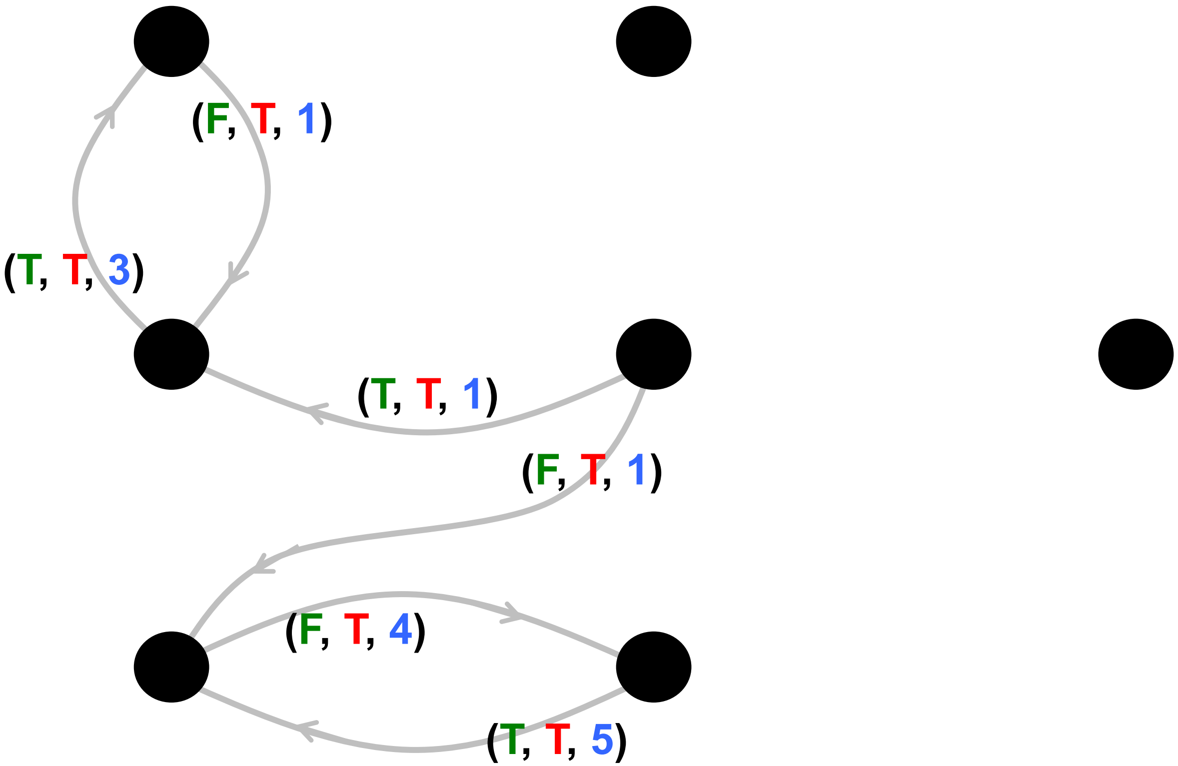
```
G.addEFilter(lambda e: e.weight > 0)
```




```
class edge_attr:
    isText
    isPhoneCall
    weight
```

Edge filter illustration

```
G.addEFilter(lambda e: e.weight > 0)
G.addEFilter(lambda e: e.isPhoneCall)
```

