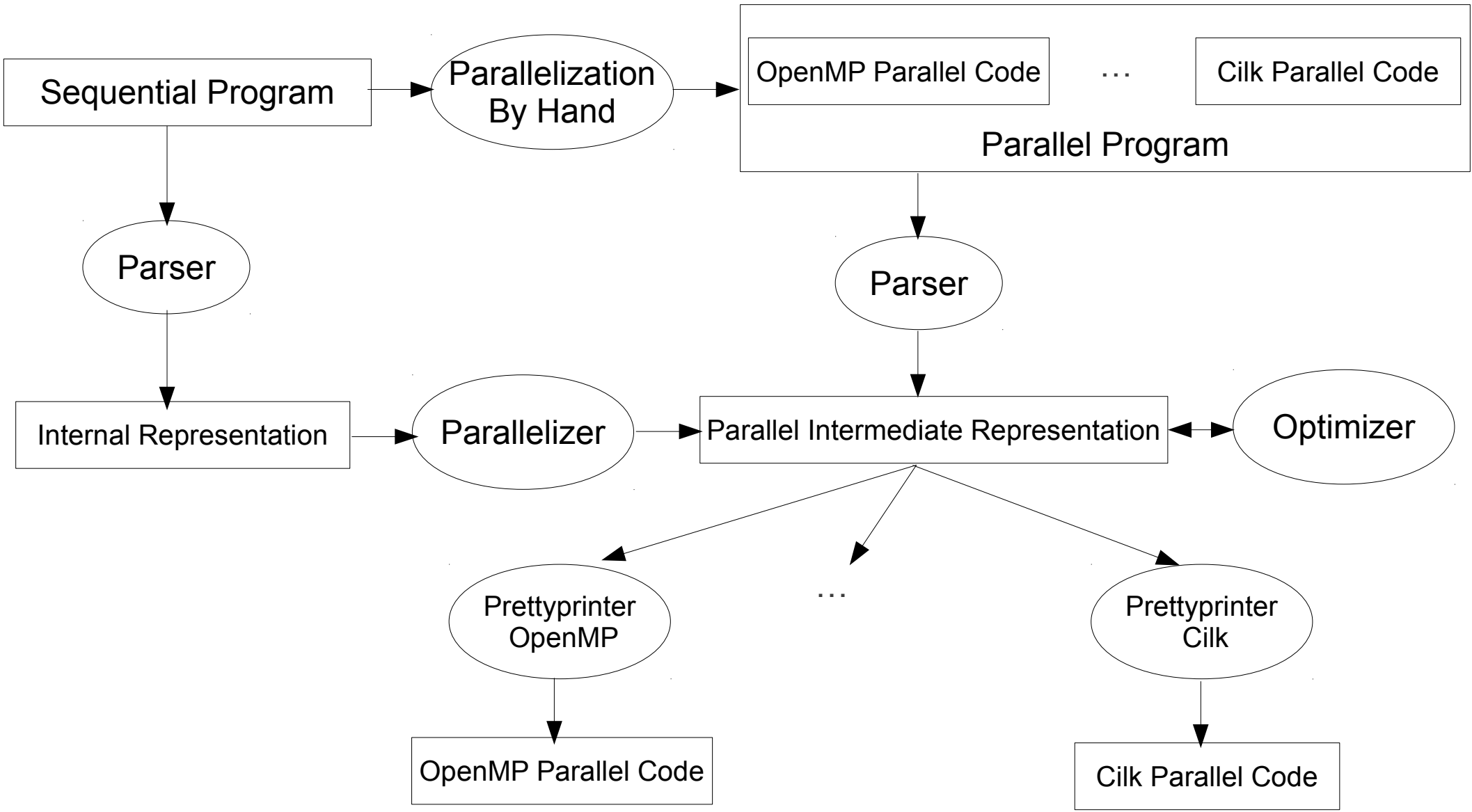
Towards a Better Abstraction of Multi-threading and Synchronization: A Case Study of Explicit Parallel Programming Languages

Dounia KHALDI
CRI, Mathématiques et systèmes
MINES ParisTech

Quatrièmes rencontres de la communauté française de compilation
Saint-Hippolyte, 05-07 décembre 2011

Context and Objectives



Which parallel language?

Challenges to write parallel code?

