Programming ExaScale Systems

Tim Mattson

Parallel Computing Lab

Acknowledgements: I had help on this presentation from the following people at Intel … Jeff Hammond, Rob van der Wijngaart, Srinivas Sridharan, and Romain Cledat

Agenda

➡
- Preliminaries:
  - Setting the stage for a conversation about Exascale computing.
- Methodologies:
  - Benchmarks, dirty hands, and the NIH syndrome.
- The landscape of Exascale runtimes
  - Who are we watching?
- Are tasks a productive path?
  - Some suggestive but inconclusive results
- The ACR program
  - Another spelling of OCR (a task based runtime)

Preliminaries: Some definitions

- **ExaScale Computer**: An ensemble of nodes with aggregate performance of 10^{18} operations per second when running a single exascale application.
- **ExaScale Application**: An Application composed a vast number of *interacting* tasks for which a single invocation scales to make *effective* use of the full exaScale System.
- **ExaSkeptic**: A curmudgeon who questions the sanity of trying to build an exaScale computer requiring applications with O(billion) concurrency and a 20 MWatt power budget by 2020.

Cloud

A ~~grid~~ of 1000 petaFLOP computers is not an ExaScale computer.
A parameter sweep problem is not an ExaScale Application.

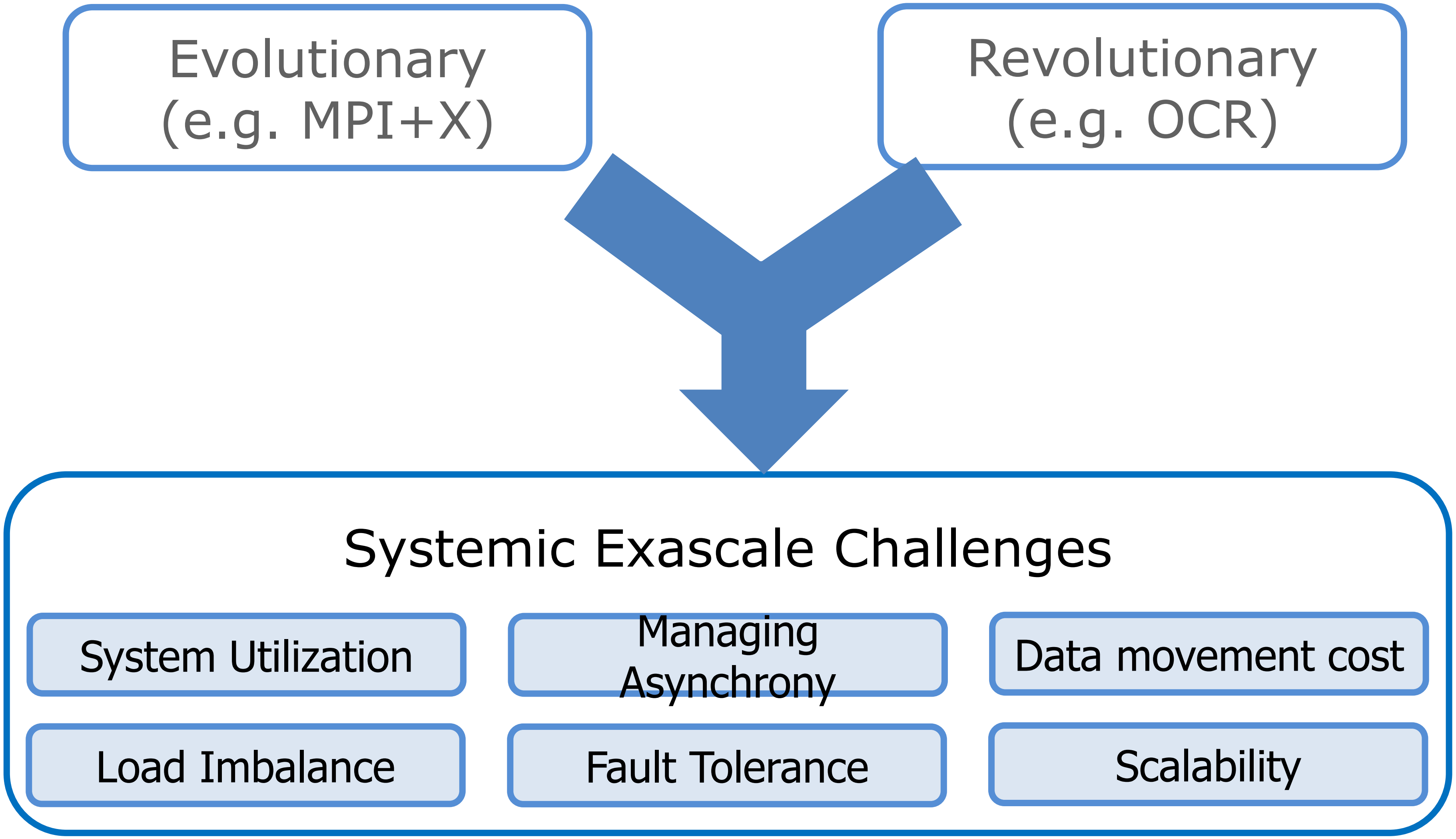
Preliminaries: Concerns of an ExaSkeptic

- Most scientists are still trying to figure out what to do with PetaScale … why are we so eager for exaScale?.
- Most of our collective energy should be directed towards mega-PetaScale
  - Open Standard programming models (MPI, OpenMP, OpenCL)
  - Frameworks that support common patterns … programmers write apps by plugging mostly serials patches into these frameworks.
- **If we build an exaScale machine in 2020 running at 20 MWatts … will it be so bizarre that the techniques utilized are unlikely to inform what we do in mainstream HPC?**

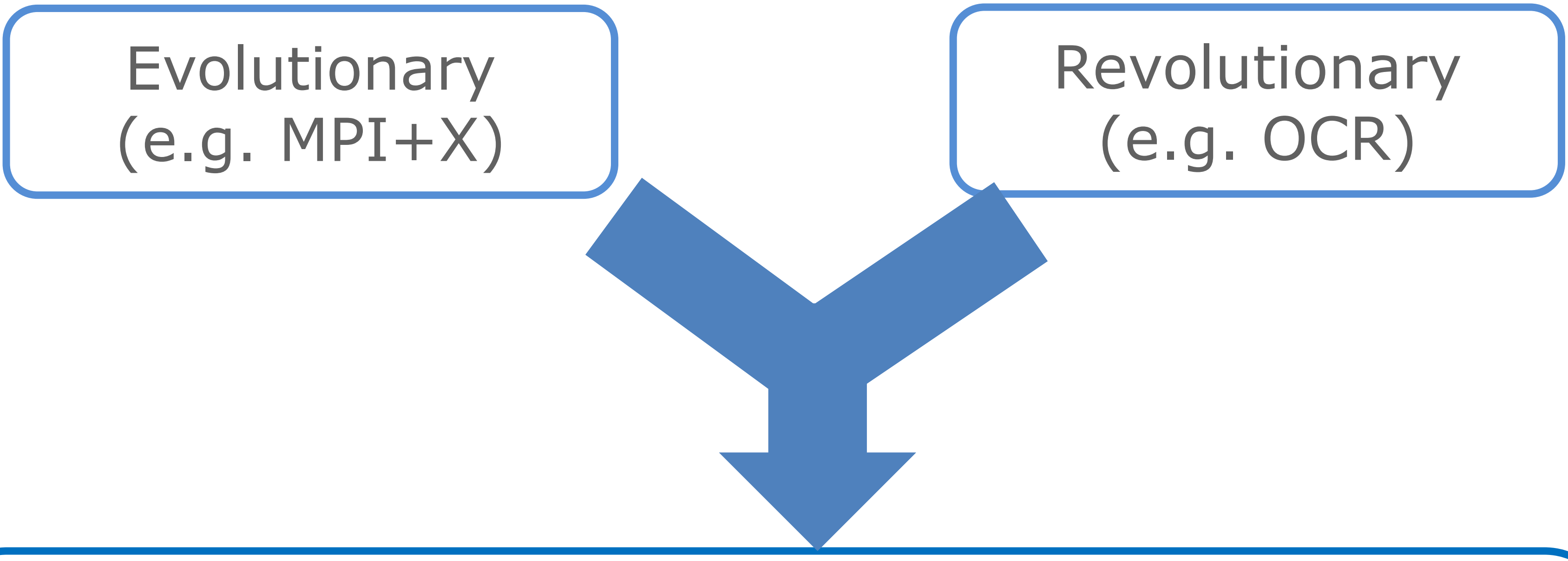
But for now … I will suppress my ExaSkeptic mindset and "drink the cool-aid".

*Third party names are the property of their owners.

2 pathways to Exascale Runtime Research



2 pathways to Exascale Runtime Research



Systemic Exascale Challenges

We love MPI+X and believe it can be made to work if programmers use new features in MPI 3 (or maybe MPI 4).

MPI+X is the “status quo” and is well taken care of … so we can focus exclusively on revolutionary approaches.

Agenda

- Preliminaries:
  - Setting the stage for a conversation about Exascale computing.
- Methodologies:
  - Benchmarks, dirty hands, and the NIH syndrome.
- The landscape of Exascale runtimes
  - Who are we watching?
- Are tasks a productive path?
  - Some suggestive but inconclusive results
- The ACR program
  - Another spelling of OCR (a task based runtime)

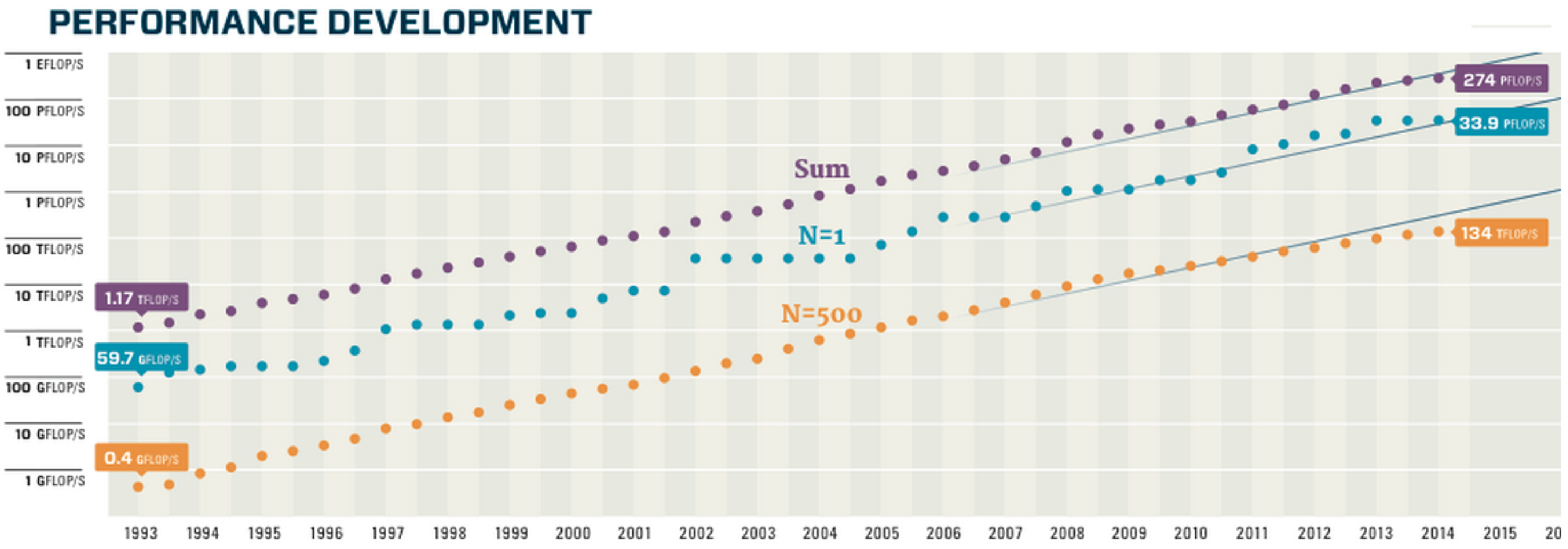
How to design a great supercomputer?

- Hardware is only useful to the extent it helps you solve problems you care about.
- Therefore, you must understand the application software people will run to guide hardware design.

High Quality Benchmarks are essential for effective system design.

The number one benchmark in use today!

- MP Linpack and the Top500 list.   It's a lot of fun … provides a far reaching, historical performance metric.

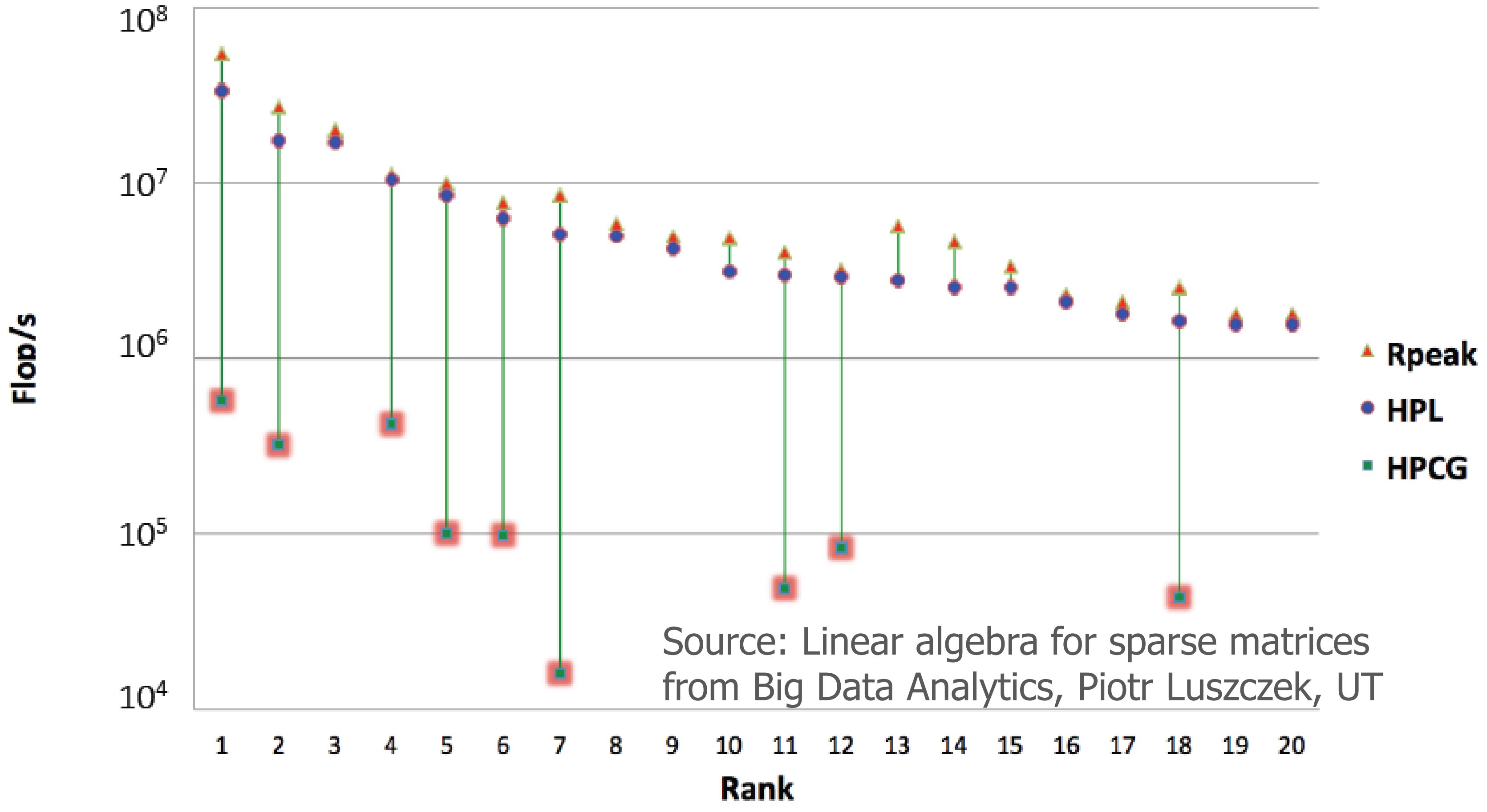


- Most real applications don't look anything like MP-Linpack.