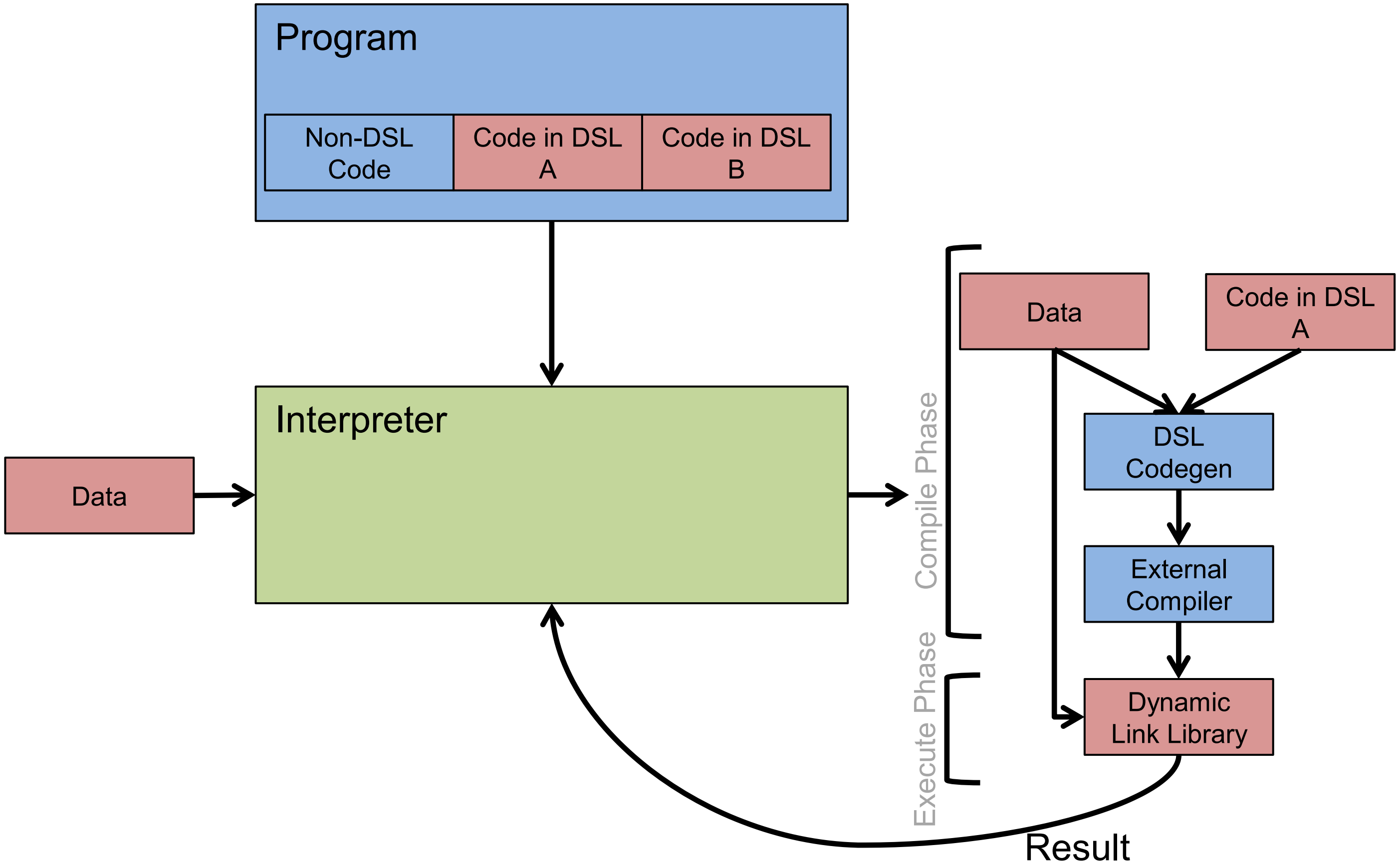



```
class edge_attr:
    isText
    isPhoneCall
    weight
```

Problems with Customizing in KDT

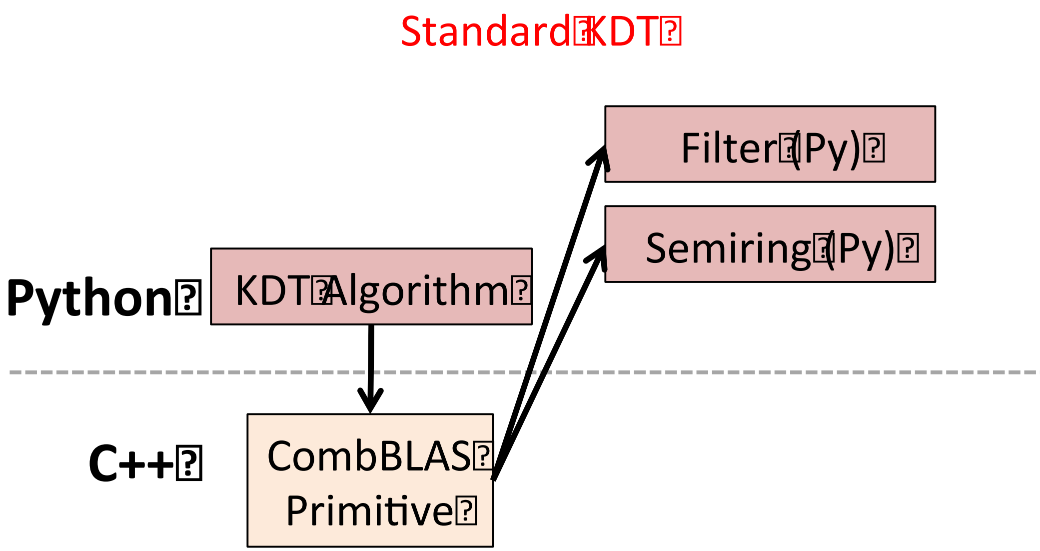
- Filtering on attributed semantic graphs is slow
  - In plain KDT, filters are pure Python functions.
  - Requires a per-vertex or per-edge upcall into Python
  - Can be as slow as 80X compared to pure C++
- Adding new graph algorithms to KDT is slow
  - A new graph algorithm = composing linear algebraic primitives + customizing the *semiring* operation
  - *Semirings* in Python; similar performance bottleneck