Main Parallel Language Features

- Data Parallelism (SIMD)
- Task Parallelism (MIMD)
  - Thread creation (spawn / fork, cobegin, futures...),
  - Thread termination (finish, …).
- Synchronization
  - Mutual exclusion in accesses to shared resources,
  - Thread termination (finish, …),
  - Collective barrier synchronization,
  - Point to point synchronization,
  - Single operation.
- Data Localization
- Shared Memory, Distributed Memory, PGAS (Partitioned Global Address Space).

A Case Study of Task-Parallel Languages

- Data and task parallel execution models,
- Explicit parallelism,
- High level abstraction,
- New constructs for synchronization.

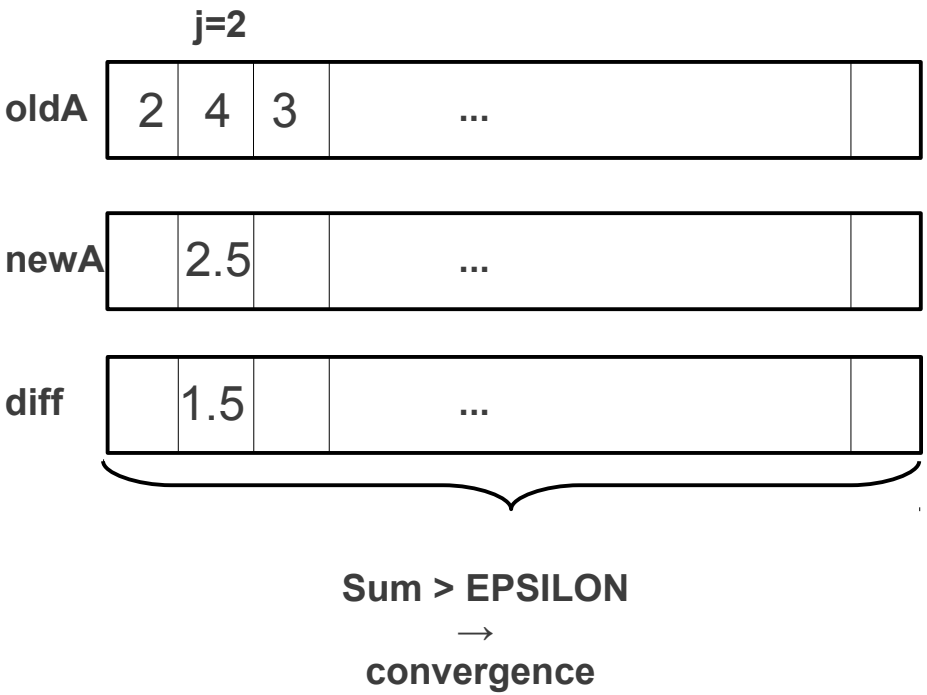
Language	Organization	Architecture
Cilk	MIT	SMP
Chapel	Cray	PGAS+DMP(GASNet)
X10	IBM	PGAS
Habanero-Java	Rice	PGAS
OpenMP		SMP
(OpenCL)		GPU,CPU,GPUs+CPUs

Cilk

- **Cilk** : function capable of being spawned in parallel,
- **Spawn** : child thread execution.
- **sync** : local barrier
- **Inlet :**
  - Internal, local to a Cilk function,
  - An inlet takes a result of a spawned function and use it into a function,
- **Abort :**
  - Allows to abort a speculative work,
  - Called inside an inlet,
  - Causes all of the other spawned children of the procedure to terminate.
- Lock : **cilk_lockvar, cilk_lock, cilk_unlock.**

```
cilk char *compute_next(char *elt);
cilk char *find(char *elt)
{
   char *newElt;
   inlet void isFound(char *res)
   {
     if(res == searched_elt)
       abort;
   }
   nextElt=compute_next(elt);
   isFound(spawn find(nextElt));
   sync;
   return nextElt;
}
cilk int main()
{
  char *elt = initElt();
  char *result = spawn find(elt);
  sync;
  return 0;
}
```

Cilk : One-Dimensional Iterative Averaging




```
#define EPSILON 1
#define N 10000
cilk void average (int j)
{
  double *newA = newA0; //Local pointer
  double *oldA = oldA0; //Local pointer
  newA[j] = (oldA[j-1]+oldA[j+1])/2.0f ;
  diff[j] = abs(newA[j]-oldA[j]);
}
cilk void main()
{
   double *newA0=malloc(n+2*sizeof(double));
   double *oldA0=malloc(n+2*sizeof(double));
   double *diff=malloc(n+2*sizeof(double));
   double *temp;
   int iters = 0,j; float sum = EPSILON+1;
   while ( sum > EPSILON ) {
      for ( j = 1 ; j <= N ; j++ ) {
         spawn average (j);
      } // for
      sync;
      for ( j = 1 ; j <= N ; j++ )
         sum += diff[j];
      iters++;
      temp = newA; newA = oldA; oldA = temp;
   }
   printf("Iterations:%d " , iters);
}
```