The drive to "set records" has led to machines of questionable value.
Focusing on the wrong benchmark has damaged HPC.

HPCG: A Better benchmark?

- HPCG (High Performance Conjugate Gradient) closely approximates real applications … so it's a "better" benchmark.



Source: Linear algebra for sparse matrices from Big Data Analytics, Piotr Luszczek, UT

- But as applications continue to evolve (e.g. shift to big data and graph analytics) maybe HPCG will be just as misleading as MP-Linpack.

The only rational approach to long term benchmarking?

- Give up … Nobody can confidently predict the key future workloads.   So don’t even try.

- Our conjecture …
  - experienced application programmers know the sorts of scalable operations they will depend on.
  - Therefore, benchmark those scalable ops … a machine that gets those ops right will most likely be good for the apps we will care about in the future.

The Parallel Research Kernels version 1.0

PRK: Low level constructs that capture the essence of what parallel programmers require from parallel computers.

- Dense matrix transpose
- Synchronization: global (collective) and point to point
- Scaled vector addition (Stream triad)*
- Atomic reference counting, both contended and uncontended (locks/TSX)
- Vector reduction
- Sparse matrix-vector multiplication
- Random access update
- Stencil computation
- Dense matrix-matrix multiplication (~DGEMM)
- Branch (inner-loop conditionals + PC panic)*

**embarrassingly parallel*

But don’t we need real apps?

- Real applications must deal with all the messy details C.S. researchers try to avoid
  - Legacy code
  - The ugly “boundary cases”
  - Users
- These “messy details” are essential … so they must be addressed.
- We will find a set of proxy apps to work with and a small number of full apps … stay tuned.

Our approach

- Drive the research by writing lots of apps/code.
- If you don't get your "hands dirty" working with real code, you can be easily fooled by hype.
- Strenuously reject the *Not Invented Here Syndrome*:
- Our goal:
  - Find the "Franken-runtime" that really works.
    - We call this Mary … named after Mary Shelly (or our trusty admin … Mary McCargar-van Arkel). First there was Ada, then the great Linda programming language, and now Mary.

Agenda

- Preliminaries:
  - Setting the stage for a conversation about Exascale computing.
- Methodologies:
  - Benchmarks, dirty hands, and the NIH syndrome.
- The landscape of Exascale runtimes
  - Who are we watching?
- Are tasks a productive path?
  - Some suggestive but inconclusive results
- The ACR program
  - Another spelling of OCR (a task based runtime)

CHARM++ (Sanjay Kale, UIUC)

- CHARM++ is a message-driven execution model supporting an actors programming model.
- Based on message driven relocatable objects.
  - Objects created one-by-one (explicit tasks) or in groups (chare-arrays or chare-groups) to express data-parallel algorithms.
- Charm++ uses over-decomposition with concurrent schedulers that exploit parallel slackness to keep the load balanced (and hide latency).
- Relocatable data-block and objects plus local check-pointing and message logging to support resilience.

http://charm.cs.uiuc.edu/

Global Arrays (PNNL/ANL)

- GA is a global view data model and SPMD execution model that assumes replicated, static distributed or dynamic distributed processing.
- GA is built upon flexible RMA and bulk synchronous data operations and (unfortunately)
- GA encourages the use of shared counters for dynamic load-balancing. Multiple research efforts show work-stealing is a superior approach.
- GA template is FORALL(data): Get-Compute-Update.
- GA allows process-ID (procid) agnostic code and supports resilience through RAID-like data replication.

http://hpc.pnl.gov/globalarrays/

APGAS (asynchronous PGAS)

- Used to describe X10 and Chapel, not traditional PGAS (UPC, CAF, SHMEM, …).
- Locales/places for locality+hierarchy as in MPI but not PGAS
- Begin-sync/spawn-finish for asynchronous tasking
- Violates Mattson's Law (No New Languages!)
- Assuming magical compiler+runtime, APGAS surely can do anything!

Honorable Mentions …

- HPX
  - A many-tasking runtime system based on relocatable PGAS objects and a dataflow model, implemented using asynchronous, remote method invocation.
- Legion
  - event driven task model … similar to OCR.

Agenda

- Preliminaries:
  - Setting the stage for a conversation about Exascale computing.
- Methodologies:
  - Benchmarks, dirty hands, and the NIH syndrome.
- The landscape of Exascale runtimes
  - Who are we watching?
- ➡ Are tasks a productive path?
  - Some suggestive but inconclusive results
- The ACR program
  - Another spelling of OCR (a task based runtime)

More definitions

- **Work**: The sequence of operations defined by an execution of a program.
- **Unit of execution (UE)**: an agent that advances the work defined by an executing program.
- **Data**: The dynamic state embodied by the execution of a program.
- **Memory**: the system that holds the data available to an executing program
- **Task**: A logically related sequence of operations and its associated data environment.

Exascale SW Issues

ExaScale Assumptions
Parallelism ~O(Billion)
MTBF<<App_runtime Global checkpoints unacceptably slow relative to computation time
Data Movement dominates energy and performance
Software lifespan is greater than hardware lifespan

Exascale SW Issues

ExaScale Assumptions	Response …
Parallelism ~O(Billion)	Amdahl's law still applies. Hide overheads … oversubscription? Asynchrony? Aggressive load balancing? All of the above?
MTBF<<App_runtime Global checkpoints unacceptably slow relative to computation time	Resilience must be built into the runtime system … and probably the programming models and algorithms as well.
Data Movement dominates energy and performance	Abstract the hardware but don't' hide it … programmers must be able to control how data maps onto memory.
Software lifespan is greater than hardware lifespan	Portability is essential … and the performance better by "mostly" portable.

Requires decoupling of work from the UEs that carry it out … i.e. task based execution models.

Why do I believe in the promise of task based systems

- Consider the following two examples
  - OpenCL matrix multiplication
  - Linear Algebra expressed as a DAG of tasks

Blocked matrix multiply: kernel

```
#define blksz 16
__kernel void mmul(
                const unsigned int N,
                __global float* A,
                __global float* B,
                __global float* C,
                __local  float* Awrk,
                __local  float* Bwrk)
{
   int kloc, Kblk;
   float Ctmp=0.0f;

   //  compute element C(i,j)
   int i = get_global_id(0);
   int j = get_global_id(1);

   // Element C(i,j) is in block C(Iblk,Jblk)
   int Iblk = get_group_id(0);
   int Jblk = get_group_id(1);

   // C(i,j) is element C(iloc, jloc)
   //  of block C(Iblk, Jblk)
   int iloc = get_local_id(0);
   int jloc = get_local_id(1);
   int Num_BLK = N/blksz;

 // upper-left-corner and inc for A and B
 int Abase = Iblk*N*blksz;        int Ainc  = blksz;
 int Bbase = Jblk*blksz;    int Binc  = blksz*N;

// C(Iblk,Jblk) = (sum over Kblk) A(Iblk,Kblk)*B(Kblk,Jblk)
 for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
 {
    //Load A(Iblk,Kblk) and B(Kblk,Jblk).
    //Each work-item loads a single element of the two
    //blocks which are shared with the entire work-group

    Awrk[iloc*blksz+jloc] = A[Abase+iloc*N+jloc];
    Bwrk[iloc*blksz+jloc] = B[Bbase+iloc*N+jloc];

    barrier(CLK_LOCAL_MEM_FENCE);

    #pragma unroll
    for(kloc=0; kloc<blksz; kloc++)
       Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

    barrier(CLK_LOCAL_MEM_FENCE);
    Abase += Ainc;       Bbase += Binc;
  }
  C[j*N+i] = Ctmp;
}
```

Matrix multiplication … Portable Performance (in MFLOPS)

Case	CPU	Xeon Phi	Core i7, HD Graphics	NVIDIA Tesla
Sequential C (compiled /O3)	224.4		1221.5	
C(i,j) per work-item, all global	841.5	13591		3721
C row per work-item, all global	869.1	4418		4196
C row per work-item, A row private	1038.4	24403		8584
C row per work-item, A private, B local	3984.2	5041		8182
Block oriented approach using local (blksz=16)	12271.3	74051 (126322*)	38348 (53687*)	119305
Block oriented approach using local (blksz=32)	16268.8			

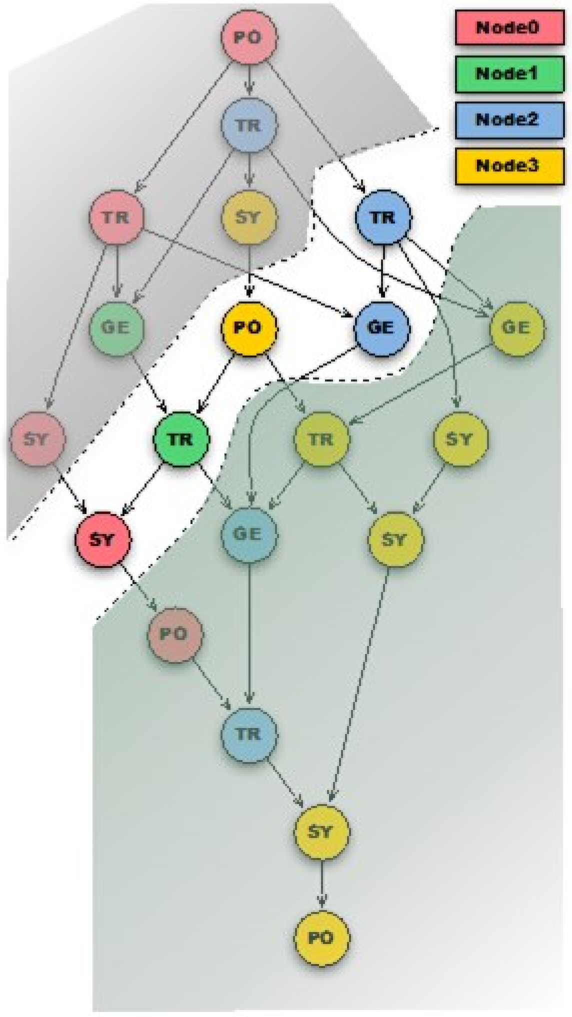
Xeon Phi SE10P, CL_CONFIG_MIC_DEVICE_2MB_POOL_INIT_SIZE_MB = 4 MB
* The comp was run twice and only the second time is reported (hides cost of memory movement.

Intel® Core™ i5-2520M CPU @2.5 GHz (dual core) Windows 7 64 bit OS, Intel compiler 64 bit version 13.1.1.171, OpenCL SDK 2013, MKL 11.0 update 3.

Intel Core i7-4850HQ @ 2.3 GHz which has an Intel HD Graphics 5200 w/ high speed memory. ICC 2013 sp1 update 2.
Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs

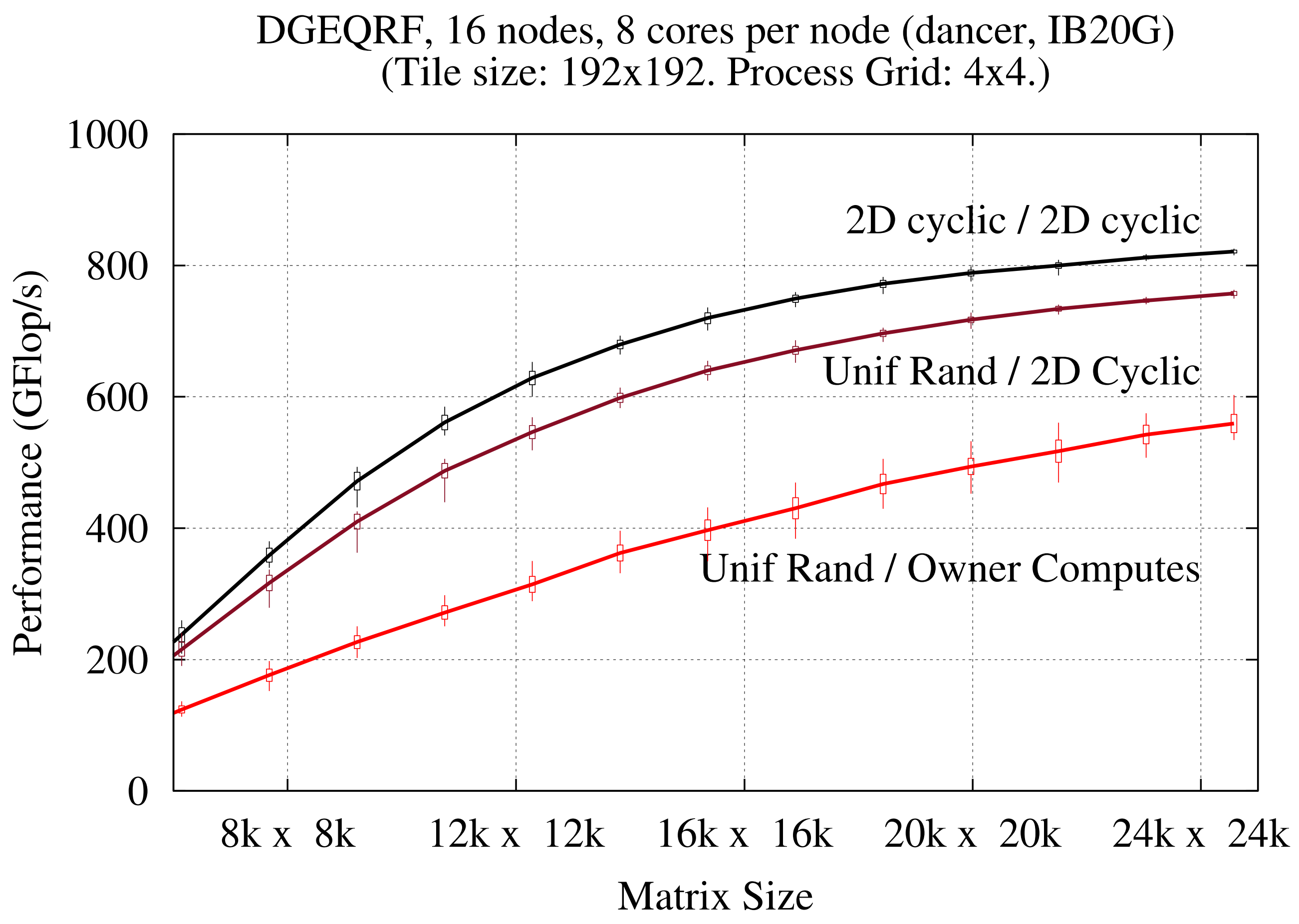
Task Based Algorithms & Runtime

http://icl.utk.edu/parsec



- Dataflow Scheduling Engine
- Distributed
  - Task Placement through data affinity
  - *Allows to run the algorithm on any data distribution*
    - (use 2D block cyclic for perf.)
- NUMA oriented
  - Favor cache reuse
  - Limit far accesses
- Manycore & Accelerators
  - Tasklet system
  - Automatic load balance

