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

Dounia Khaldi, Pierre Jouvelot, Corinne Ancourt and François Irigoin

CRI, Mathématiques et systèmes
MINES ParisTech
35 rue Saint-Honoré, 77300 Fontainebleau, France
firstname.lastname@mines-paristech.fr

Abstract. Programming parallel machines as effectively as sequential ones would ideally require a language that provides high-level programming constructs to avoid the programming errors frequent when expressing parallelism. Since task parallelism is considered more error-prone than data parallelism, we survey six popular and efficient parallel language designs that tackle this difficult issue: Cilk, Chapel, X10, Habanero-Java, OpenMP and OpenCL. Using as single running example a parallel implementation of the computation of the Mandelbrot set, this paper describes how the fundamentals of task parallel programming, i.e., collective and point-to-point synchronization and mutual exclusion, are dealt with in these languages. We discuss how these languages allocate and distribute data over memory. Our study suggests that, even though there are many keywords and notions introduced by these languages, they all boil down, as far as control issues are concerned, to three key task concepts: creation, synchronization and atomicity. Regarding memory models, these languages adopt one of three approaches: shared memory, message passing and PGAS (Partitioned Global Address Space). The paper is designed to give users and language and compiler designers an up-to-date comparative overview of current parallel languages.

1 Introduction

Parallel computing is about 50 years old. The market dominance of multi- and many-core processors and the growing importance and the increasing number of clusters in the Top500 list (`top500.org`) are making parallelism a key concern when implementing current applications such as weather modeling [13] or nuclear simulations [12]. These important applications require large computational power and thus need to be programmed to run on powerful parallel supercomputers. Programming languages adopt one of two ways to deal with this issue: (1) high-level languages hide the presence of parallelism at the software level, thus offering a code easy to build and port, but the performance of which is not guaranteed, and (2) low-level languages use explicit constructs for communication patterns and specifying the number and placement of threads, but the resulting code is difficult to build and not very portable, although usually efficient.

Recent programming models explore the best trade-offs between expressiveness and performance when addressing parallelism. Traditionally, there are two general ways to break an application into concurrent parts in order to take advantage of a parallel computer and execute them simultaneously on different CPUs: data and task parallelisms.

In data parallelism, the same instruction is performed repeatedly and simultaneously on different data. In task parallelism, the execution of different processes (threads) is distributed across multiple computing nodes. Task parallelism is often considered more difficult to specify than data parallelism, since it lacks the regularity present in the latter model; processes (threads) run simultaneously different instructions, leading to different execution schedules and memory access patterns. Task management must address both control and data issues, in order to optimize execution and communication times.

This paper describes how six popular and efficient parallel programming language designs tackle the issue of task parallelism specification: Cilk, Chapel, X10, Habanero-Java, OpenMP and OpenCL. They are selected based on the richness of their functionality and their popularity; they provide simple high-level parallel abstractions that cover most of the parallel programming language design spectrum. We use a popular parallel problem (the computation of the Mandelbrot set [4]) as a running example. We consider this an interesting test case, since it exhibits a high-level of embarrassing parallelism while its iteration space is not easily partitioned, if one wants to have tasks of balanced run times. Since our focus is here the study and comparison of the expressiveness of each language's main parallel constructs, we do not give performance measures for these implementations.

Our paper is useful to (1) programmers, to choose a parallel language and write parallel applications, (2) language designers, to compare their ideas on how to tackle parallelism in new languages with existing proposals, and (3) designers of optimizing compilers, to develop automatic tools for writing parallel programs. Our own goal was to use this case study for the design of SPIRE [14], a sequential to parallel intermediate representation extension of the intermediate representations used in compilation frameworks, in order to upgrade their existing infrastructure to address parallel languages.

After this introduction, Section 2 presents our running example. We discuss the parallel language features specific to task parallelism, namely task creation, synchronization and atomicity, and also how these languages distribute data over different processors in Section 3. In Section 4, a selection of current and important parallel programming languages are described: Cilk, Chapel, X10, Habanero Java, OpenMP and OpenCL. For each language, an implementation of the Mandelbrot set algorithm is presented. Section 5 compares and discusses these languages. We conclude in Section 6.