Chapel

- Structured task parallel creation : **cobegin{stmt$_1$;stmt$_2$;...;stmt$_n$}**.
- Unstructured Task-Parallel creation : **begin{stmt}.**
- Loop variant of the cobegin statement : **coforall index in 0..n do stmt.**
- **sync** variable : full or empty + value,
  - Reading empty variable, writing full variable : thread suspension,
  - Writing empty variable : state changement to full atomically,
  - Reading full variable : value consumption and state changement to empty atomically,
  - Used for futures when combined with **begin**.
- Atomic sections : **atomic{stmt}**.

```
var F$: sync real;              //empty
begin F$ = AsyncCompute();      //full
OtherCompute();
ReturnResults(F$);              //empty
```

Chapel

One-Dimensional Iterative Averaging using coforall

```
var iters = 0;
var delta :real;
delta = epsilon+1;
while ( delta > epsilon ) {
  coforall  j in 2..n-1 do {
   newA[j] = (oldA[j-1]+oldA[j+1])/2.0;
   diff[j] = abs(newA[j]-oldA[j]);
  }
  delta = 0;
  for  j in 1..n do
          delta += diff[j];
  iters = iters+1;
  temp = newA; newA = oldA; oldA = temp;
}
writeln(" Iterations ", iters);
```

X10 "ten-times productivity boost"

- **async <stmt>:** new asynchronous thread creation.
- **finish <stmt>:** local barrier on child threads created within stmt.
- Parent activity is concurrent with children's activities.
- **future f:** Future task creation
  - *f* will run,
  - *f*.**force**() waits completion of *f* , and propagates its return value.

```
finish {
   async { // Compute oddSum
      for (int i = 1 ; i <= n ; i += 2 )
         oddSum += i;
   }
   // Compute evenSum
   for (int j = 1 ; j <= n ; j += 2 )
        evenSum += j;
} // finish
```

```
future<int> Fi = future{ fib(10)};
int i = Fi.force();
```

X10

- **Clock:** computation that occurs in phases.
- A task is registered with zero or more clocks.
- **next:** thread suspension until all clocks that it is registered on can advance,
- clock advancement when all tasks registered with it execute a next.
- **Atomic:** atomicity for the set of instructions actually executed in the atomic statement.

```
atomic count += countPoints(n, rand));
```

```
finish async { // Outer async for clock allocation
   clock C = clock.factory.clock();
   async clocked(C){
     Phase1;
     next; // Barrier
     Phase2;
   } // async
   async clocked(C){
     Phase3;
     next; // Barrier
     Phase4;
   } // async
} // finish async
```

X10

One-Dimensional Iterative Averaging using Async, Finish and Clocks

```
finish async { // Outer async for clock allocation
   double[] newA0 = new double[n+2];
   double[] oldA0 = new double[n+2];
   ...
   delta = EPSILON+1; iters = 0;
   clock C = clock.factory.clock();
   for ( j = 1 ; j <= n ; j++ ) {
      async clocked(C){
          double[] newA = newA0; //Local pointer
          double[] oldA = oldA0; //Local pointer
          while ( delta > EPSILON ) {
            newA[j] = (oldA[j-1]+oldA[j+1])/2.0f ;
            diff[j] = Math.abs(newA[j]-oldA[j]);
            next; // Barrier
            if (j == 1) { //one activity calcultes the changed elements
               delta = diff.sum(); iters++;
            }
            next; // Barrier
            temp = newA; newA = oldA; oldA = temp;
          } // while
      } // async
   } // for
} // finish async
System.out.println("Iterations: " + iters);
```

Habanero-Java = X10 + {phaser, isolated}

- **Phasers**: programming constructs that unify collective and point-to-point synchronization in task parallel programming,
- Scope of phaser is limited to immediately enclosing finish,
- Registration modes: SIG, WAIT, SIG_WAIT (Default mode), SIG_WAIT_SINGLE.
- **Next**
  - Phaser advancement to next phase,
  - Task suspension if wait capability (WAIT, SIG_WAIT, SINGLE).
- **isolated**
  - *Weak atomicity* : no guarantee on interaction with non-isolated statements.

```
finish async{
 phaser ph = new phaser();
 async phased(ph:SIG_WAIT){
  Phase1;
  next;
  Phase2;
 }
 async phased(ph) {
  Phase3;
  next;
  Phase4;
 }
}
```

```
forall (point [i]:[0:MAX_THREADS-1]) {
     int myCount = 0;
     for (int j=0; j<lengthPerThread; j++)
       if (test(i,j)==3) myCount++;
     isolated count += myCount;
   } // forall
```