Data Distribution & Task Placement



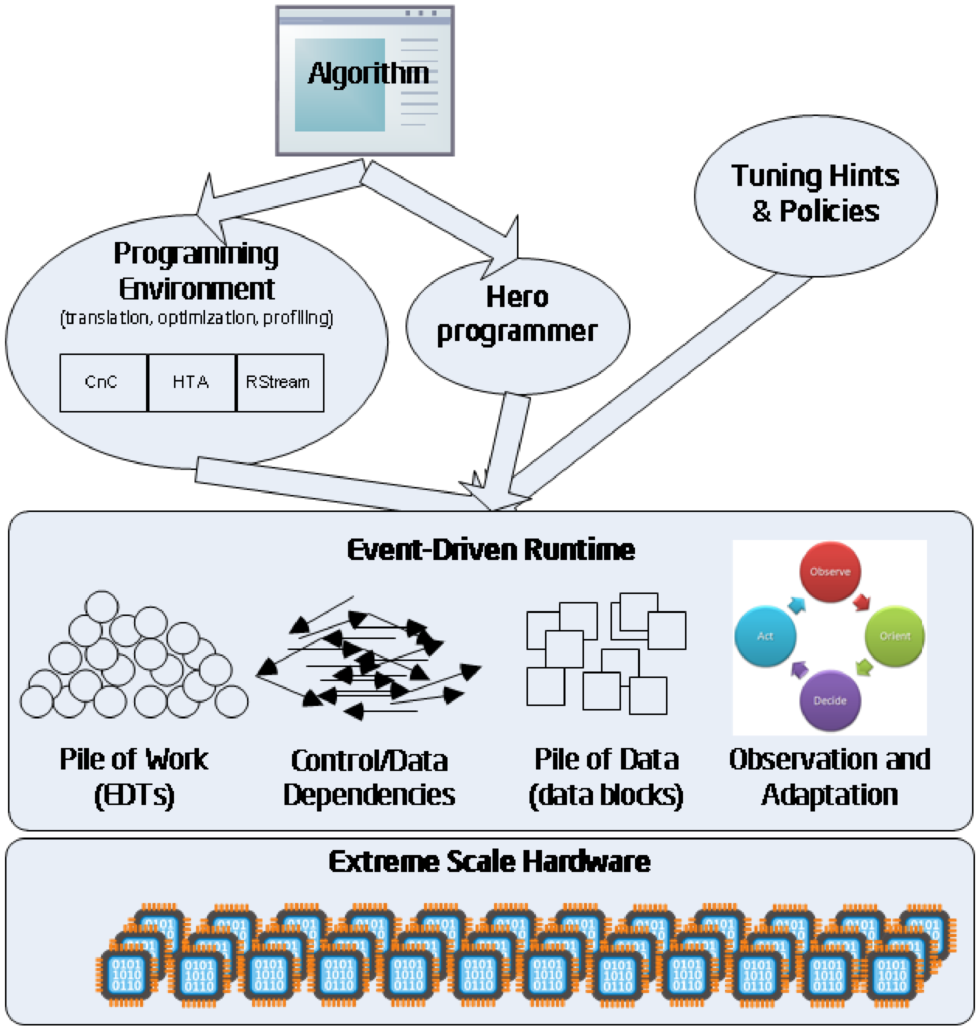
Computing Tasks placement is defined by tiles affinity
When the data distribution of tiles changes, the tasks execute on different nodes

Agenda

- Preliminaries:
  - Setting the stage for a conversation about Exascale computing.
- Methodologies:
  - Benchmarks, dirty hands, and the NIH syndrome.
- The landscape of Exascale runtimes
  - Who are we watching
- Are tasks a productive path?
  - Some suggestive but inconclusive results
- ➡ The ACR program
  - Another spelling of OCR (a task based runtime)

Example: Execution Model (ACR*)

- Fine-grained, event-driven (FGED) model with sophisticated observation
  - work broken into small tasks and relocatable data regions
  - explicit data-flow and control-flow dependencies
  - system maps work and data onto resources
  - adapts based on (static and dynamic) observations



Source: Rob Knauerhase's Cuyahoga review

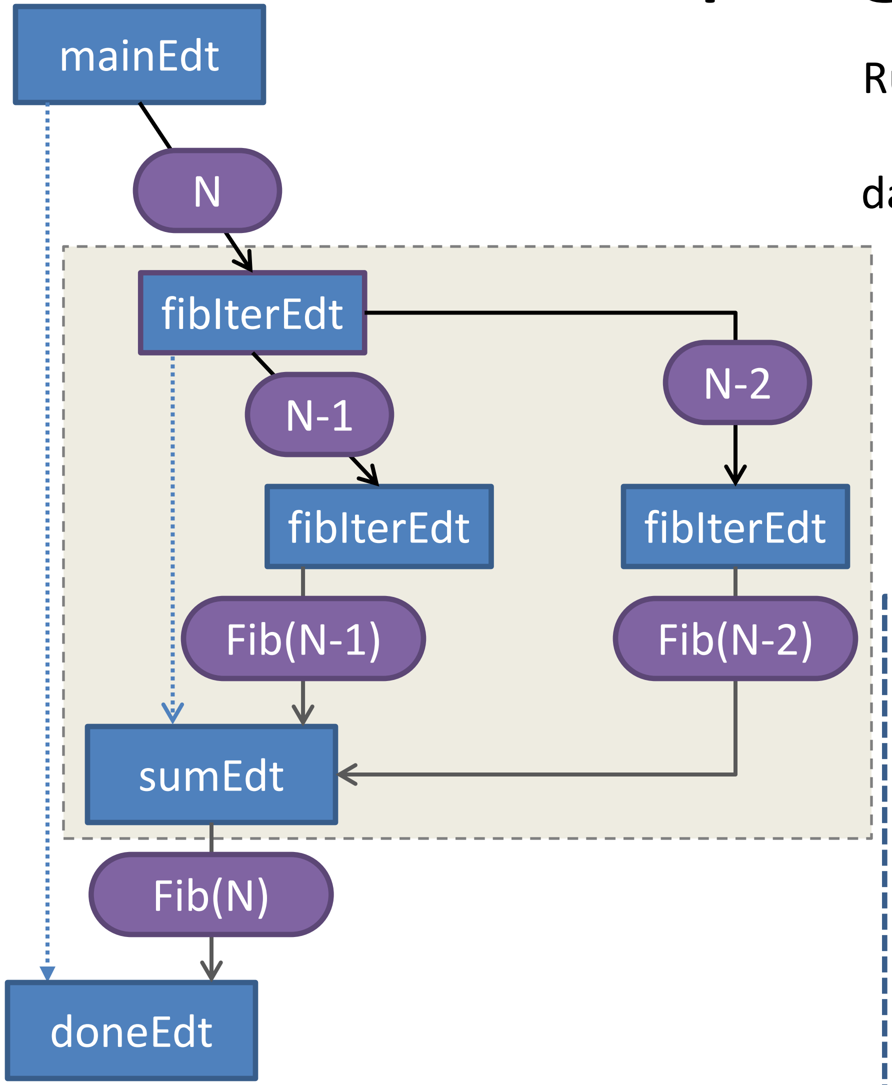
*ACR: Advanced Compute Runtime: A Rice/Intel project to add advanced locality-aware/power-aware scheduling to OCR

OCR

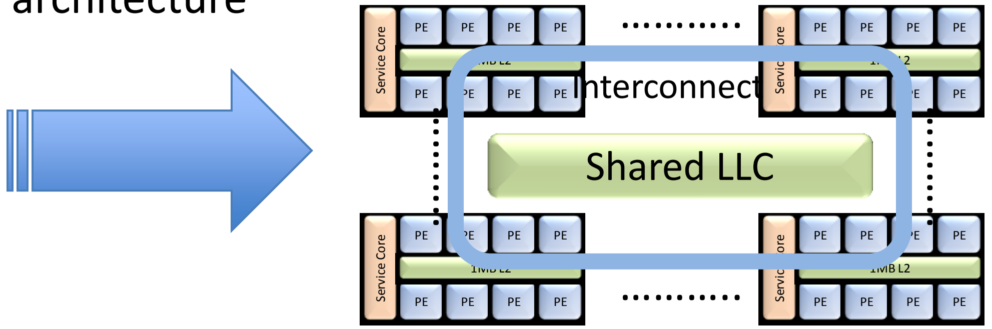
- OCR
  - **O**pen **C**ommunity **R**untime
  - Developed collaboratively with multiple partners (mainly Rice University, Reservoir Labs and Intel)

- The term ‘OCR’ is used to refer to
  - A programming model
  - A user-level API
  - A runtime framework
  - One of several reference runtime implementations

Dataflow programming model

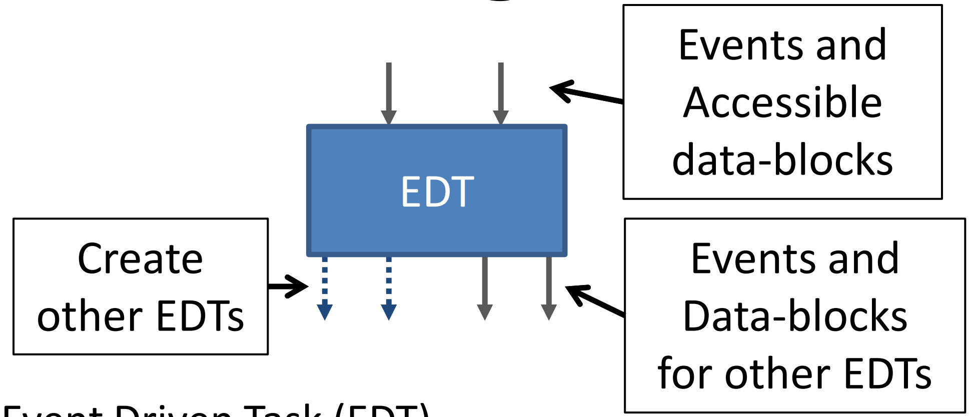


Runtime maps the constructed data-flow graph to architecture



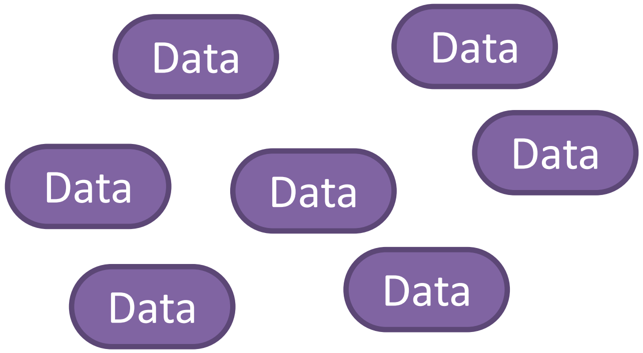
Datablock	Data shared between EDTs
EDT	A non-blocking unit of work. Runnable once all dependences are satisfied.
┈┈┈▶	Creation link: Source EDT creates destination
──▶	Dependence: Source EDT satisfies one of destination's dependences
━━▶	Both creation and dependence link

High level OCR concepts


Events and Accessible data-blocks
EDT
Create other EDTs
Events and Data-blocks for other EDTs

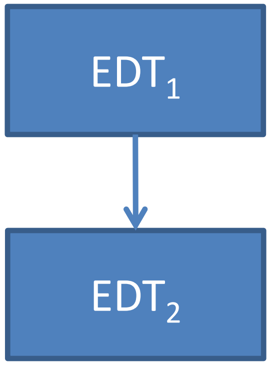
Event Driven Task (EDT)

- Uncoupled from the notion of a thread/core
- Scheduled for exeuction when all required data-blocks and dependencies have been provided
- Creates other EDTs and provides data-blocks to them



Globally visible namespace of data-blocks

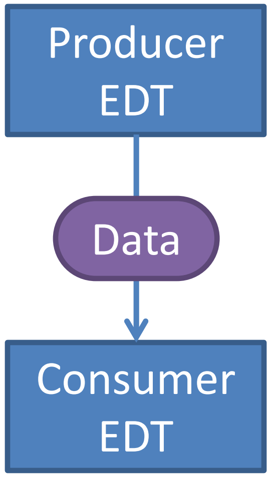
- Explicitly created and destroyed
- Only available “global” memory
- Data-blocks can move



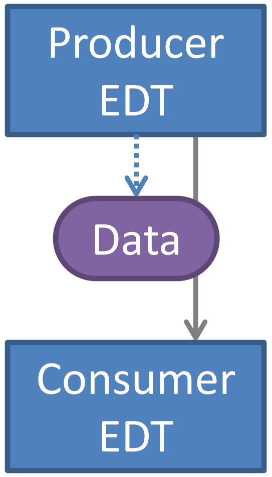
Dependences

- EDT_1 provides data to EDT_2
- EDT_1 “triggers” EDT_2 with an event
- EDTs can create other EDTs

Example 1: Producer/Consumer



Concept



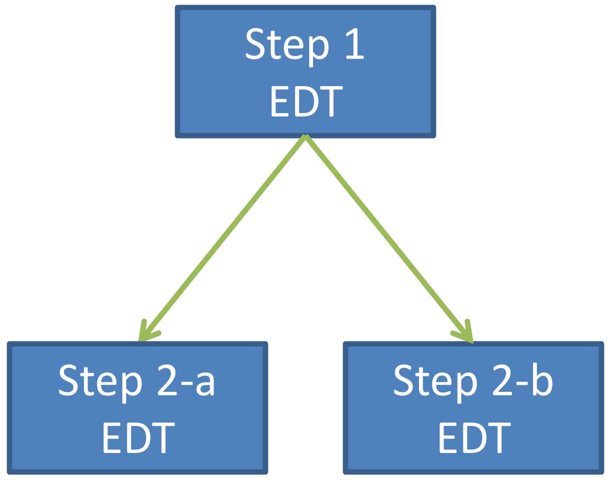
OCR

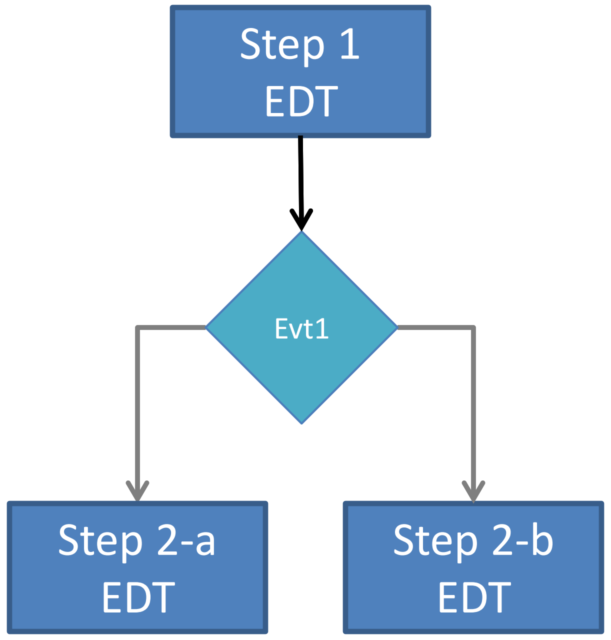
- Dynamic dependence construction
- Focus on minimum needed for placement and scheduling

OCR execution model

- Event Driven Tasks (EDTs)
  - An EDT is scheduled for execution after all its dependences are satisfied
  - The number of dependences must be known at creation time
  - Dependence satisfaction can occur in any order
  - An EDT can, during its execution:
    - Create other EDTs and data-blocks (DBs)
    - Manipulate the dependence graph for future (not ready) EDTs
    - Access stack and ephemeral local heap, but NO global memory other than the data-blocks.
    - Access data-blocks passed in as a dependence or created by the EDT
  - An EDT cannot block during its execution
- Data Blocks (DBs)
  - Contiguous block of global memory visible to any EDT

Example 2a: Simple synchronization





Concept

OCR

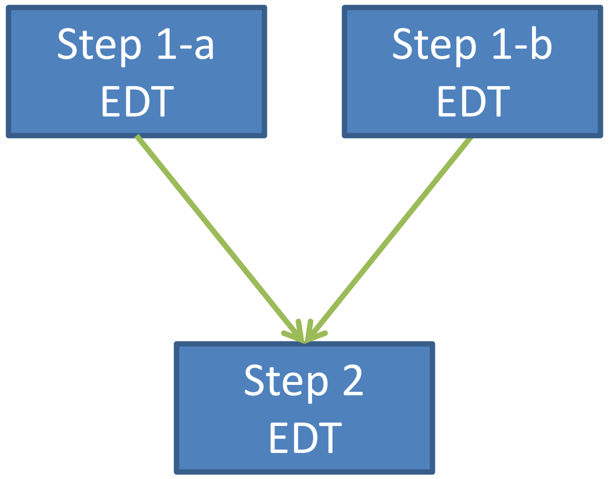
- Steps 1, 2-a, and 2-b need not know about each others' existence – they may all have been created by another EDT

