Task Parallelism and Data Distribution: An Overview of Explicit Parallel Programming Languages

Dounia Khaldi   Pierre Jouvelot   Corinne Ancourt   François Irigoin

CRI, Mathématiques et systèmes
MINES ParisTech
35 rue Saint-Honoré, 77300 Fontainebleau, France

25th International Workshop on Languages and Compilers for Parallel Computing, Tokyo
September 12, 2012

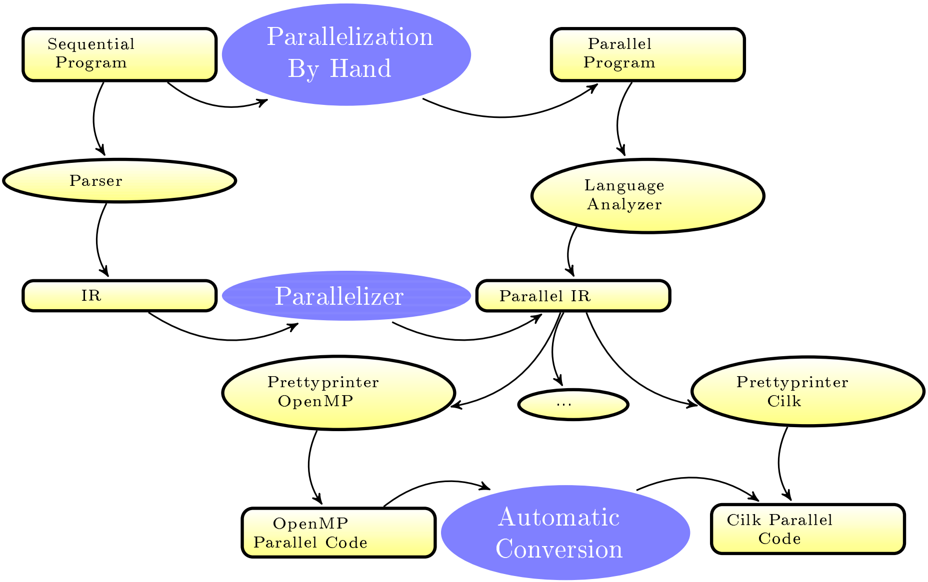
Main Parallel Language Features

1. Data parallelism (SIMD)
2. Task parallelism (MIMD)
   - Thread creation (spawn, cobegin, futures...)
   - Thread termination (finish, . . .).
3. Synchronization
   - Mutual exclusion in accesses to shared resources
   - Join operations (finish, . . .)
   - Collective barrier synchronization
   - Point-to-point synchronization
4. Memory models
   - Shared memory
   - Message passing
   - PGAS (Partitioned Global Address Space) [Yelick et al., 2007]

Response to the Many/Multicore Challenge

- Cilk, X10, Habanero-Java, Chapel, OpenMP, (OpenCL)
- Data and task parallel execution models
- Explicit parallelism
- High-level parallel abstractions
- New constructs for synchronization
- Rich memory models

Motivation



Sequential Program
Parallelization By Hand
Parallel Program
Parser
Language Analyzer
IR
Parallelizer
Parallel IR
Prettyprinter OpenMP
...
Prettyprinter Cilk
OpenMP Parallel Code
Automatic Conversion
Cilk Parallel Code

Outline

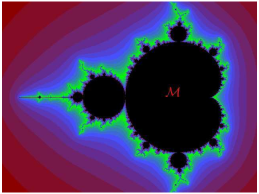
1. Mandelbrot Set Computation
2. Overview of Cilk, X10 and Habanero-Java
3. Point-to-Point Synchronization
4. Overview of Chapel and OpenMP
5. Atomicity
6. Conclusion

Mandelbrot Set Computation

$$M = \{c \in \mathbb{C} / \lim_{n\to\infty} z_n(c) < \infty\}$$

where

$$z_0(c) = c$$
$$z_{n+1}(c) = z_n^2(c) + c.$$




```c
for (row = 0; row < height; ++row) {
  for (col = 0; col < width; ++col) {
    z.r = z.i = 0;
    /* Scale c as display coordinates of current point */
    c.r = r_min + ((double) col * scale_r);
    c.i = i_min + ((double) (height-1-row) * scale_i);
    /* Iterates z = z*z+c while modulus(z) < 2, or maxiter is reached */
    k = 0;
    do {
      temp = z.r*z.r - z.i*z.i + c.r;
      z.i = 2*z.r*z.i + c.i; z.r = temp;
      ++k;
    } while (z.r*z.r + z.i*z.i < (2*2) && k < maxiter);
    /* Set color and display point */
    color = (ulong) ((k-1) * scale_color) + min_color;
    XSetForeground (display, gc, color);
    XDrawPoint (display, win, gc, col, row);
  }
}
```