OpenMP

- Dynamic scheduling (**omp task**)
  - Task instance generation each time a thread encounters a task directive.
  - Immediate scheduling on the same thread or postponement and assignement to any thread in the team.
- Static scheduling (**omp section**)
  - Concurrently execution of the enclosed sections.

```
#pragma omp parallel
{
   #pragma omp single nowait
   {
      /* this is the initial root task */
       #pragma omp task
       {
          /* this is first child task */
       }
       #pragma omp task
       {
          /* this is second child task */
       }
   }
}
```

```
#pragma omp parallel
{
   #pragma omp sections
   {
       #pragma omp section
       {
          /* this is first child task */
       }
       #pragma omp section
       {
          /* this is second child task */
       }
   }
}
```

OpenMP

- **omp barrier:**

  Scope : the innermost enclosing parallel region.

- **omp taskwait:**

  Suspension on the completion of child tasks generated since the beginning of the current task.

- **atomic** and **critical**.

```
void foo ()
{
    int a, b, c, x, y;

    #pragma omp task shared(a)
    a = A();

    #pragma omp task shared (b, c, x)
    {
        #pragma omp task shared(b)
        b = B();

        #pragma omp task shared(c)
        c = C();

        #pragma omp taskwait
    }
    x = f1 (b, c);

    #pragma omp taskwait

    y = f2 (a, x);
}
```

OpenMP

One-Dimensional Iterative Averaging using task

```
void main()
{
   int iters = 0; float delta = epsilon+1;
   #pragma omp parallel
   {
       #pragma omp single nowait
       {
           while ( delta > epsilon ) {
              for ( j = 1 ; j <= n ; j++ ) {
                #pragma omp task
                {
                    newA[j] = (oldA[j-1]+oldA[j+1])/2.0f;
                    diff[j] = abs(newA[j]-oldA[j]);
                }
              } // for
              #pragma omp taskwait
              for ( j = 1 ; j <= n ; j++ )
                 delta += diff[j];
              iters++;
              temp = newA; newA = oldA; oldA = temp;
           }//while
        }//single
   }//parallel
   printf("Iterations:%d " , iters);
}
```

Discussion: Common Features

Language	Task spawn	Task join	Synchronization	Critical section	Data parallelism
Cilk(MIT)	Spawn	Sync	sync	locks	For spawn
Chapel (Cray)	Begin cobegin	Begin--> none cobegin--> implicit join	sync	sync, atomic	Forall coforall
X10 (IBM),	async	finish	Futures, clocks	atomic	foreach
Habanero-Java(Rice)	async	finish	Phasers, futures	isolated	foreach
OpenMP	omp task	Omp taskwait	Omp barrier	Omp critical Omp atomic	Omp for
OpenCl	enqueuetask	clFinish	EnqueueBarrier, events	atom_add atom_sub ...	clEnqueueNDRange-Kernel
Parallel intermediate representation	**spawn**	**wait**	**signal() + wait()**	**atomic**	**parallel**

Discussion: Specific Features

- **abort** in CILK → Speculation
- **Atomicity:**
  - **atomic** vs **critical** in OpenMP,
  - **atomic** vs **isolated** in Habanero-Java.
- **Futures** :
  - X10 and HJ → **future**,
  - Chapel → combining **begin** with **sync**.
- **Single**:
  - OpenMP → **omp single,**
  - HJ → **SIG_WAIT_SINGLE**.

```
#pragma omp parallel for shared (x,n)private(i,y)
For ( i =0; i<n ; i ++) {
#pragma omp atomic
   x [i] += work1(i);
   y [i] += work2(i);
}
```

Updates of two different elements of x are allowed to occur in parallel

```
Thread 1
1 :
2 : ptr = head ;
3 : isolated {
4 : ready = true;
5 : }
6 :
```

```
Thread 2
isolated {
if (ready )
  temp -> next = ptr;
}
```

Non-isolated statement

```
var F$: sync real;              //empty
begin F$ = AsyncCompute();      //full
OtherCompute();
ReturnResults(F$);              //empty
```