2 Mandelbrot Set Computation

The Mandelbrot set is a fractal set. For each complex $c \in \mathbb{C}$, the set of complex numbers $z_n(c)$ is defined by induction as follows: $z_0(c) = c$ and $z_{n+1}(c) = z_n^2(c) + c$. The Mandelbrot set M is then defined as $\{c \in \mathbb{C} / \lim_{n\to\infty} z_n(c) < \infty\}$; thus, M is the set of all complex numbers c for which the series $z_n(c)$ converges. One can show [4] that a finite limit for $z_n(c)$ exists only if the modulus of $z_m(c)$ is less than 2, for some positive m. We give a sequential C implementation of the computation of the Mandelbrot set in Figure 2. Running this program yields Figure 1, in which each complex c is seen as a pixel, its color being related to its convergence property: the Mandelbrot set is the black shape in the middle of the figure. We use this base program as our test case in our parallel implementations, in Section 4, for the parallel languages we

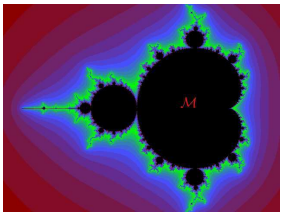

Fig. 1. Result of the Mandelbrot set

```
unsigned long min_color = 0, max_color = 16777215;
unsigned int width = NPIXELS; uint height = NPIXELS; uint N = 2, maxiter = 10000;
double r_min = –N, r_max = N, i_min = –N, i_max = N;
double scale_r = (r_max – r_min)/width;
double scale_i = (i_max – i_min)/heigth;
double scale_color = (max_color – min_color)/maxiter;
Display *display; Window win; GC gc;
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 |z| < N, 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 < (N*N) && 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);
  }
}
```

Fig. 2: Sequential C implementation of the Mandelbrot set

selected. This is an interesting case for illustrating parallel programming languages: (1) it is an embarrassingly parallel problem, since all computations of pixel colors can be performed simultaneously, and thus is obviously a good candidate for expressing parallelism, but (2) its efficient implementation is not obvious, since good load balancing cannot be achieved by simply grouping localized pixels together because convergence can vary widely from one point to the next, due to the fractal nature of the Mandelbrot set.

3 Task Parallelism Issues

Among the many issues related to parallel programming, the questions of task creation, synchronization, atomicity and memory model are particularly acute when dealing with task parallelism, our focus in this paper.

3.1 Task Creation

In this paper, a task is a static notion, i.e., a list of instructions, while processes and threads are running instances of tasks. Creation of system-level task instances is an expensive operation, since its implementation, via processes, requires allocating and later possibly releasing system-specific resources. If a task has a short execution time, this overhead might make the overall computation quite inefficient. Another way to introduce parallelism is to use lighter, user-level tasks, called threads. In all languages addressed in this paper, task management operations refer to such user-level tasks. The problem of finding the proper size of tasks, and hence the number of tasks, can be decided at compile or run times, using heuristic means.

In our Mandelbrot example, the parallel implementations we provide below use a static schedule that allocates a number of iterations of the loop `row` to a particular thread; we interleave successive iterations into distinct threads in a round-robin fashion, in order to group loop body computations into chunks, of size $\texttt{height}/P$, where P is the (language-dependent) number of threads. Our intent here is to try to reach a good load balancing between threads.

3.2 Synchronization

Coordination in task-parallel programs is a major source of complexity. It is dealt with using synchronization primitives, for instance when a code fragment contains many phases of execution where each phase should wait for the precedent ones to proceed. When a process or a thread exits before synchronizing on a barrier that other processes are waiting on or when processes operate on different barriers using different orders, a deadlock occurs. Programs must avoid these situations (and be deadlock-free). Different forms of synchronization constructs exist, such as mutual exclusion when accessing shared resources using locks, join operations that terminate child threads, multiple synchronizations using barriers[1], and point-to-point synchronization using counting semaphores [18].