Events

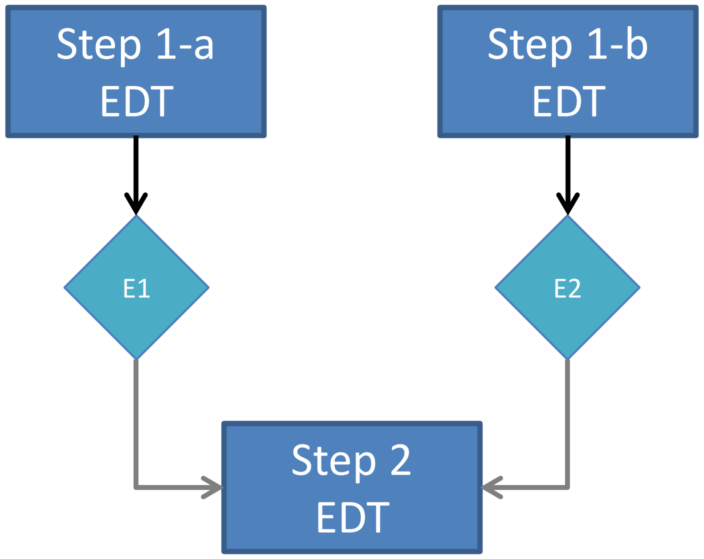
- Events used to:
  - Satisfy one or more of an EDT’s dependents
  - Dynamically change the flow graph

- Events capture the concepts of:
  - Data dependence: data-blocks “flow” along the edges
  - Pure control dependence

Example 2b: Multiple dependences



Concept

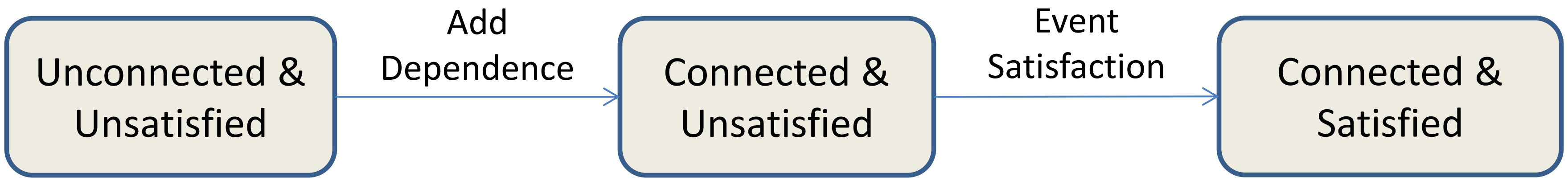


OCR

- Slots are used to order the dependences of an EDT (akin to the order of arguments in a C function)

Slots of an EDT

- Each EDT dependence has one slot assigned to it
  - Each slot can optionally receive a data-block
- Slots are initially unsatisfied; events connected to the slot propagate the "satisfied" state



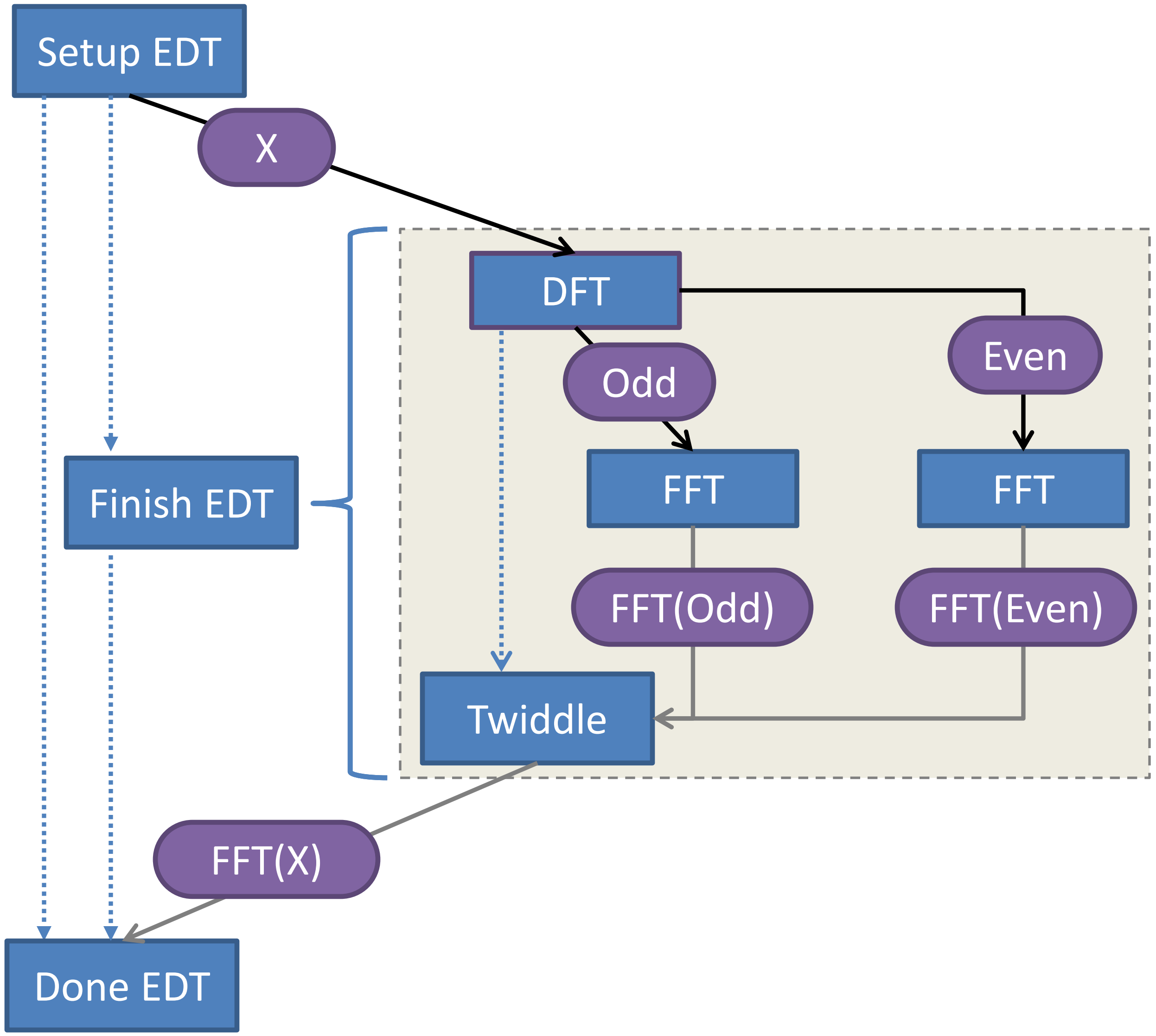
- An EDT becomes *runnable* once all of its slots are satisfied, with the *order of satisfaction* unimportant

Finish-EDT

- All EDTs have a completion event associated with them
  - The event becomes satisfied when the EDT completes and carries the data-block returned by the EDT
- A finish-EDT's completion event has a special semantic
  - The event becomes satisfied when the EDT ***and all of its children*** complete
  - The event carries no data-block
- Use cases
  - Localized barrier-like synchronization
  - Allows for an unknown number of ancestors

- Note that no EDT "waits" for completion

Example 3: FFT with a finish-EDT



Real programmers look at code

OCR Hello world

- The OCR runtime system will run a programmer's EDT called mainEdt() to start an OCR program

```
# inc lude < o c r . h>
ocrGuid_t mainEdt (
        u32 paramc , u64 paramv ,
        u32 depc , ocrEdtDep_t depv [ ] )
{
    PRINTF ( ' He l l o World ! \ n ' ) ;
    ocrShutdown ( ) ;
    return NULL_GUID;
}
```

Variables passed into the EDT func. (i.e. not OCR Data blocks)

Dependencies … available before an EDT is runnable (Data Blocks or events).

Shut down OCR (including other active EDTs … so you need to be careful when you call this).

Real programmers look at code

API cheat sheet

- EDT
  - **Task templates**: *ocrEdtTemplateCreate(), ocrEdtTemplateDestroy()*
  - **Tasks**: *ocrEdtCreate(), ocrEdtDestroy()*
- DBs
  - **Datablock management**: *ocrDbCreate(), ocrDbDestroy()*
  - **Datablock usage**: *ocrDbRelease()*
- Events
  - **Event management**: *ocrEventCreate(), ocrEventDestroy()*
  - **Event satisfaction**: *ocrEventSatisfy()*
  - **Dependence definition**: *ocrAddDependence()*
- Miscellaneous
  - **Entry point of OCR**: *mainEdt()*
  - **Shutdown**: *ocrShutdown()*

Producer-Consumer: mainEDT

Real programmers look at code

```
# inc lude < o c r . h>
ocrGuid_t mainEdt ( parameters , dependences ) {
    ocrGuid_t  t1, t2, t3, edt1, edt2, edt3;
    ocrGuid_t  outProdDB, outConsEvt;

    ocrEdtTemplateCreate( &t1, prod(), …);
    ocrEdtTemplateCreate( &t2, cons(), …);
    ocrEdtTemplateCreate( &t3,  end(), …);

    ocrEdtCreate(&edt1, t1,…stuff …, &outProdDB);
    ocrEdtCreate(&edt2, t2, …stuff …, &outConsEvt);
    ocrEdtCreate(&edt3, t3, …stuff …, NULL);

    ocrAddDependence(&outProdDB,  edt2, …stuff);
    ocrAddDependence(&outConsEvt,  edt2, …stuff);

     ocrAddDependence(NULL,  edt1, …stuff);

     // clean up code to release resources .. Not shown
     return NULL_GUID:
}
```

All OCR objects referenced by a Global Unique ID (GUID)

Templates for EDT creation … to connect function to EDT and define patterns of parameters and dependences

Create the actual EDTs … output GUIDs connected to post-slot/return from EDT function.

Dependences created explicitly and dynamically … maybe a bit verbose but the flexibility is empowering!

Trigger the first EDT to start the computation

Producer-Consumer: EDT functions

Real programmers look at code

```
# inc lude < o c r . h>
ocrGuid_t prod ( ……. ocrEdtDep_t dep[]) {
    int k;  ocrGuid_t  db1;
    ocrDbCreate(&db1, (void**)&k, sizeof(int) …);
    k[0] = 42;
    return db1;
}
ocrGuid_t cons ( ……. ocrEdtDep_t dep[]) {
    int data =(int*)dep[0].ptr;
    PRINTF{" I consumed %lu\n", *data);
    return  dep[0].guid;
}
ocrGuid_t end ( ……. ocrEdtDep_t dep[]) {
     ocrDbDestroy{dep[0].guid);
     ocrShutdown();
     return NULL_GUID;
}
```

Create a data block to hold a single int

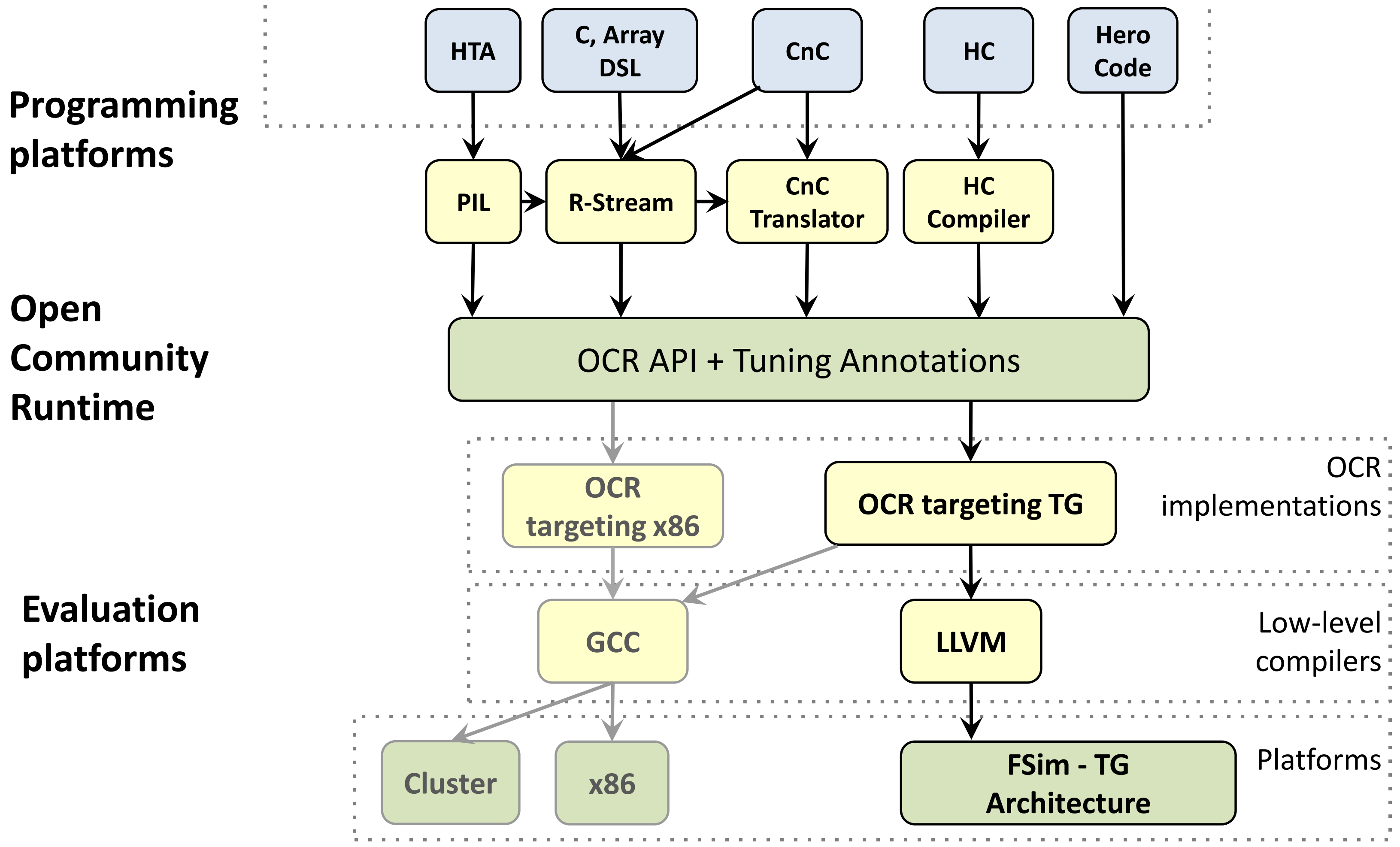
Return an event bound to the data block. This is used to trigger other EDTs