Cilk [Blumofe et al., 1995]

- `cilk`: function capable of being spawned in parallel
- `spawn`: child thread creation
- `sync`: local barrier
- `inlet`:
  - Local to a Cilk function
  - Uses the result of a spawned function
- `abort`:
  - Allows to stop a speculative work
  - Called inside an inlet
  - Causes all spawned children to terminate
- Lock: `cilk_lockinit`, `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 Implementation of the Mandelbrot Set (-nproc P)

```
cilk int main(){
  ...
  cilk_lock_init(display_lock);
  for (m = 0; m < P; m++)
    spawn compute_points(m);
  sync;
}
cilk void compute_points(uint m) {
  for (row = m; row < height; row +=P)
    for (col = 0; col < width; ++col) {
      do {
        temp = z.r*z.r - z.i*z.i + c.r;
        z.i = 2*z.r*z.i + c.i; z.r = temp;
        ++k;
      } while (z.r*z.r + z.i*z.i < (2*2) && k < maxiter);
      color = (ulong) ((k-1) * scale_color) + min_color;
      cilk_lock(display_lock);
      XSetForeground (display, gc, color);
      XDrawPoint (display, win, gc, col, row);
      cilk_unlock(display_lock);
    }
}
```

row number	thread
$0 + \text{row} * P$	0
$1 + \text{row} * P$	1
$2 + \text{row} * P$	0
$3 + \text{row} * P$	1

X10 [Charles et al., 2005] and Habanero-Java (HJ) [Cavé et al., 2011]

- `async` <stmt>: asynchronous thread creation
- `finish` <stmt>: barrier on child threads created within stmt
- Parent activity is concurrent with children's activities

```
finish {
   async { // Compute oddSum in child activity
      for (int i = 1 ; i <= n ; i += 2 )
         oddSum.val += i;
   }
   // Compute evenSum in parent activity
   for (int j = 2 ; j <= n ; j += 2 ) evenSum += j;
} // finish
```

- `future` f: future task creation
  - f will run
  - f.`force`() waits the completion of f , and yields its value

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

- `atomic`(HJ:`isolated`): atomicity for a set of instructions

X10 (HJ) Implementation of the Mandelbrot Set

Data Distribution: Places

```
finish {
  for (m = 0; m < place.MAX_PLACES; m++){
    place pl_row = place.places(m);
    async at (pl_row) {
      for (row = m; row < height; row+=place.MAX_PLACES){
        for (col = 0; col < width; ++col) {
          do {
            temp = z.r*z.r - z.i*z.i + c.r;
            z.i = 2*z.r*z.i + c.i; z.r = temp;
            ++k;
          } while (z.r*z.r + z.i*z.i < (2*2) && k < maxiter);
          color = (ulong) ((k-1) * scale_color) + min_color;
          atomic {
            XSetForeground (display, gc, color);
            XDrawPoint (display, win, gc, col, row);
          }
        }
      }
    }
} }
```

row number			place.places(...)
	0 + row * place.MAXPLACES	0	
	1 + row * place.MAXPLACES	1	
	2 + row * place.MAXPLACES	0	
	3 + row * place.MAXPLACES	1	

Point-to-Point Synchronization: a Hide-and-Seek Game HJ: Phaser, X10: Clock, Cilk?

- Clock/Phaser: operates in phases of execution
- Each phase should wait for precedent ones to proceed
- `next`: thread suspension until all clocks/phasers which it is registered can advance
- Clock/Phaser advancement ⇒ all tasks registered with it execute a `next`

```
//X10
finish async {
 clock cl = clock.make();
 async clocked(cl) {
  count_to_a_number();
  next;
  start_searching();
 }
 async clocked(cl) {
  hide_oneself();
  next;
  continue_to_be_hidden();
 }
}
```

```
//Habanero-Java
finish async{
 phaser ph = new phaser();
 async phased(ph) {
  count_to_a_number();
  next;
  start searching();
 }
 async phased(ph) {
  hide_oneself();
  next;
  continue_to_be_hidden();
 }
}
```

```
//Cilk
cilk void searcher() {
 count_to_a_number();
 //missing
 point_to_point_sync();
 start_searching();
}
cilk void hidder() {
 hide_oneself();
 //missing
 point_to_point_sync();
 continue_to_be_hidden();
}
void main() {
  spawn searcher();
  spawn hidder();
}
```