In our Mandelbrot example, we need to synchronize all pixel computations before exiting; one also needs to use synchronization to deal with the atomic section (see next subsection). Even though synchronization is rather simple in this example, caution is always needed; an example that may lead to deadlocks is mentioned in Section 5.

3.3 Atomicity

Access to shared resources requires atomic operations that, at any given time, can be executed by only one process or thread. Atomicity comes in two flavors: weak and strong [16]. A weak atomic statement is atomic only with respect to other explicitly atomic statements; no guarantee is made regarding interactions with non-isolated statements (not declared as atomic). By opposition, strong atomicity enforces non-interaction of atomic statements with all operations in the entire program. It usually

[1]The term “barrier” is used in various ways by authors [17]; we consider here that barriers are synchronization points that wait for the termination of sets of threads, defined in a language-dependent manner.

requires specialized hardware support (e.g., atomic "compare and swap" operations), although a software implementation that treats non-explicitly atomic accesses as implicitly atomic single operations (using a single global lock) is possible.

In our Mandelbrot example, display accesses require connection to the X server; drawing a given pixel is an atomic operation since GUI-specific calls need synchronization. Moreover, two simple examples of atomic sections are provided in Section 5.

3.4 Memory Models

The choice of a proper memory model to express parallel programs is an important issue in parallel language design. Indeed, the ways processes and threads communicate using the target architecture and impact the programmer's computation specification affect both performance and ease of programming. There are currently three main approaches.
Message Passing This model uses communication libraries to allow efficient parallel programs to be written for distributed memory systems. These libraries provide routines to initiate and configure the messaging environment as well as sending and receiving data packets. Currently, the most popular high-level message-passing system for scientific and engineering applications is MPI (Message Passing Interface) [1]. OpenCL [2] uses a variation of the message passing memory model.
Shared memory Also called global address space, this model is the simplest one to use [7]. There, the address spaces of the threads are mapped onto the global memory; no explicit data passing between threads is needed. However, synchronization is required between the threads that are writing and reading data to and from the shared memory. OpenMP [3] and Cilk [8] use the shared memory model.
Partitioned Global Address Space PGAS-based languages combine the programming convenience of shared memory with the performance control of message passing by partitioning logically a global address space; each portion is local to each thread. From the programmer's point of view programs have a single address space and one task of a given thread may refer directly to the storage of a different thread. The three other programming languages in this paper use the PGAS memory model.

4 Parallel Programming Languages

We present here six parallel programming language designs and describe how they deal with the concepts introduced in the previous section. Given the large number of parallel languages that exist, we focus primarily on languages that are in current use and popular and that support simple high-level task-oriented parallel abstractions.

4.1 Cilk

Cilk [8], developed at MIT, is a multithreaded parallel language based on C for shared memory systems. Cilk is designed for exploiting dynamic and asynchronous parallelism. A Cilk implementation of the Mandelbrot set is provided in Figure 3[2].

[2]From now on, variable declarations are omitted, unless required for the purpose of our presentation.

```
{
  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) {
      //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 < (N*N) && 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);
    }
}
```

Fig. 3: Cilk implementation of the Mandelbrot set (`--nproc P`)

Task Parallelism The `cilk` keyword identifies functions that can be spawned in parallel. A Cilk function may create threads to execute functions in parallel. The `spawn` keyword is used to create child tasks, such as *compute_points* in our example, when referring to Cilk functions.

Cilk introduces the notion of inlets [5], which are local Cilk functions defined to take the result of spawned tasks and use it (performing a reduction). The result should not be put in a variable in the parent function. All the variables of the function are available within an inlet. `Abort` allows to abort a speculative work by terminating all of the already spawned children of a function; it must be called inside an inlet. Inlets are not used in our example.

Synchronization The `sync` statement is a local barrier, used in our example to ensure task termination. It waits only for the spawned child tasks of the current procedure to complete, and not for all tasks currently being executed.

Atomic Section Mutual exclusion is implemented using locks of type `cilk_lockvar`, such as *display_lock* in our example. The function `cilk_lock` is used to test a lock and block if it is already acquired; the function `cilk_unlock` is used to release a lock. Both functions take a single argument which is an object of type `cilk_lockvar`. `cilk_lock_init` is used to initialize the lock object before it is used.