Access the contents of a data block

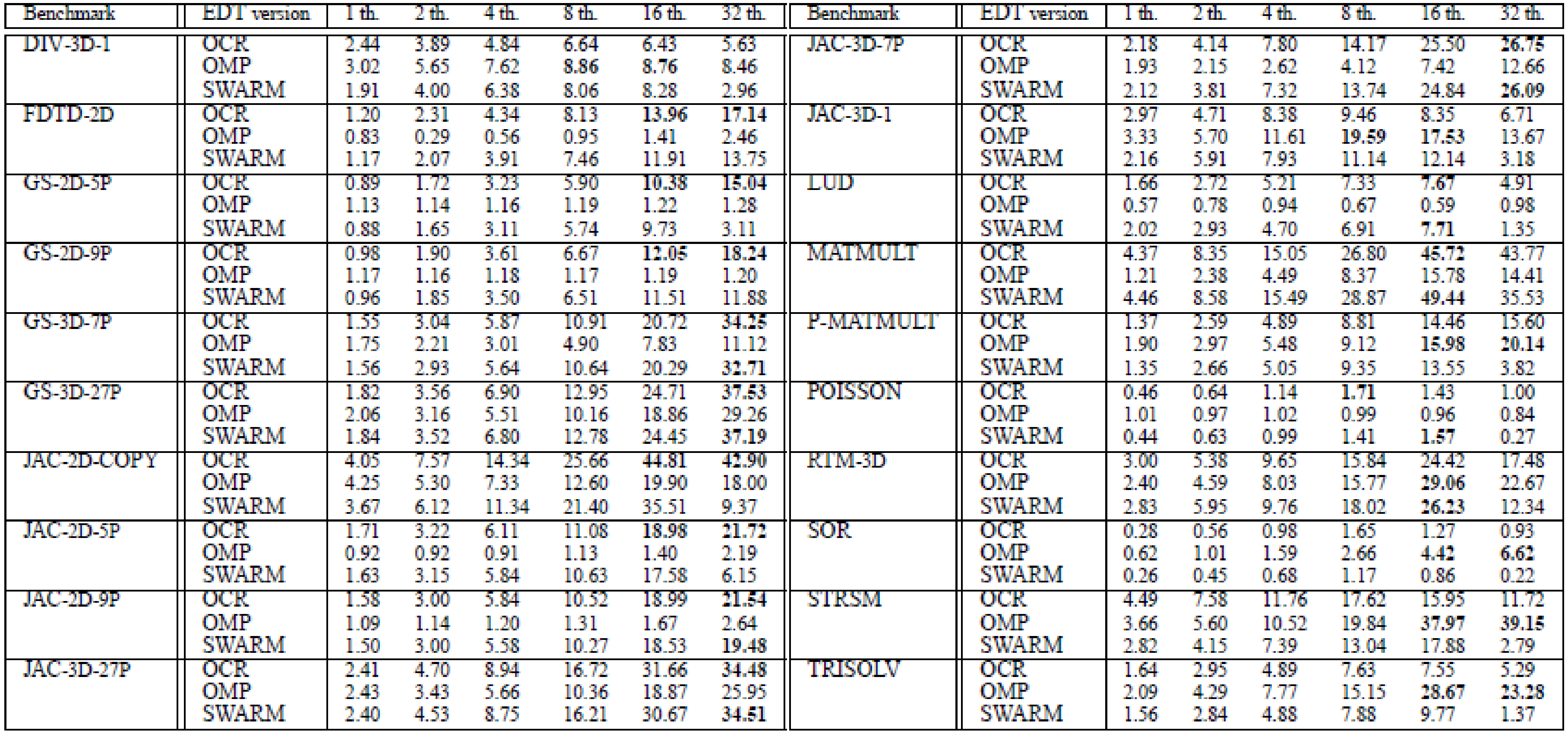
Return an event bound to the guid of the data block.

Clean up data blocks in memory and shutdown OCR.

OCR ecosystem



Preliminary results

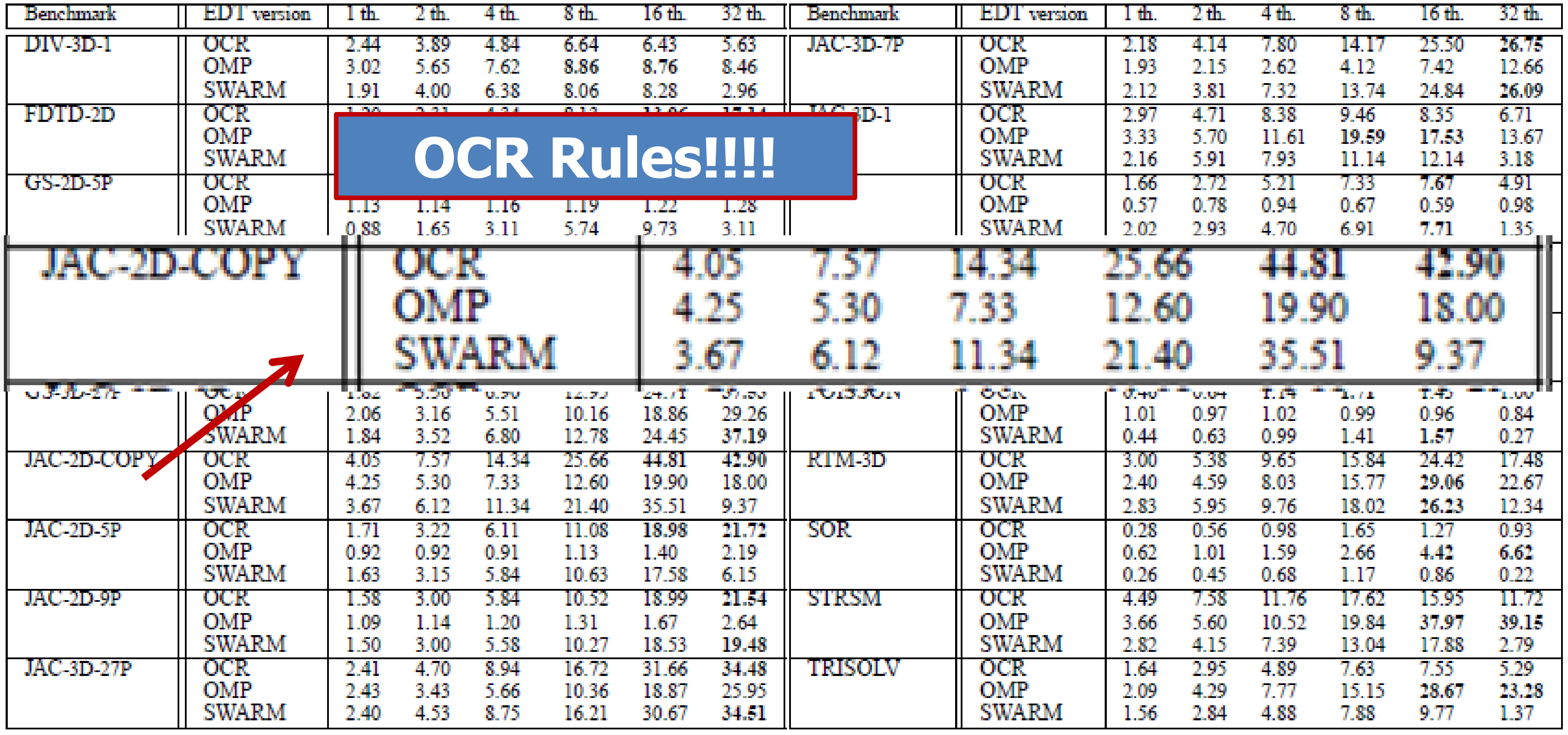


Benchmark	EDT version	1 th.	2 th.	4 th.	8 th.	16 th.	32 th.
DIV-3D-1	OCR	2.44	3.89	4.84	6.64	6.43	5.63
	OMP	3.02	5.65	7.62	**8.86**	**8.76**	**8.46**
	SWARM	1.91	4.00	6.38	8.06	8.28	2.96
FDTD-2D	OCR	1.20	2.31	4.34	8.13	**13.96**	**17.14**
	OMP	0.83	0.29	0.56	0.95	1.41	2.46
	SWARM	1.17	2.07	3.91	7.46	11.91	13.75
GS-2D-5P	OCR	0.89	1.72	3.23	5.90	**10.38**	**15.04**
	OMP	1.13	1.14	1.16	1.19	1.22	1.28
	SWARM	0.88	1.65	3.11	5.74	9.73	3.11
GS-2D-9P	OCR	0.98	1.90	3.61	6.67	**12.05**	**18.24**
	OMP	1.17	1.16	1.18	1.17	1.19	1.20
	SWARM	0.96	1.85	3.50	6.51	11.51	11.88
GS-3D-7P	OCR	1.55	3.04	5.87	10.91	20.72	**34.25**
	OMP	1.75	2.21	3.01	4.90	7.83	11.12
	SWARM	1.56	2.93	5.64	10.64	20.29	**32.71**
GS-3D-27P	OCR	1.82	3.56	6.90	12.95	24.71	**37.53**
	OMP	2.06	3.16	5.51	10.16	18.86	29.26
	SWARM	1.84	3.52	6.80	12.78	24.45	**37.19**
JAC-2D-COPY	OCR	4.05	7.57	14.34	25.66	**44.81**	**42.90**
	OMP	4.25	5.30	7.33	12.60	19.90	18.00
	SWARM	3.67	6.12	11.34	21.40	35.51	9.37
JAC-2D-5P	OCR	1.71	3.22	6.11	11.08	**18.98**	**21.72**
	OMP	0.92	0.92	0.91	1.13	1.40	2.19
	SWARM	1.63	3.15	5.84	10.63	17.58	6.15
JAC-2D-9P	OCR	1.58	3.00	5.84	10.52	18.99	**21.54**
	OMP	1.09	1.14	1.20	1.31	1.67	2.64
	SWARM	1.50	3.00	5.58	10.27	18.53	**19.48**
JAC-3D-27P	OCR	2.41	4.70	8.94	16.72	31.66	34.48
	OMP	2.43	3.43	5.66	10.36	18.87	25.95
	SWARM	2.40	4.53	8.75	16.21	30.67	**34.51**
JAC-3D-7P	OCR	2.18	4.14	7.80	14.17	25.50	**26.75**
	OMP	1.93	2.15	2.62	4.12	7.42	12.66
	SWARM	2.12	3.81	7.32	13.74	24.84	**26.09**
JAC-3D-1	OCR	2.97	4.71	8.38	9.46	8.35	6.71
	OMP	3.33	5.70	11.61	**19.59**	**17.53**	13.67
	SWARM	2.16	5.91	7.93	11.14	12.14	3.18
LUD	OCR	1.66	2.72	5.21	7.33	**7.67**	4.91
	OMP	0.57	0.78	0.94	0.67	0.59	0.98
	SWARM	2.02	2.93	4.70	6.91	**7.71**	1.35
MATMULT	OCR	4.37	8.35	15.05	26.80	**45.72**	43.77
	OMP	1.21	2.38	4.49	8.37	15.78	14.41
	SWARM	4.46	8.58	15.49	28.87	**49.44**	35.53
P-MATMULT	OCR	1.37	2.59	4.89	8.81	14.46	15.60
	OMP	1.90	2.97	5.48	9.12	**15.98**	**20.14**
	SWARM	1.35	2.66	5.05	9.35	13.55	3.82
POISSON	OCR	0.46	0.64	1.14	**1.71**	1.43	1.00
	OMP	1.01	0.97	1.02	0.99	0.96	0.84
	SWARM	0.44	0.63	0.99	1.41	**1.57**	0.27
RTM-3D	OCR	3.00	5.38	9.65	15.84	24.42	17.48
	OMP	2.40	4.59	8.03	15.77	**29.06**	22.67
	SWARM	2.83	5.95	9.76	18.02	**26.23**	12.34
SOR	OCR	0.28	0.56	0.98	1.65	1.27	0.93
	OMP	0.62	1.01	1.59	2.66	**4.42**	**6.62**
	SWARM	0.26	0.45	0.68	1.17	0.86	0.22
STRSM	OCR	4.49	7.58	11.76	17.62	15.95	11.72
	OMP	3.66	5.60	10.52	19.84	**37.97**	**39.15**
	SWARM	2.82	4.15	7.39	13.04	17.88	2.79
TRISOLV	OCR	1.64	2.95	4.89	7.63	7.55	5.29
	OMP	2.09	4.29	7.77	15.15	**28.67**	**23.28**
	SWARM	1.56	2.84	4.88	7.88	9.77	1.37

Table 4. SWARM, OCR and OpenMP performance in Gflops/s

- On some code, OCR matches or bests OMP
- Simple scheduler, no data-blocks (very preliminary but promising)

Preliminary results



OCR Rules!!!!

JAC-2D-COPY	OCR	4.05	7.57	14.34	25.66	44.81	42.90
	OMP	4.25	5.30	7.33	12.60	19.90	18.00
	SWARM	3.67	6.12	11.34	21.40	35.51	9.37

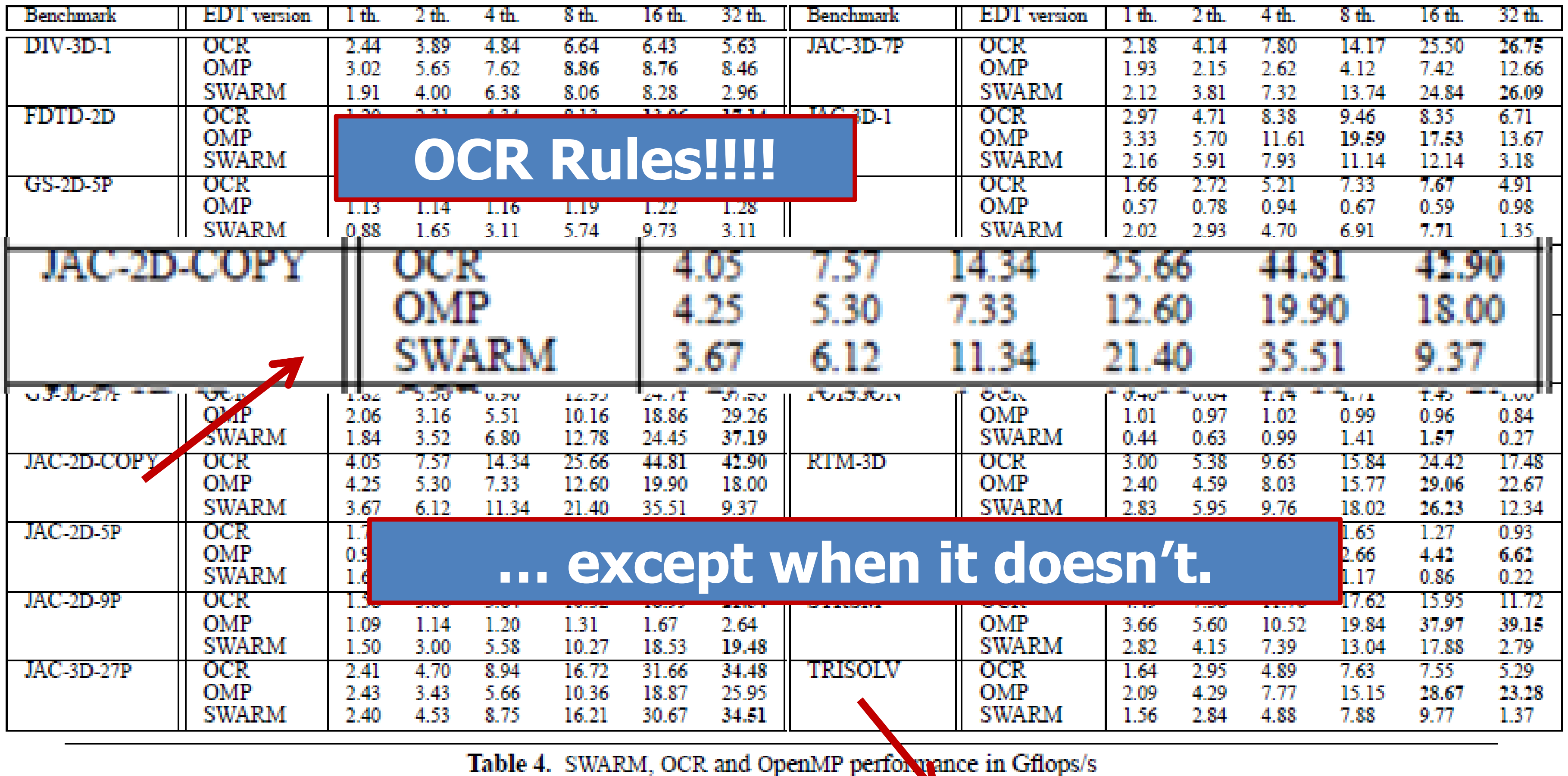
Benchmark	EDT version	1 th.	2 th.	4 th.	8 th.	16 th.	32 th.
DIV-3D-1	OCR	2.44	3.89	4.84	6.64	6.43	5.63
	OMP	3.02	5.65	7.62	8.86	8.76	8.46
	SWARM	1.91	4.00	6.38	8.06	8.28	2.96
FDTD-2D	OCR	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	OMP	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	SWARM	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
GS-2D-5P	OCR	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	OMP	1.13	1.14	1.16	1.19	1.22	1.28
	SWARM	0.88	1.65	3.11	5.74	9.73	3.11
[illegible]	OCR	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	OMP	2.06	3.16	5.51	10.16	18.86	29.26
	SWARM	1.84	3.52	6.80	12.78	24.45	37.19
JAC-2D-COPY	OCR	4.05	7.57	14.34	25.66	44.81	42.90
	OMP	4.25	5.30	7.33	12.60	19.90	18.00
	SWARM	3.67	6.12	11.34	21.40	35.51	9.37
JAC-2D-5P	OCR	1.71	3.22	6.11	11.08	18.98	21.72
	OMP	0.92	0.92	0.91	1.13	1.40	2.19
	SWARM	1.63	3.15	5.84	10.63	17.58	6.15
JAC-2D-9P	OCR	1.58	3.00	5.84	10.52	18.99	21.54
	OMP	1.09	1.14	1.20	1.31	1.67	2.64
	SWARM	1.50	3.00	5.58	10.27	18.53	19.48
JAC-3D-27P	OCR	2.41	4.70	8.94	16.72	31.66	34.48
	OMP	2.43	3.43	5.66	10.36	18.87	25.95
	SWARM	2.40	4.53	8.75	16.21	30.67	34.51