Review: Selective Embedded Just In Time Specialization (SEJITS)

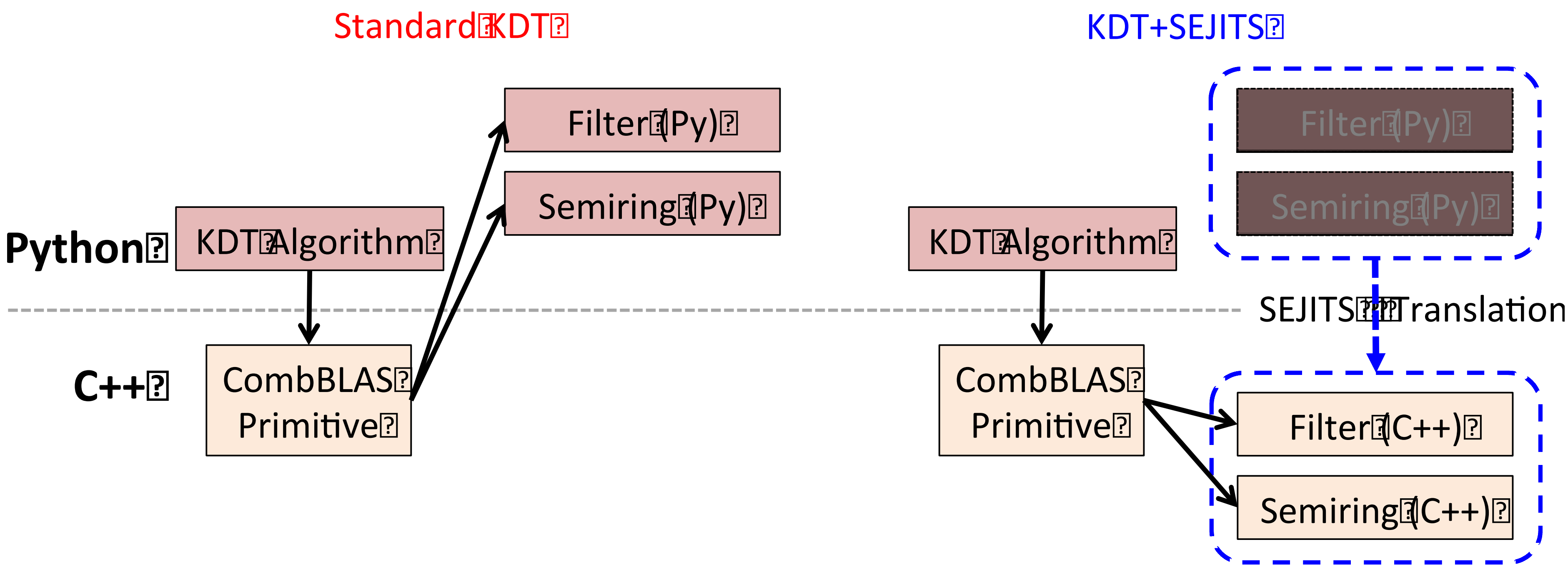


Catanzaro, Kamil, Lee, Asanovic, Demmel, Keutzer, Shalf, Yelick, Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. *PMEA,* 2009