Data Distribution In Cilk's shared memory model, all variables declared outside Cilk functions are shared. To avoid possible non-determinism due to data races, the programmer should avoid the situation when a task writes a variable that may be read or written concurrently by another task, or use the primitive `cilk_fence` that ensures that all memory operations of a thread are committed before the next operation execution.

4.2 Chapel

Chapel [10], developed by Cray, supports both data and control flow parallelism and is designed around a multithreaded execution model based on PGAS for shared and

distributed-memory systems. A Chapel implementation of the Mandelbrot set is provided in Figure 4.

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

Fig. 4: Chapel implementation of the Mandelbrot set

Task Parallelism Chapel provides three types of task parallelism [6], two structured ones and one unstructured. `cobegin`$\{stmts\}$ creates a task for each statement in $stmts$; the parent task waits for the $stmts$ tasks to be completed. `coforall` is a loop variant of the `cobegin` statement, where each iteration of the `coforall` loop is a separate task and the main thread of execution does not continue until every iteration is completed. Finally, in `begin`$\{stmt\}$, the original parent task continues its execution after spawning a child running $stmt$.
Synchronization In addition to `cobegin` and `coforall`, used in our example, which have an implicit synchronization at the end, synchronization variables of type `sync` can be used for coordinating parallel tasks. A `sync` [6] variable is either empty or full, with an additional data value. Reading an empty variable and writing in a full variable suspends the thread. Writing to an empty variable atomically changes its state to full. Reading a full variable consumes the value and atomically changes the state to empty.
Atomic Section Chapel supports atomic sections (`atomic`$\{stmt\}$): this indicates that $stmt$ should be executed atomically with respect to any other thread.
Data Distribution Chapel introduces a type called `locale` to refer to a unit of the machine resources on which a computation is running. A locale is a mapping of Chapel data and computations to the physical machine. In Figure 4, Array `Locales` represents the set of locale values corresponding to the machine resources on which this code is running; `numLocales` refers to the number of locales. Chapel also introduces new `domain` types to specify array distribution; they are not used in our example.

4.3 X10 and Habanero-Java

X10 [11], developed at IBM, is a distributed asynchronous dynamic parallel programming language for multi-core processors, symmetric shared-memory multiprocessors (SMPs), commodity clusters, high end supercomputers, and even embedded processors like Cell. A X10 implementation of the Mandelbrot set is provided in Figure 5.

Habanero-Java [9], under development at Rice University, is derived from X10 [11], and introduces additional synchronization and atomicity primitives surveyed below.

```
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) {
          //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 < (N*N) && k < maxiter);
          color = (ulong) ((k-1) * scale_color) + min_color;
          atomic {
            XSetForeground (display, gc, color);
            XDrawPoint (display, win, gc, col, row);
          }
}}}}}
```

Fig. 5: X10 implementation of the Mandelbrot set

Task Parallelism X10 provides two task creation primitives: (1) the `async` $stmt$ construct creates a new asynchronous task that executes $stmt$, while the current thread continues, and (2) the `future` exp expression launches a parallel task that returns the value of exp.

Synchronization With `finish` $stmt$, the current running task is blocked at the end of the `finish` clause, waiting till all the children spawned during the execution of $stmt$ have terminated. The expression `f.force()` is used to get the actual value of the "future" task `f`.

X10 introduces a new synchronization concept: the clock. It acts as a barrier for a dynamically varying set of tasks [19] that operate in phases of execution where each phase should wait for previous ones before proceeding. A task that uses a clock must first register with it (multiple clocks can be used). It then uses the statement `next` to signal to all the tasks that are registered with its clocks that it is ready to move to the following phase, and waits until all the clocks with which it is registered can advance. A clock can advance only when all the tasks that are registered with it have executed a `next` statement.