Benchmark	EDT version	1 th.	2 th.	4 th.	8 th.	16 th.	32 th.
JAC-3D-7P	OCR	2.18	4.14	7.80	14.17	25.50	26.75
	OMP	1.93	2.15	2.62	4.12	7.42	12.66
	SWARM	2.12	3.81	7.32	13.74	24.84	26.09
JAC-3D-1	OCR	2.97	4.71	8.38	9.46	8.35	6.71
	OMP	3.33	5.70	11.61	19.59	17.53	13.67
	SWARM	2.16	5.91	7.93	11.14	12.14	3.18
[illegible]	OCR	1.66	2.72	5.21	7.33	7.67	4.91
	OMP	0.57	0.78	0.94	0.67	0.59	0.98
	SWARM	2.02	2.93	4.70	6.91	7.71	1.35
POISSON	OCR	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	OMP	1.01	0.97	1.02	0.99	0.96	0.84
	SWARM	0.44	0.63	0.99	1.41	1.57	0.27
RTM-3D	OCR	3.00	5.38	9.65	15.84	24.42	17.48
	OMP	2.40	4.59	8.03	15.77	29.06	22.67
	SWARM	2.83	5.95	9.76	18.02	26.23	12.34
SOR	OCR	0.28	0.56	0.98	1.65	1.27	0.93
	OMP	0.62	1.01	1.59	2.66	4.42	6.62
	SWARM	0.26	0.45	0.68	1.17	0.86	0.22
STRSM	OCR	4.49	7.58	11.76	17.62	15.95	11.72
	OMP	3.66	5.60	10.52	19.84	37.97	39.15
	SWARM	2.82	4.15	7.39	13.04	17.88	2.79
TRISOLV	OCR	1.64	2.95	4.89	7.63	7.55	5.29
	OMP	2.09	4.29	7.77	15.15	28.67	23.28
	SWARM	1.56	2.84	4.88	7.88	9.77	1.37

Table 4. SWARM, OCR and OpenMP performance in Gflops/s

- On some code, OCR matches or bests OMP
- Simple scheduler, no data-blocks (very preliminary but promising)

Preliminary results

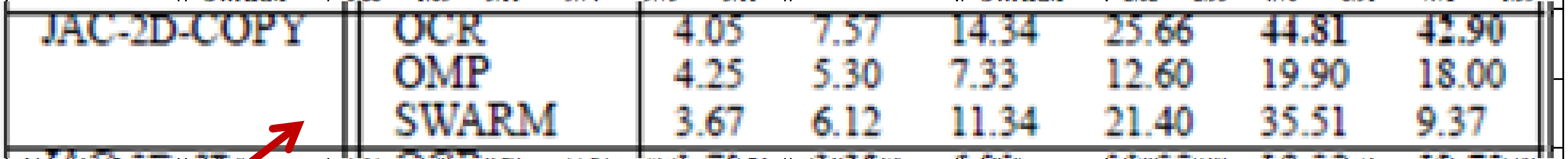


Benchmark	EDT version	1 th.	2 th.	4 th.	8 th.	16 th.	32 th.	Benchmark	EDT version	1 th.	2 th.	4 th.	8 th.	16 th.	32 th.
DIV-3D-1	OCR	2.44	3.89	4.84	6.64	6.43	5.63	JAC-3D-7P	OCR	2.18	4.14	7.80	14.17	25.50	**26.75**
	OMP	3.02	5.65	7.62	8.86	8.76	8.46		OMP	1.93	2.15	2.62	4.12	7.42	12.66
	SWARM	1.91	4.00	6.38	8.06	8.28	2.96		SWARM	2.12	3.81	7.32	13.74	24.84	**26.09**
FDTD-2D	OCR	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]D-1	OCR	2.97	4.71	8.38	9.46	8.35	6.71
	OMP	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]		OMP	3.33	5.70	11.61	**19.59**	**17.53**	13.67
	SWARM	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]		SWARM	2.16	5.91	7.93	11.14	12.14	3.18
GS-2D-5P	OCR	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	OCR	1.66	2.72	5.21	7.33	**7.67**	4.91
	OMP	1.13	1.14	1.16	1.19	1.22	1.28		OMP	0.57	0.78	0.94	0.67	0.59	0.98
	SWARM	0.88	1.65	3.11	5.74	9.73	3.11		SWARM	2.02	2.93	4.70	6.91	**7.71**	1.35
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	OMP	2.06	3.16	5.51	10.16	18.86	29.26		OMP	1.01	0.97	1.02	0.99	0.96	0.84
	SWARM	1.84	3.52	6.80	12.78	24.45	37.19		SWARM	0.44	0.63	0.99	1.41	**1.57**	0.27
JAC-2D-COPY	OCR	4.05	7.57	14.34	25.66	**44.81**	**42.90**	RTM-3D	OCR	3.00	5.38	9.65	15.84	24.42	17.48
	OMP	4.25	5.30	7.33	12.60	19.90	18.00		OMP	2.40	4.59	8.03	15.77	**29.06**	22.67
	SWARM	3.67	6.12	11.34	21.40	35.51	9.37		SWARM	2.83	5.95	9.76	18.02	**26.23**	12.34
JAC-2D-5P	OCR	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	OCR	[illegible]	[illegible]	[illegible]	1.65	1.27	0.93
	OMP	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]		OMP	[illegible]	[illegible]	[illegible]	2.66	**4.42**	**6.62**
	SWARM	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]		SWARM	[illegible]	[illegible]	[illegible]	1.17	0.86	0.22
JAC-2D-9P	OCR	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	OCR	[illegible]	[illegible]	[illegible]	17.62	15.95	11.72
	OMP	1.09	1.14	1.20	1.31	1.67	2.64		OMP	3.66	5.60	10.52	19.84	**37.97**	**39.15**
	SWARM	1.50	3.00	5.58	10.27	18.53	19.48		SWARM	2.82	4.15	7.39	13.04	17.88	2.79
JAC-3D-27P	OCR	2.41	4.70	8.94	16.72	31.66	34.48	TRISOLV	OCR	1.64	2.95	4.89	7.63	7.55	5.29
	OMP	2.43	3.43	5.66	10.36	18.87	25.95		OMP	2.09	4.29	7.77	15.15	**28.67**	**23.28**
	SWARM	2.40	4.53	8.75	16.21	30.67	**34.51**		SWARM	1.56	2.84	4.88	7.88	9.77	1.37

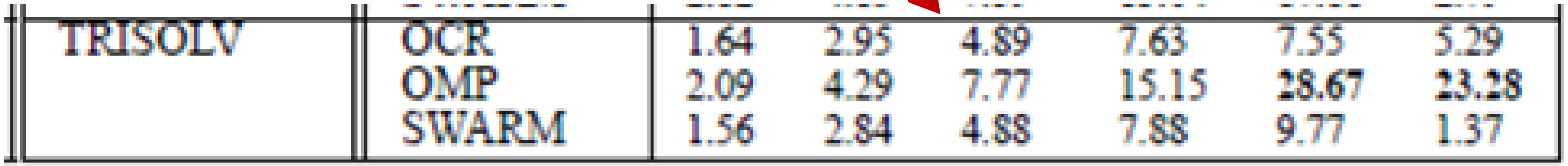
Table 4. SWARM, OCR and OpenMP performance in Gflops/s

OCR Rules!!!!

… except when it doesn't.



JAC-2D-COPY	OCR	4.05	7.57	14.34	25.66	**44.81**	**42.90**
	OMP	4.25	5.30	7.33	12.60	19.90	18.00
	SWARM	3.67	6.12	11.34	21.40	35.51	9.37



TRISOLV	OCR	1.64	2.95	4.89	7.63	7.55	5.29
	OMP	2.09	4.29	7.77	15.15	**28.67**	**23.28**
	SWARM	1.56	2.84	4.88	7.88	9.77	1.37

OCR Summary

- OCR is not about what it is but what it isn't.
  - OCR is one possible result when one deletes every concept that isn't exascale-worthy.
- Exascale needs data encapsulation:
  - deprecate heap -> use (relocatable) datablocks.
- Exascale needs encapsulation:
  - deprecate procedural flow -> event-driven (restartable?) tasks
- OCR might be adequate as an exascale runtime but it's unclear how to map applications to OCR.
  - HPC users want look-and-feel of MPI.

So is OCR the future of extreme scalability?

- OCR is Great!
  - OCR is a great test-bed as we work out the details of how to make task based systems work.
  - OCR is a productive research vehicle for our collaboration with Rice.
  - Uncouples **tasks** from **UEs** and **Data** from **Memory** so we can experiment with those features and how they help us with reliability, load balancing and Performance/watt optimization.
- OCR has "issues"
  - It does not expose a platform model and therefore lacks abstractions for programmers to manage locality.
  - Data blocks do not provide functionality needed to support collectives.
  - OCR cannot express data parallelism
  - OCR codes aren't modular; the simplest example of that is iterations. You cannot just take an OCR code segment and put it in a loop. More ominously, if you change anything in your task graph of dependencies, you have to reconnect all the "ducts."

The challenge that scares me: Algorithms

- Exascale algorithms can not depend on checkpoint restart.
  - Silent Errors … you'll get them and not even know it.
  - Checkpoint is massive data motion, which is to opposite of exa-style
- Need algorithms that make progress and converge to the right answer even when faults occurs.
  - Many machine learning algorithms map onto a master-less map-reduce pattern and can tolerate faults.
  - Some classes of linear algebra algorithms can progress around faults if subsets of the computation can be made reliable (by replicating tasks).
  - Stochastic algorithms
- Research question:
  - Can we find fault resilient algorithms for the problems we care about for exascale systems?

Conclusion/Next steps

- We know the problems to solve … and OCR is a good research vehicle at this time.
- But we have much work to do … We have not settled on a long term solution.
- Our project in PCL (the extreme scalability group) will engage in a HW/SW co-design effort:
  - Define a set of driving applications.
  - Abstract them into a small set of fundamental design patterns.
  - Access how key competing models (OCR, HPX,, CHARM++, etc) work for the above.
  - Adopt a current system and adapt it to our needs … or as a last resort create “yet another programming environment” that does solve the problem **(but it will not be a new language!)**
- Stay tuned … we will do this over the next few years! This will not be solved over night.

Case Studies:

- Vector Reduction
  - Scalability, modularity, composability & OCR
  - Source: Rob Van der Wijngaart of Intel
- FFT
  - Divide and conquer with OCR
  - Source: Univ. OR undergrad project

Vector reduction, an example

- Objective (Parallel Research Kernel *reduce*):
  - Compute element-wise sum of large collection (N) of vector pairs
  - Do this using various methods to demonstrate performance implications
- Assumption:
  - Vector pairs created concurrently by different workers, spread across system, so need to be in different Data Blocks and be created in different EDTs

Three implementations, distributed memory

- Naïve serial:
  - One worker combines all vectors
- Short vectors, asymptotically optimal:
  - Workers create (binary) reduction tree
  - Each node in tree combines two complete vectors, one “local,” one “non-local”
- Long vectors, asymptotically optimal (Van de Geijn et al.):
  - Workers execute bucket reduce scatter in a number of stages
  - Workers execute MST gather to collect all reduced vector snippets at one worker

EDTs can execute concurrently, but launch is serial bottleneck

CR approach; naïve s

Master worker allocates array of N OE1 guids: not scalable

1. Create N allocation EDTs; each allocates a MB for o

Potential solutions:

- Build EDT launch trees for embarrassingly parallel sections (sigh); more scalable, but bottleneck at extreme scale
- Do not use runtime-produced guids, but let user assign explicit values (cf tags in MPI)