Conclusion

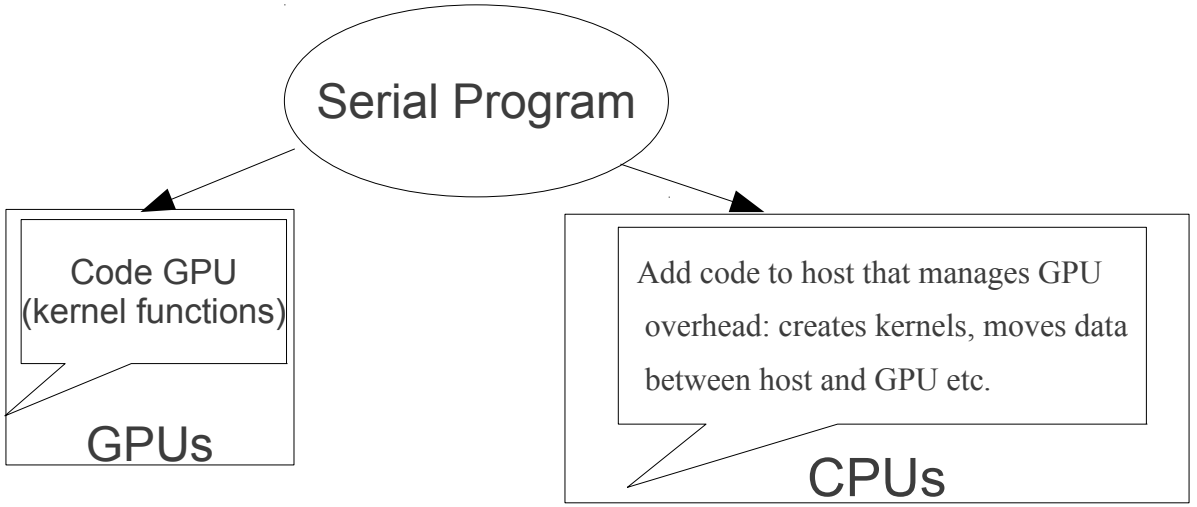
- Cilk, Chapel, X10, HJ, OpenMP, (OpenCL).
- General parallelism (data and task parallelism supported).
- Abstractions for parallelism do not extend easily to synchronization.
- Atomicity via locks and critical sections.