SEJITS for filter/semiring acceleration



SEJITS for filter/semiring acceleration

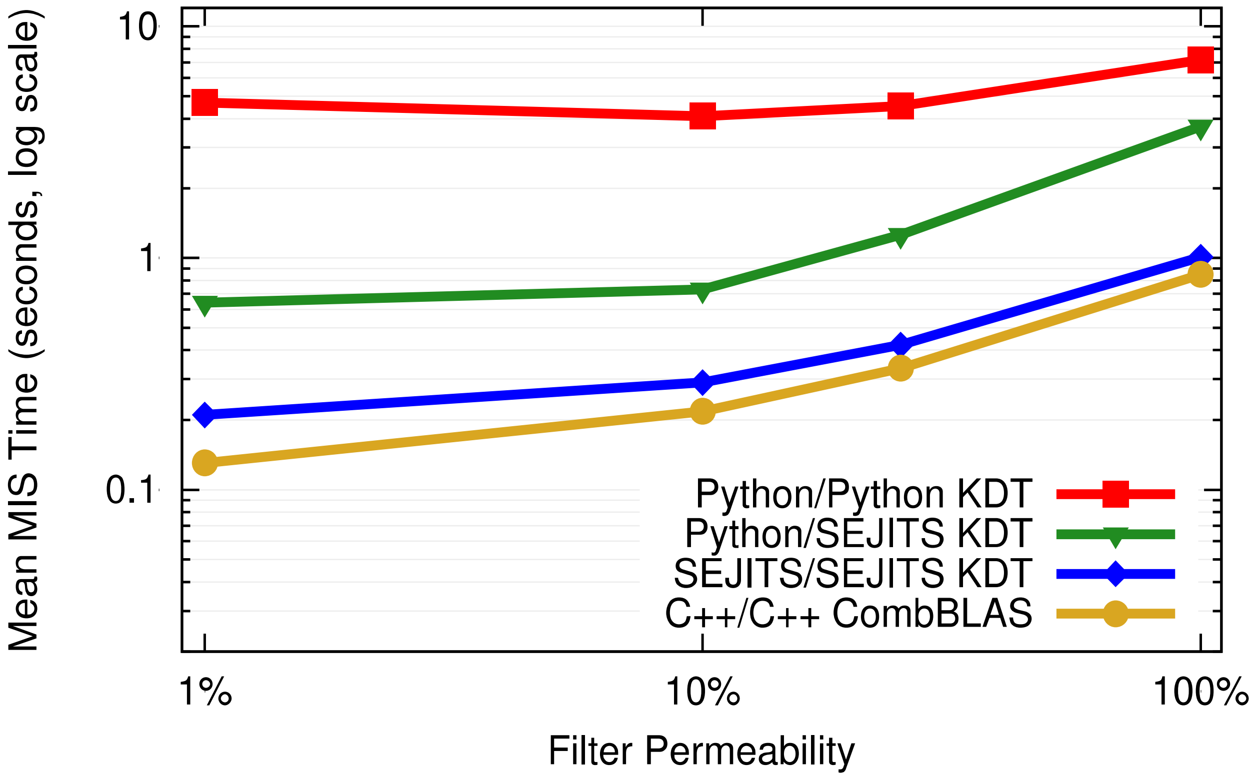


Embedded DSL: Python for the whole application

- Introspect, translate Python to equivalent C++ code
- Call compiled/optimized C++ instead of Python

B., Duriakova, Gilbert, Fox, Kamil, Lugowski, Oliker, Williams. High-Performance and High-Productivity Analysis of Filtered Semantic Graphs, *IPDPS*, 2013

SEJITS+KDT multicore performance



- MIS= Maximal Independent Set
- 36 cores of Mirasol(Intel® Xeon™ E7-8870 processor)
- Erdős-Rényi matrix (Scale 22, edgefactor=4)

Synthetic data with weighted randomness to match filter permeability

Notation: [semiring impl] / [filter impl]

Summary … we've discussed 3 recent developments in Parallel computing …

- C++'11 Standardizes state of the art in multi-threading.
- OpenCL continues to evolve … expanding the range of algorithms it can address (nested parallelism) and support the latest devices with HW supported shared address spaces (SVM).
- Application specific BLAS-like libraries and software transformation tools (e.g. SEJITS) suggest a different path to solving the parallel programming problem.

I leave the assignment of "the Good, the bad and the ugly" to you.