2. C one O nd sa

3. C N O events, sums all vect output OE3

EDTs can execute concurrently, but launch is serial bottleneck

Master worker allocates array of N OE2 guids: not scalable

wrapup EDT, which E3, then shuts down runtime

Implicit assumption: "create" done by master worker

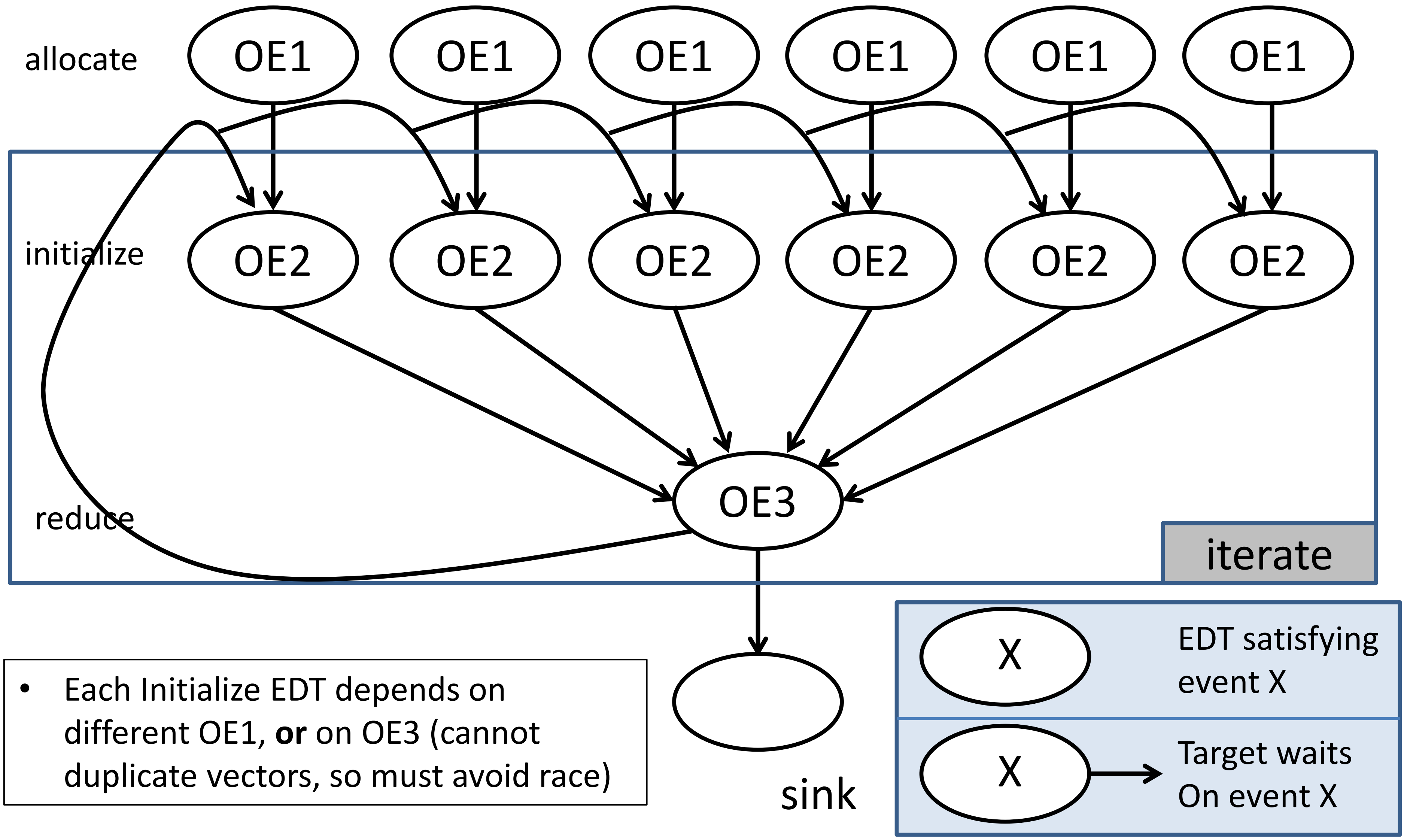
OCR approach; naïve serial

1. Create N allocation EDTs; each allocates a DB for one vector, and satisfies output event OE1
2. Create N initialization EDTs, each waits on one OE1, (re)initializes corresponding vector, and satisfies output event OE2
3. Create one summation EDT, which waits on N OE2 events, sums all vectors, satisfies output event OE3

iterate

4. Create wrapup EDT, which waits on OE3, then shuts down runtime

Graph representation



- Each Initialize EDT depends on different OE1, **or** on OE3 (cannot duplicate vectors, so must avoid race)

Observation

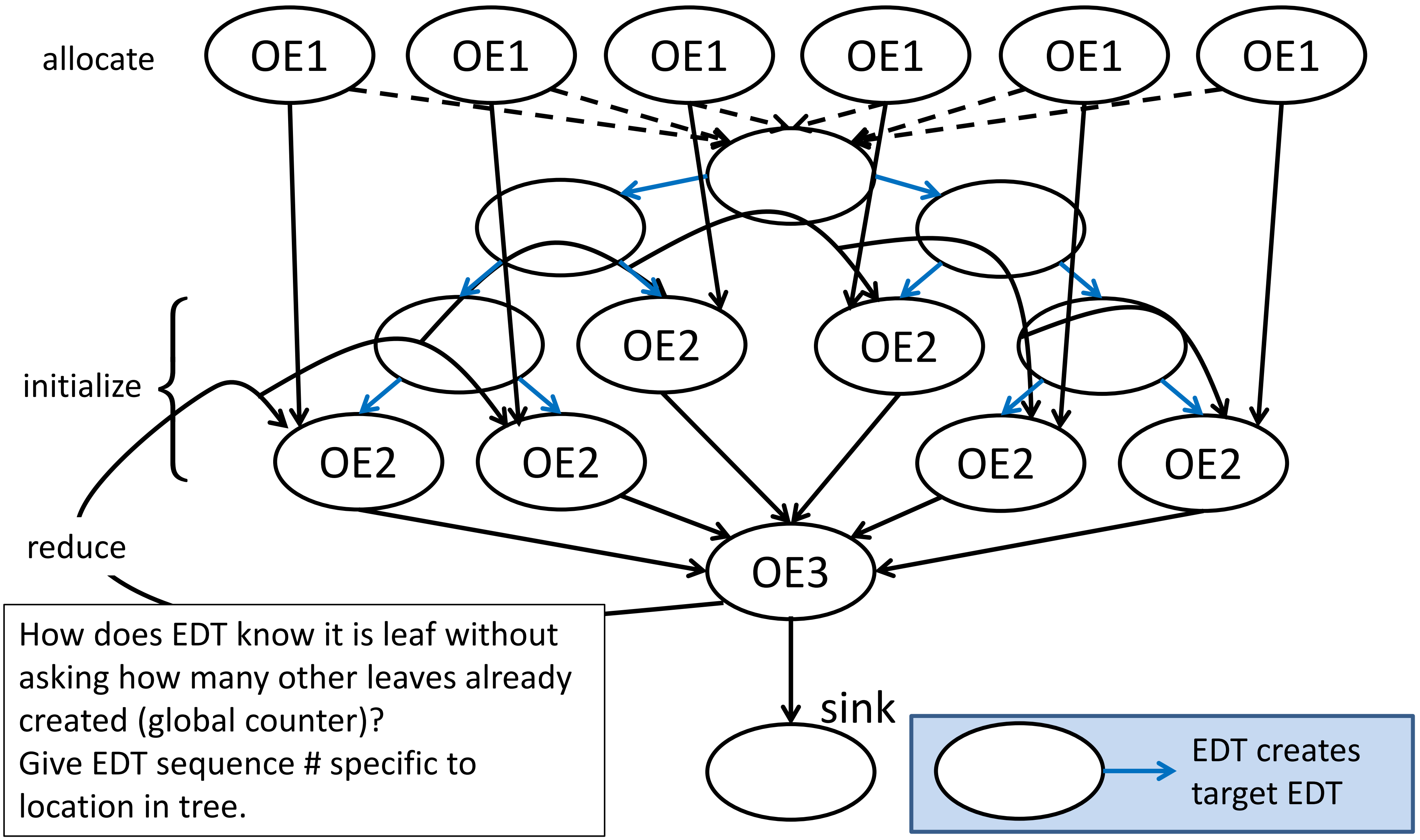
Modularity/composability:

- Do not want to or cannot inspect/change details of dependency structure of program modules
- Can wrap phases before and after loop, as well as iteration body, in Finish EDTs$\rightarrow$ unscalable fork/join style parallelism

OCR approach 2; naïve serial

1. Create N allocation EDTs, each of which allocates a DB for one vector, and satisfies output event OE1
2. Create **tree** of N intialization EDTs; each **leaf** waits on one OE1, initializes the corresponding vector, and satisfies output event OE2
3. Create one summation EDT which waits on N OE2 events, sums all the vectors, and satisfies output event OE3
4. Create a wrapup EDT which waits on OE3 and then shuts down the runtime

Graph representation

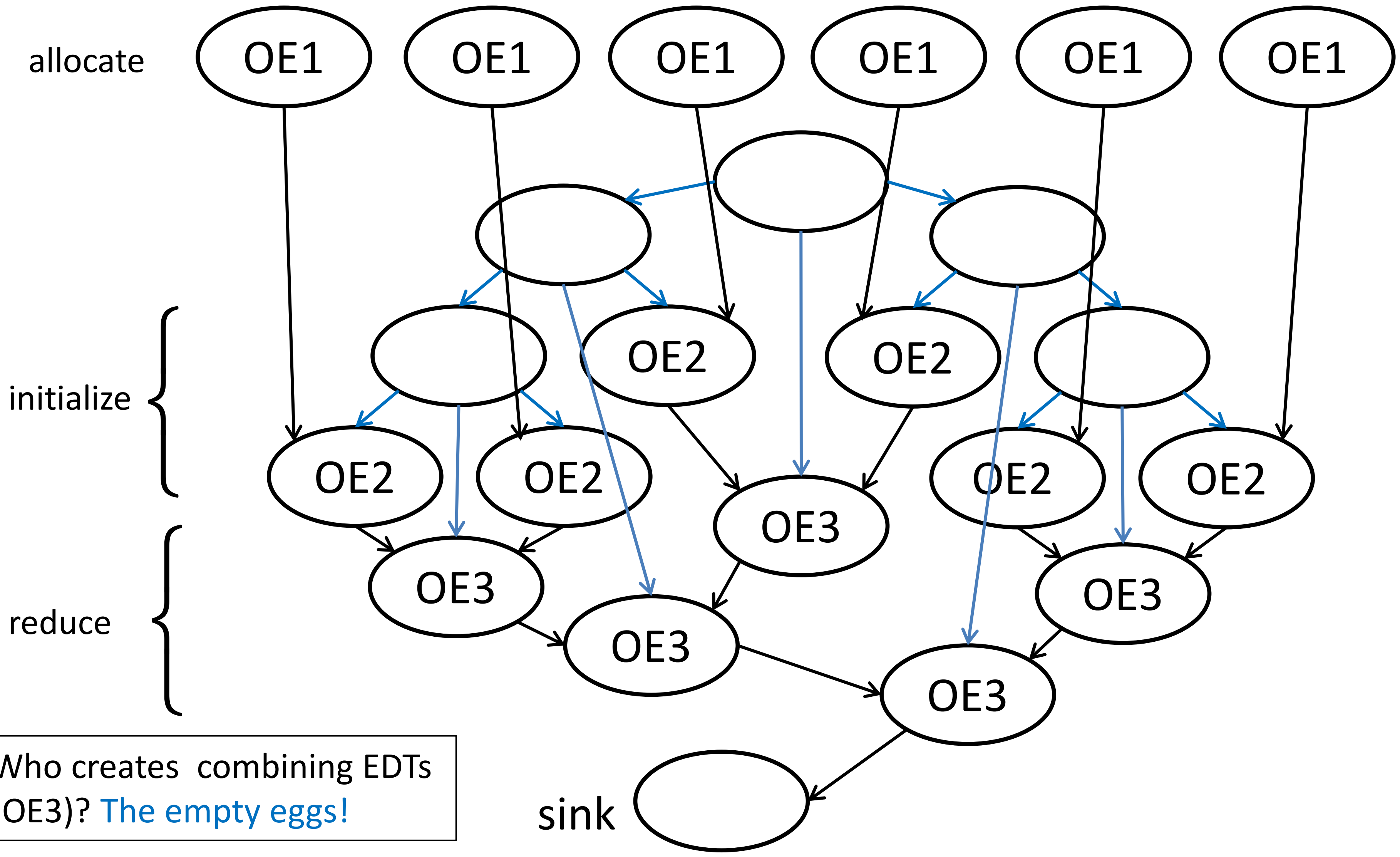


How does EDT know it is leaf without asking how many other leaves already created (global counter)?
Give EDT sequence # specific to location in tree.

OCR approach 3; binary reduction tree

1. Create N allocation EDTs, each of which allocates a DB for one vector, and satisfies output event OE1
2. Create tree of N initialization EDTs; each leaf waits on one OE1, initializes the corresponding vector, and satisfies ouput event OE2
3. Create tree of combine EDTs; each waits on two OE2s. Root combine EDT satisfies output event OE3
4. Create a wrapup EDT which waits on OE3 and then shuts down the runtime

Graph representation



Who creates combining EDTs (OE3)? The empty eggs!

Observations

- Need to know what comes after a certain module (e.g. how reduction takes place) to write that module: ~~Causality~~
- Trees abound
  - Crowns intertwined
  - Cannot afford to build new trees often at exascale
  - Cannot create all EDTs in single tree instantiation
    - ~~Modularity~~
    - Task queue overflow
    - Don’t always know number of iterations
- Guids galore
  - Where to store? Must distribute.
  - How to pass to other EDTs?
  - If replaced by user assigned IDs, how to guarantee object has come into existence when referencing ID?

Ponder this

Premise:

- OCR was not designed for data parallelism or static load balancing
- OCR was designed for exascale

Question:

Is there a reason to believe that OCR will scale better on problems that map to graphs with complicated, dynamically discovered dependencies than on those that map to simple graphs that can be load balanced statically? If yes, why?

Case Studies:

- Vector Reduction
  - Scalability, modularity, composability & OCR
  - Rob Van der Wijngaart of Intel
- FFT
  - Divide and conquer with OCR
  - Univ. OR undergrad project

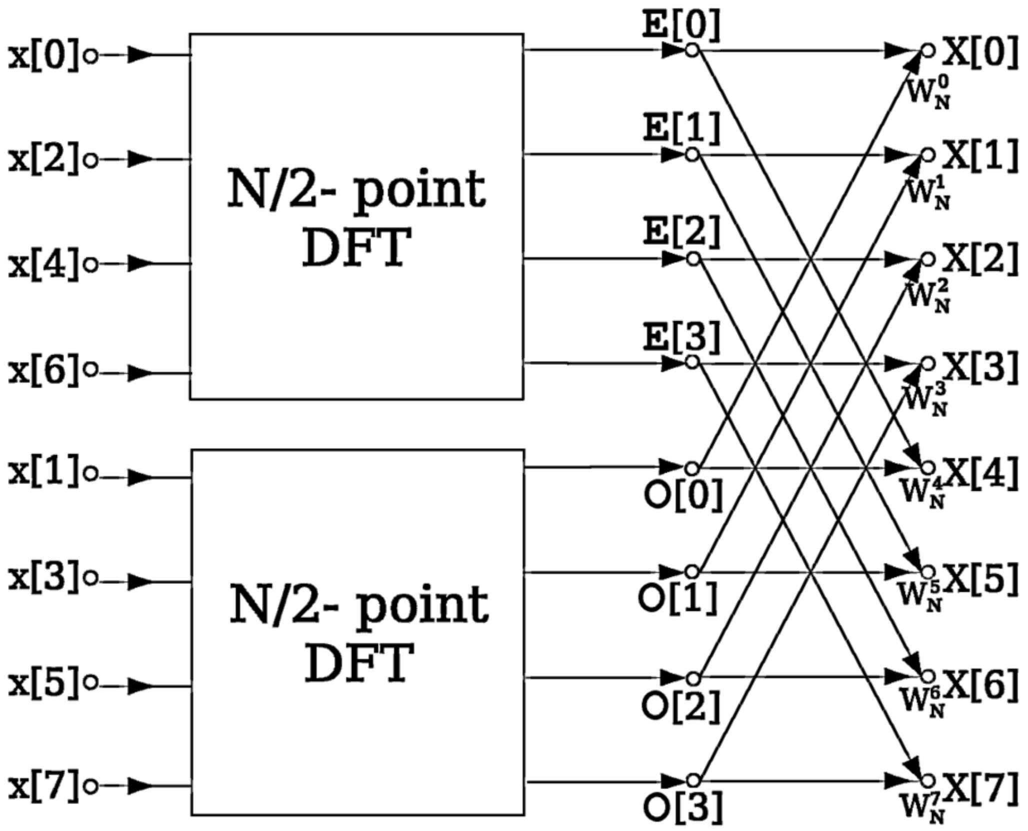
Background

- **Final year undergraduate project in Oregon State University**
- **OCR implementation of Fast Fourier Transform**
  - Cooley-Tukey algorithm
  - Evolution from serial version
  - OCR behavior

This research was, in part, funded by the U.S. Government, DOE and DARPA. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