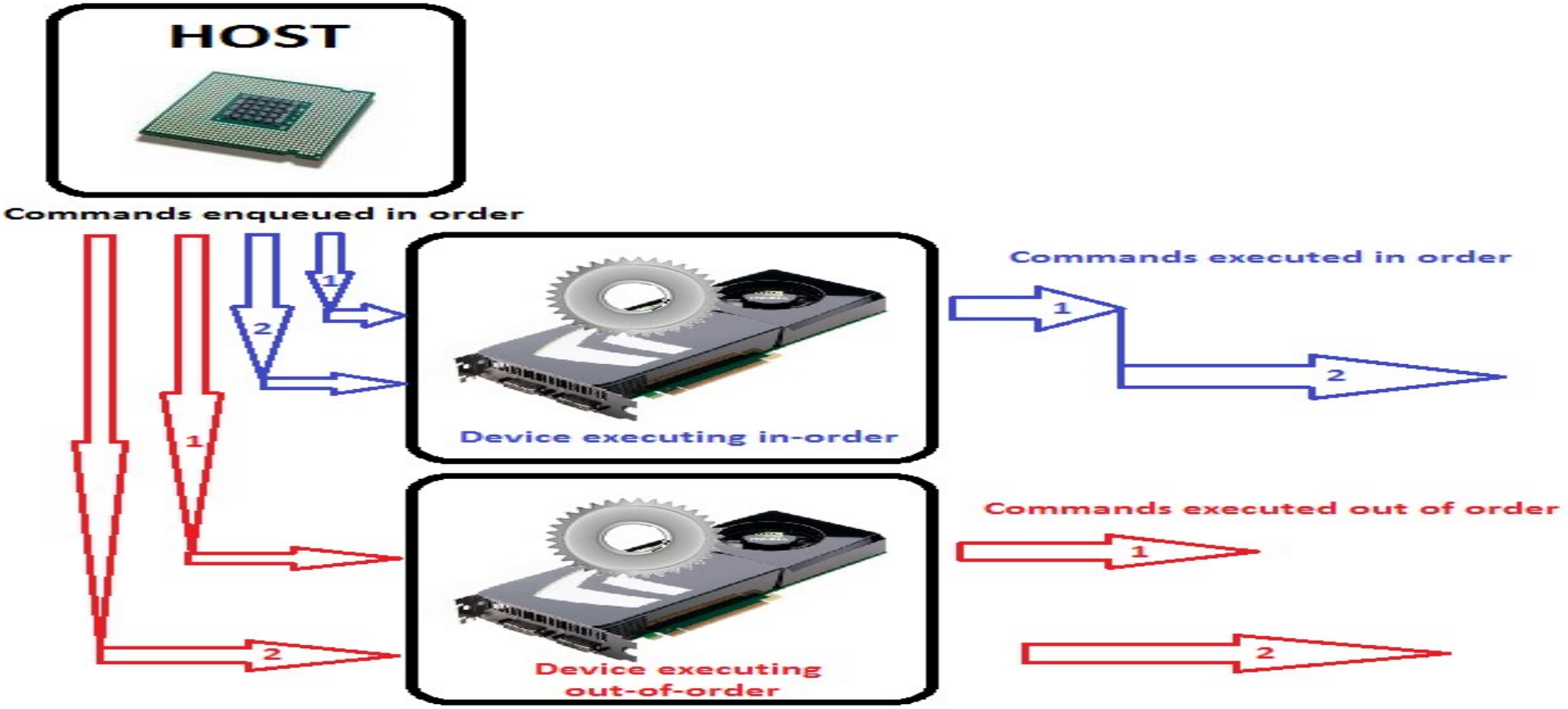
Current Work

- Parallel intermediate representation for the PIPS source-to-source compilation framework.

OpenCL

- Work-item (thread),
- Work-group,
- Global work size.





OpenCL

ClEnqueueNDRangeKernel

- Kernel execution on a device.
- Multiple work-groups execution in parallel.

ClEnqueueTask

- Event object check.
- Kernel execution on a device using a single work-item.

```
command_queue = clCreateCommandQueue(context, device_id, NULL, &err);
err = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global_work_size,
NULL, 0, NULL, NULL);
//create queue enabled for out of order (parallel) execution
commands = clCreateCommandQueue(context, device_id,
OUT_OF_ORDER_EXEC_MODE_ENABLE, &err);
...
// no synchronization
clEnqueueTask(command_queue, kernel_E, 0, NULL,NULL);
// synchronize so that kernel E starts only after kernels A,B,C,D finish
cl_event events[4]; // define event object array
clEnqueueTask(commands, kernel_A, 0, NULL, &events[0]);
clEnqueueTask(commands, kernel_B, 0, NULL, &events[1]);
clEnqueueTask(commands, kernel_C, 0, NULL, &events[2]);
clEnqueueTask(commands, kernel_D, 0, NULL, &events[3]);
clEnqueueTask(commands, kernel_E, 4, events, NULL);
//clEnqueueWaitForEvents (commands,4,events);
```

OpenCL

- Coarse grained synchronization : **clEnqueueBarrier**.
- Fine grained synchronization : **ClEnqueueWaitForEvents.**
- Atomic Operations

atom_add(), atom_sub(), atom_xchg(), atom_inc(), atom_dec(), atom_cmpxchg().

```
//create queue enabled for out of order (parallel) execution
commands = clCreateCommandQueue(context, device_id,
OUT_OF_ORDER_EXEC_MODE_ENABLE, &err);
...
// no synchronization
clEnqueueTask(command_queue, kernel_E, 0, NULL,NULL);
clEnqueueTask(commands, kernel_A, 0, NULL, NULL);
clEnqueueTask(commands, kernel_B, 0, NULL, NULL);
clEnqueueTask(commands, kernel_C, 0, NULL, NULL);
clEnqueueTask(commands, kernel_D, 0, NULL, NULL);
// synchronize so that kernel E starts only after kernels A,B,C,D finish
clEnqueueBarrier(commands);
clEnqueueTask(commands, kernel_E, 0, NULL, NULL);
```

OpenCL

One-Dimensional Iterative Averaging

```
__kernel void kernel_average(int j)
{
  newA[j] = (oldA[j-1]+oldA[j+1])/2.0f;
  diff[j] = abs(newA[j]-oldA[j]);
}
```

```
/* ---- Main program ---- */
cl_context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);
program = clCreateProgramWithSource(context, 1, &program_source, NULL, NULL);
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "kernel_average", NULL);
// set the args values to the compute kernel
global_work_size[0] = n;
// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue,kernel,1,NULL,global_work_size,
                NULL,0,NULL, NULL);
```