Habanero-Java introduces phasers to extend this clock mechanism. A phaser is created and initialized to its first phase using the function `new`. The scope of a phaser is limited to the immediately enclosing `finish` statement. A task can be registered with zero or more phasers, using one of four registration modes: the first two are the traditional SIG and WAIT signal operations for producer-consumer synchronization; the SIG_WAIT mode implements barrier synchronization, while SIG_WAIT_SINGLE ensures, in addition, that its associated statement is executed by only one thread. As in X10, a `next` instruction is used to advance each phaser that this task is registered with to its next phase, in accordance with this task's registration mode, and waits on each phaser that task is registered with, with a WAIT submode. We illustrate the use of

clocks and phasers in Figure 8; note that they are not used in our Mandelbrot example, since a collective barrier based on the `finish` statement is sufficient.

Atomic Section When a thread enters an `atomic` statement, no other thread may enter it until the original thread terminates it.

Habanero-Java supports weak atomicity using the `isolated` $stmt$ primitive for mutual exclusion and isolation. The Habanero-Java implementation takes a single-lock approach to deal with isolated statements.

Data Distribution In order to distribute work across processors, X10 and HJ introduce a type called `place`. A place is an address space within which a task may run; different places may however refer to the same physical processor and share physical memory. The program address space is partitioned into logically distinct places. `Place.MAX_PLACES`, used in Figure 5, is the number of places available to a program.

4.4 OpenMP

OpenMP [3] is an application program interface providing a multi-threaded programming model for shared memory parallelism; it uses directives to extend sequential languages. A C OpenMP implementation of the Mandelbrot set is provided in Figure 6.

```
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) {
        //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 < (N*N) && k < maxiter);
        color = (ulong) ((k-1) * scale_color) + min_color;
#pragma omp critical
        {
          XSetForeground (display, gc, color);
          XDrawPoint (display, win, gc, col, row);
        }
}}}
```

Fig. 6: C OpenMP implementation of the Mandelbrot set

Task Parallelism OpenMP allows dynamic (`omp task`) and static (`omp section`) scheduling models. A task instance is generated each time a thread (the encountering thread) encounters a `omp task` directive. This task may either be scheduled immediately on the same thread or deferred and assigned to any thread in a thread team, which is the group of threads created when an `omp parallel` directive is encountered. The `omp sections` directive is a non-iterative work-sharing construct. It specifies that

the enclosed sections of code, declared with `omp section`, are to be divided among the threads in the team; these sections are independent blocks of code that the compiler can execute concurrently.

Synchronization OpenMP provides synchronization constructs that control the execution inside a team thread: `barrier` and `taskwait`. When a thread encounters a `barrier` directive, it waits until all other threads in the team reach the same point; the scope of a barrier region is the innermost enclosing parallel region. The `taskwait` construct is a restricted barrier that blocks the thread until all child tasks created since the beginning of the current task are completed. The `omp single` directive identifies code that must be run by only one thread.

Atomic Section The `critical` and `atomic` directives are used for identifying a section of code that must be executed by a single thread at a time. The `atomic` directive works faster than `critical`, since it only applies to single instructions, and can thus often benefit from hardware support. Our implementation of the Mandelbrot set in Figure 6 uses `critical`.

Data Distribution OpenMP variables are either global (`shared`) or local (`private`); see Figure 6 for examples. A shared variable refers to one unique block of storage for all threads in the team. A private variable refers to a different block of storage for each thread. More memory access modes exist, such as `firstprivate` or `lastprivate`, that may require communication or copy operations.

4.5 OpenCL

OpenCL (Open Computing Language) [2] is a standard for programming heterogeneous multiprocessor platforms where programs are divided into several parts: some called "the kernels" that execute on separate devices, e.g., GPUs, with their own memories and the others that execute on the host CPU. The main object in OpenCL is the command queue, which is used to submit work to a device by the enqueueing of OpenCL commands to be executed. An OpenCL implementation of the Mandelbrot set is provided in Figure 7.

Task Parallelism OpenCL provides the parallel construct `clEnqueueTask`, which enqueues a command requiring the execution of a kernel on a device by a work item (OpenCL thread). OpenCL uses two different models of execution of command queues: in-order, used for data parallelism, and out-of-order. In an out-of-order command queue, commands are executed as soon as possible, and no order is specified, except for wait and barrier events. We illustrate the out-of-order execution mechanism in Figure 7, but currently this is an optional feature and is thus not supported by many devices.