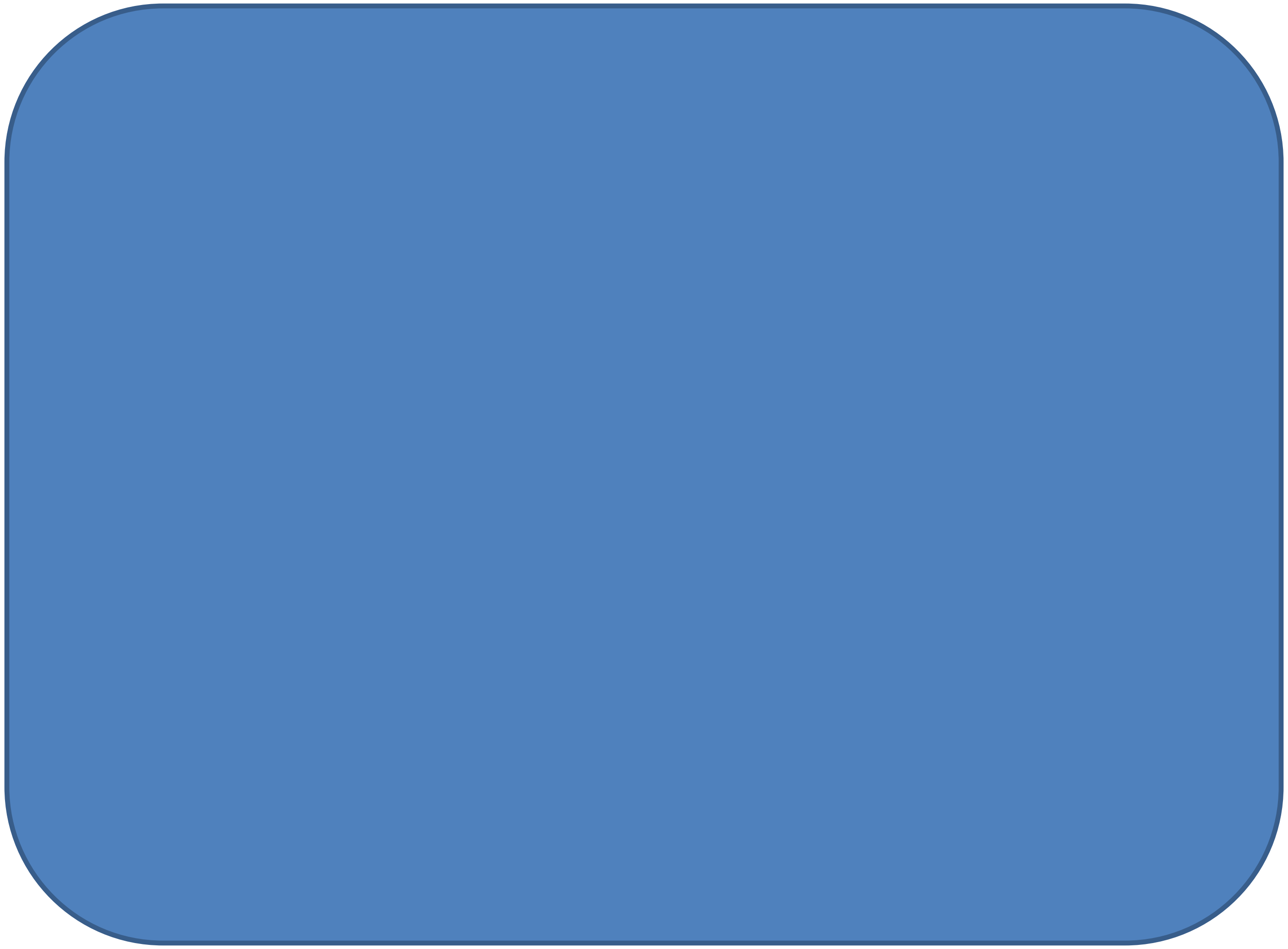
Algorithm

- **Divide-and-conquer**
- **Data-flow friendly**



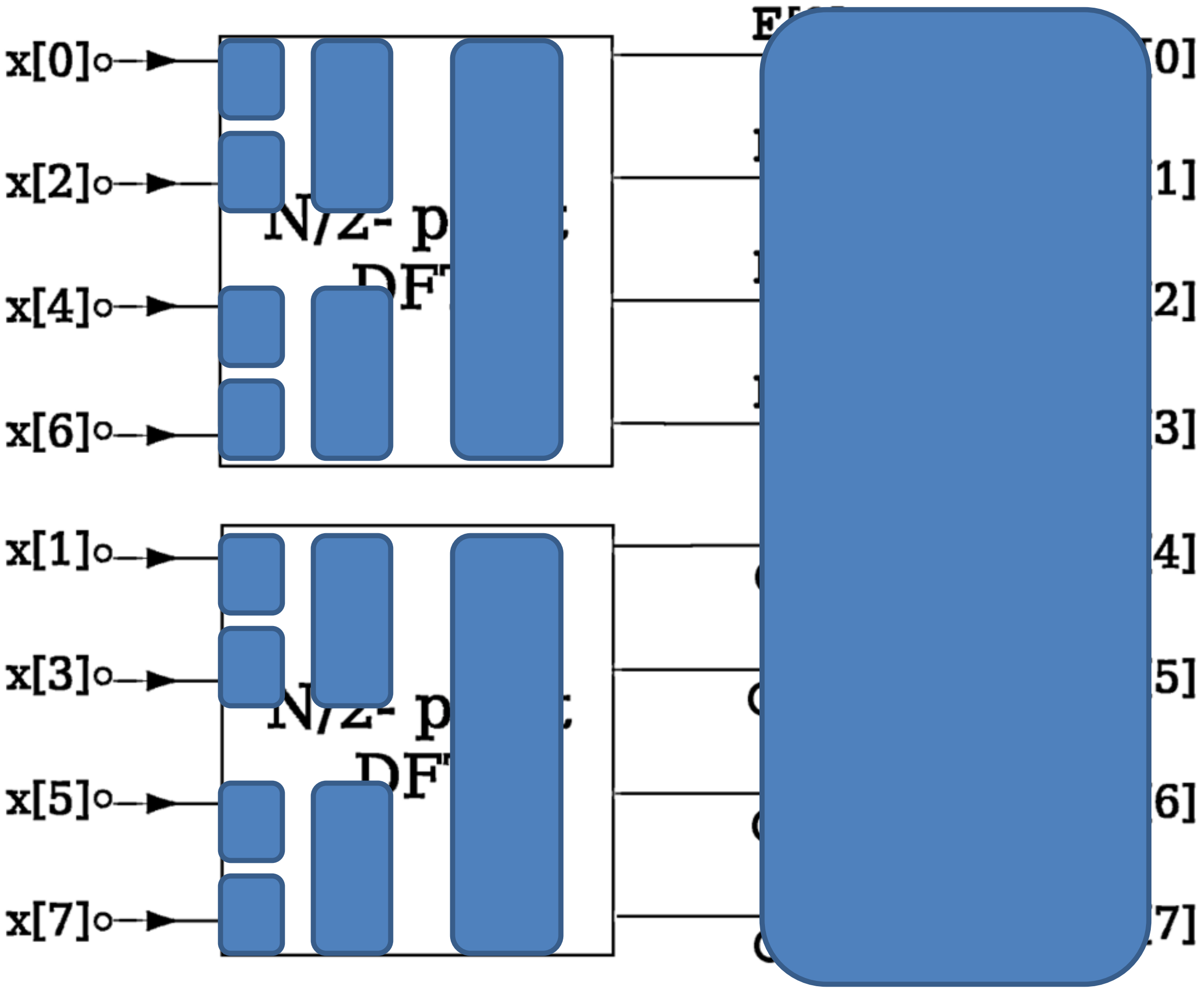
Source:Wikimedia Commons

Serial implementation



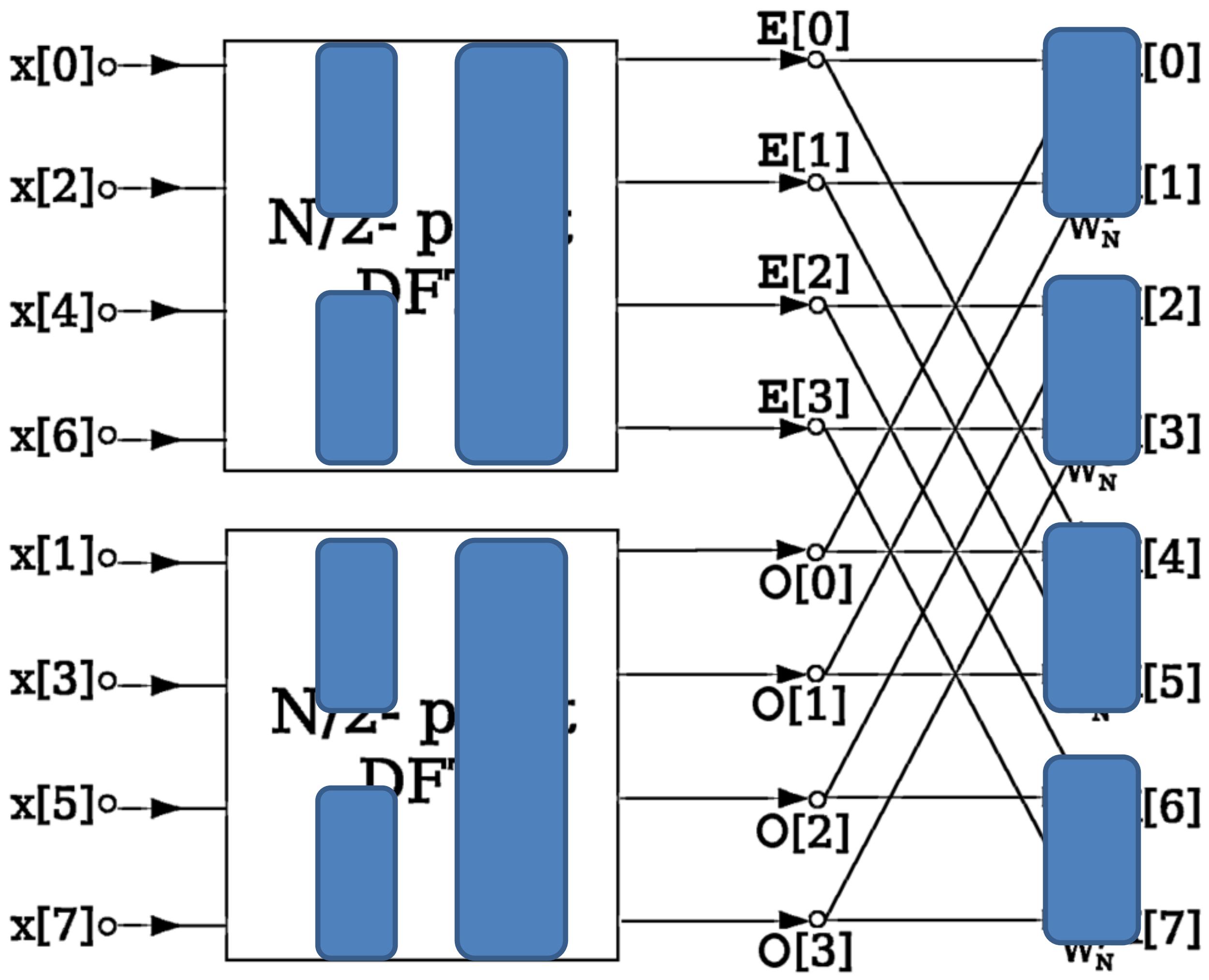
Source:Wikimedia Commons

Naïve implementation



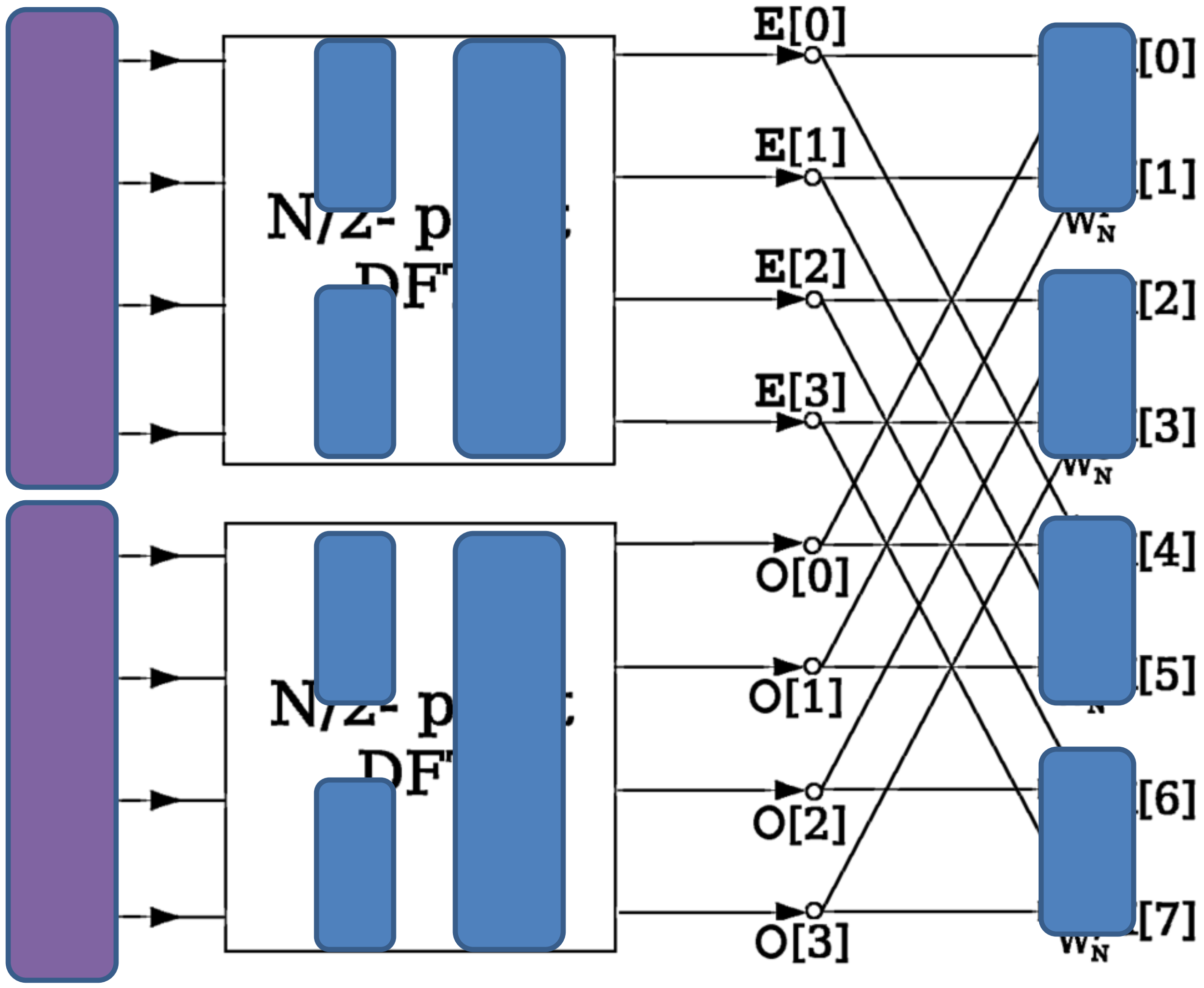
Source:Wikimedia Commons

Bounded implementation



Source:Wikimedia Commons

Bounded implementation with datablock



Source:Wikimedia Commons

Behavior

Version	No. of EDTs	Mean EDT Longevity (us)	Load variance across cores (%)	Running time (s)
Serial	2	1673420	70.7	3.36
Naïve parallel	12582913	253	5.1	877.0
Bounded parallel	1793	1982	2.7	0.46
Bounded parallel w/ datablocks	1793	1946	2.9	0.45

- **OCR X86 running FFT on 2^{32} sized dataset**
  - 2.9GHz Xeon 16 cores; 8 cores made available to OCR
- **Balance to be achieved between number and size of EDTs**

Summary

- **Serial implementation**
- **Naïve parallelization – recursive division of DFT**
- **Bounded parallelization – division bounded by a working set size**
- **Bounded parallelization with datablocks – additionally, use 3 datablocks (input, real, imaginary portions)**

- **Possible next steps for better parallelism**
  - Finer datablocks
  - Staggered creation of EDTs in the twiddle phase