Chapel [Chamberlain et al., 2007]

- Structured-task parallel creation: `cobegin`{$stmt_1$;$stmt_2$;...;$stmt_n$}
- Loop variant of the `cobegin` statement:
  `coforall` index `in` 0..n `do` stmt
- Unstructured task-parallel creation: `begin`{stmt}
- `sync` variable: (full or empty) + value
  - Reading empty, writing full variable: thread suspension
  - Writing empty: atomic state change to full
  - Reading full: value consumption and atomic state change to empty
  - Used for futures when combined with `begin`

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

- Atomic sections: `atomic`{stmt}

Data Distribution: Locales

```
coforall loc in Locales do
  on loc {
    for row in loc.id..height by numLocales do {
      for col in 1..width do {
        do {
          temp = z.r*z.r - z.i*z.i + c.r;
          z.i = 2*z.r*z.i + c.i; z.r = temp;
          k = k+1;
        } while (z.r*z.r + z.i*z.i < (2*2) && k < maxiter);
        color = (ulong) ((k-1) * scale_color) + min_color;
        atomic {
          XSetForeground (display, gc, color);
          XDrawPoint (display, win, gc, col, row);
        }
      }
    }
  }
}
```

row number	loc.id
$0 + row * numLocales$	0
$1 + row * numLocales$	1
$2 + row * numLocales$	0
$3 + row * numLocales$	1

OpenMP [http://www.openmp.org]

- Dynamic scheduling (`omp task`)
  - Task instance generated each time encountered
  - Immediate scheduling on the same thread or postponement and assignement to any thread
- Static scheduling (`omp section`): Concurrent execution of the enclosed sections
- `omp barrier`: for 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`

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

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

OpenMP Implementation of the Mandelbrot Set

```
P = omp_get_num_threads();
#pragma omp parallel shared(height,width,scale_r,\
         scale_i,maxiter,scale_color,min_color,r_min,i_min)\
         private(row,col,k,m,color,temp,z,c)
#pragma omp single
  for (m = 0; m < P; m++)
#pragma omp task
    for (row = m; row < height; row+=P){
      for (col = 0; col < width; ++col){
        do {
          temp = z.r*z.r - z.i*z.i + c.r;
          z.i = 2*z.r*z.i + c.i; z.r = temp;
          ++k;
        } while (z.r*z.r + z.i*z.i < (2*2) && k < maxiter);
        color = (ulong) ((k-1) * scale_color) + min_color;
#pragma omp critical
         {
           XSetForeground (display, gc, color);
           XDrawPoint (display, win, gc, col, row);
         }
}}
```

row number	thread
$0 + \text{row} * P$	0
$1 + \text{row} * P$	1
$2 + \text{row} * P$	0
$3 + \text{row} * P$	1

Atomicity
OpenMP: atomic vs critical, HJ: isolated vs atomic

- In OpenMP, `atomic` works faster than the `critical` directive
- Many atomic operations can be replaced with processor commands (GLSC)

```
//OpenMP
#pragma omp parallel for shared(x, index, n)
for (i=0; i<n; i++) {
#pragma omp atomic
   x[index[i]] += f(i); // index is supposed injective
}
```

- `isolated`: Weak atomicity model of Habanero-Java
- Atomicity with respect to the entire program (strong) or only to other atomic statements (weak)

```
//Habanero-Java-Thread 1
ptr = head;//non isolated statement
isolated {
  ready = true;
}
```

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

Conclusion

Language	Task creation	Synchronization			Data parallelism	Memory model
		Task join	Point-to-point	Atomic section		
Cilk (MIT)	spawn	sync abort	—	cilk_lock	—	Shared
X10 (IBM)	async future	finish	next force	atomic	foreach	PGAS (Places)
Habanero-Java (Rice)	async future	finish	next get	atomic isolated	foreach	PGAS (Places)
Chapel (Cray)	begin cobegin	—	sync	sync atomic	forall coforall	PGAS (Locales)
OpenMP	omp task	omp taskwait omp barrier	—	omp critical omp atomic	omp for	Shared
OpenCL	Enqueue Task	Finish EnqueueBarrier	events	atom_add, ...	EnqueueND-RangeKernel	Message passing

- Taxonomy ⇒ programmers
- Design solutions ⇒ language designers
- Parallel execution and memory models ⇒ automatic conversion tools
- SPIRE: A Sequential to Parallel Intermediate Representation Extension [Khaldi et al., 2012]

References

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995).
Cilk: An Efficient Multithreaded Runtime System.
In Journal of Parallel and Distributed Computing, pages 207–216.