Synchronization OpenCL distinguishes between two types of synchronization: coarse and fine. Coarse grained synchronization, which deals with command queue operations, uses the construct `clEnqueueBarrier`, which defines a barrier synchronization point. Fine grained synchronization, which covers synchronization at the GPU function call granularity level, uses OpenCL events via `ClEnqueueWaitForEvents` calls.

Data transfers between the GPU memory and the host memory, via functions such as `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`, also induce synchronization between blocking or non-blocking communication commands. Events returned

```
__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<(N*N) && 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);
  }
```

Fig. 7: OpenCL implementation of the Mandelbrot set

by `clEnqueue` operations can be used to check if a non-blocking operation has completed.

Atomic Section Atomic operations are only supported on integer data, via functions such as `atom_add` or `atom_xchg`. Currently, these are only supported by some devices as part of an extension of the OpenCL standard. OpenCL lacks support for general atomic sections, thus the drawing function is executed by the host in Figure 7.

Data Distribution Each work item can either use (1) its private memory, (2) its local memory, which is shared between multiple work items, (3) its constant memory, which is closer to the processor than the `__global` memory, and thus much faster to access, although slower than `__local` memory, and (4) global memory, shared by all work items. Data is only accessible after being transferred from the host, using functions such as `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` that move data in and out of a device.

5 Discussion and Comparison

This section discusses the salient features of our surveyed languages. More specifically, we look at their design philosophy and the new concepts they introduce, how point-to-point synchronization is addressed in each of these languages, the various semantics of atomic sections and the data distribution issues. We end up summarizing the key features of all the languages covered in this paper.

Design Paradigms Our overview study, based on a single running example, namely the computation of the Mandelbrot set, is admittedly somewhat biased, since each language has been designed with a particular application framework in mind, which may, or may not, be well adapted to a given application. Cilk is well suited to deal with divide-and-conquer strategies, something not put into practice in our example. On the contrary, X10, Chapel and Habanero-Java are high-level Partitioned Global Address Space languages that offer abstract notions such as places and locales, which were put to good use in our example. OpenCL is a very low-level, verbose language that works across GPUs and CPUs; our example clearly illustrates that this approach is not providing much help here in terms of shrinking the semantic gap between specification and implementation. The OpenMP philosophy is to add compiler directives to parallelize parts of code on shared-memory machines; this helps programmers move incrementally from a sequential to a parallel implementation.

New Concepts Even though this paper does not address data parallelism per se, note that Cilk is the only language that does not provide special support for data parallelism; yet, spawned threads can be used inside loops to simulate SIMD processing. Also, Cilk adds a facility to support speculative parallelism, enabling spawned tasks abort operations via the `abort` statement. Habanero-Java introduces the `isolated` statement to specify the weak atomicity property. Phasers, in Habanero-Java, and clocks, in X10, are new high-level constructs for collective and point-to-point synchronization between varying sets of threads.

Point-to-Point Synchronization We illustrate the way our surveyed languages address the difficult issue of point-to-point synchronization via a simple example, a hide-and-seek game in Figure 8. X10 clocks or Habanero-Java phasers help express easily the different phases between threads. The notion of point-to-point synchronization cannot be expressed easily using OpenMP or Chapel. We were not able to implement this game using Cilk high-level synchronization primitives, since `sync`, the only synchronization construct, is a local barrier for recursive tasks: it synchronizes only threads spawned in the current procedure, and thus not the two searcher and hider tasks. As mentioned above, this is not surprising, given Cilk's approach to parallelism.

Atomic Section The semantics and implementations of the various proposals for dealing with atomicity are rather subtle.