Cavé, V., Zhao, J., and Sarkar, V. (2011).
Habanero-Java: the New Adventures of Old X10.
In 9th International Conference on the Principles and Practice of Programming in Java (PPPJ).

Chamberlain, B., Callahan, D., and Zima, H. (2007).
Parallel Programmability and the Chapel Language.
Int. J. High Perform. Comput. Appl, 21:291–312.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005).
X10: An Object-Oriented Approach to Non-Uniform Cluster Computing.
SIGPLAN Not., 40:519–538.

Khaldi, D., Jouvelot, P., Ancourt, C., and Irigoin, F. (2012).
SPIRE: A Sequential to Parallel Intermediate Representation Extension.
Technical Report CRI/A-487 (Submitted to CGO'13), MINES ParisTech.

Yelick, K., Bonachea, D., Chen, W.-Y., Colella, P., Datta, K., Duell, J., Graham, S. L., Hargrove, P., Hilfinger, P., Husbands, P., Iancu, C., Kamil, A., Nishtala, R., Su, J., Michael, W., and Wen, T. (2007).
Productivity and Performance Using Partitioned Global Address Space Languages.
In Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, PASCO '07, pages 24–32, New York, NY, USA. ACM.

Task Parallelism and Data Distribution: An Overview of Explicit Parallel Programming Languages

Dounia Khaldi Pierre Jouvelot Corinne Ancourt François Irigoin

CRI, Mathématiques et systèmes
MINES ParisTech
35 rue Saint-Honoré, 77300 Fontainebleau, France

25th International Workshop on Languages and Compilers for Parallel Computing, Tokyo
September 12, 2012

OpenCL [http://www.khronos.org/opencl]

- ClEnqueueNDRangeKernel
  - Kernel execution on a device
  - Multiple work-groups execution in parallel
- ClEnqueueTask
  - Event object handling
  - Kernel execution on a device using a single work-item
- Coarse grained synchronization: clEnqueueBarrier
- Fine grained synchronization: ClEnqueueWaitForEvents
- Atomic operations: atom_add(), atom_sub(), atom_xchg(), atom_inc()...

```
//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 Implementation of the Mandelbrot Set

```
__kernel void kernel_main(complex c, uint maxiter, double scale_color,
                          uint m, uint P, ulong color[NPIXELS][NPIXELS]) {
  for (row = m; row < NPIXELS; row+=P) {
    for (col = 0; col < NPIXELS; ++col) {
      //Initialization of c, k and z
      do {
        temp = z.r*z.r-z.i*z.i+c.r;
        z.i = 2*z.r*z.i+c.i; z.r = temp;
        ++k;
      } while (z.r*z.r+z.i*z.i<(2*2) && k<maxiter);
      color[row][col] = (ulong) ((k-1)*scale_color);
    }
  }
}
cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_DEFAULT, 1,
                      &device_id, &ret_num_devices);
cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
cQueue=clCreateCommandQueue(context,device_id,OUT_OF_ORDER_EXEC_MODE_ENABLE,NULL);
P = CL_DEVICE_MAX_COMPUTE_UNITS;
memc = clCreateBuffer(context, CL_MEM_READ_ONLY , sizeof(complex), c);
// ... Create read-only buffers with maxiter, scale_color and P too
memcolor = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                          sizeof(ulong)*height*width,NULL,NULL);
clEnqueueWriteBuffer(cQueue,memc,CL_TRUE,0,sizeof(complex),&c,0,NULL,NULL);
// ... Enqueue write buffer with maxiter, scale_color and P too
program = clCreateProgramWithSource(context, 1, &program_source, NULL, NULL);
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "kernel_main", NULL);
clSetKernelArg(kernel, 0, sizeof(cl_mem),(void *)&memc);
// ... Set kernel argument with memmaxiter, memscale_color, memP and memcolor too
for(m = 0; m < P; m++){
  memm = clCreateBuffer(context, CL_MEM_READ_ONLY , sizeof(uint), m);
  clEnqueueWriteBuffer(cQueue, memm, CL_TRUE, 0, sizeof(uint), &m, 0, NULL, NULL);
  clSetKernelArg(kernel, 0, sizeof(cl_mem),(void *)&memm);
  clEnqueueTask(cQueue, kernel, 0, NULL, NULL);
}
clFinish(cQueue);
clEnqueueReadBuffer(cQueue,memcolor,CL_TRUE,0,space,color,0,NULL,NULL);
for (row = 0; row < height; ++row)
  for (col = 0; col < width; ++col){
    XSetForeground (display, gc, color[col][row]);
    XDrawPoint (display, win, gc, col, row);
  }
```