Atomic operations, which apply to single instructions, can be efficiently implemented, e.g. in X10, using non-blocking techniques such as `compare-and-swap` instructions. In OpenMP, the atomic directive can be made to work faster than the critical directive, when atomic operations are replaced with processor commands such as GLSC [15]; therefore, it is better to use this directive when protecting shared memory during elementary operations. Atomic operations can be used to update different elements of a data structure (arrays, records) in parallel without using many explicit locks.

```
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();
 }
}
```

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

Fig. 8: A hide-and-seek game (X10, HJ, Cilk)

In the example of Figure 9, the updates of different elements of Array `x` are allowed to occur in parallel. General atomic sections, on the other hand, serialize the execution of updates to elements via one lock.

```
#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
}
```

Fig. 9: Example of an atomic directive in OpenMP

With the weak atomicity model of Habanero-Java, the `isolated` keyword is used instead of `atomic` to make explicit the fact that the construct supports weak rather than strong isolation. In Figure 10, Threads 1 and 2 may access to `ptr` simultaneously; since weakly atomic accesses are used, an atomic access to `temp->next` is not enforced.

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

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

Fig. 10: Data race on `ptr` with Habanero-Java

Data Distribution PGAS languages offer a compromise between the fine level of control of data placement provided by the message passing model and the simplicity of the shared memory model. However, the physical reality is that different PGAS portions, although logically distinct, may refer to the same physical processor and share physical memory. Practical performance might thus not be as good as expected.

Regarding the shared memory model, despite its simplicity of programming, programmers have scarce support for expressing data locality, which could help improve performance in many cases. Debugging is also difficult when data races occur.

Finally, the message passing memory model, where processors have no direct access to the memories of other processors, can be seen as the most general one, in which programmers can both specify data distribution and control locality. Shared memory (where there is only one processor managing the whole memory) and PGAS (where one assumes that each portion is located on a distinct processor) models can be seen as particular instances of the message passing model, when converting implicit write and read operations with explicit send/receive message passing constructs.

Summary Table We collect in Table 1 the main characteristics of each language addressed in this paper. Even though we have not discussed the issue of data parallelism in this paper, we nonetheless provide, for the interested reader, the main constructs used in each language to launch data parallel computations.

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

Table 1. Summary of parallel languages constructs

6 Conclusion

This paper presents, using the Mandelbrot set computation as a running example, an up-to-date comparative overview of six parallel programming language designs: Cilk, Chapel, X10, Habanero-Java, OpenMP and OpenCL. These languages are in current use, popular, offer rich and highly abstract functionalities, and most support both data and task parallel execution models. The paper describes how, in addition to data distribution and locality, the fundamentals of task parallel programming, namely task creation, collective and point-to-point synchronization and mutual exclusion are dealt with in these languages.

This paper can be of use to (1) programmers, by providing a taxonomy of parallel language designs useful when deciding which language is more appropriate for a given project, (2) language designers, by presenting design solutions already field-tested in previous languages, and (3) implementors of automatic program conversion tools, by helping them narrow down the issues that need to be tackled when dealing with parallel

execution and memory models. We used this case study as a basis for the design of SPIRE [14], a sequential to parallel intermediate representation extension that can be used to upgrade the intermediate representations used in compilation frameworks in order to represent task concepts in control-parallel languages and constructs; SPIRE is simple and generic enough to describe, to our knowledge, all parallel languages, even though the intricacies of the various existing synchronization models, exhibited by this study, require low-level representation support.

References

1. Message Passing Interface. `http://www-unix.mcs.anl.gov/mpi/index.html`.
2. OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems. `http://www.khronos.org/opencl`.
3. OpenMP Specifications. `http://www.openmp.org/blog/specifications/`.
4. The Mandelbrot Set. `http://warp.povusers.org/Mandelbrot/`.
5. *Cilk 5.4.6 Reference Manual*. Supercomputing Technologies Group, MIT Laboratory for Computer Science, `http://supertech.lcs.mit.edu/cilk`, 1998.
6. *Chapel Language Specification 0.796*. Cray Inc., 901 Fifth Avenue, Suite 1000, Seattle, WA 98164, October 21, 2010.
7. S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, 1996.
8. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.
9. V. Cavé, J. Zhao, and V. Sarkar. Habanero-Java: the New Adventures of Old X10. In *9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, August 2011.
10. B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl*, 21:291–312, August 2007.
11. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. *SIGPLAN Not.*, 40:519–538, October 2005.
12. E. Cuevas, A. Garcia, F. J. J.Fernandez, R. J. Gadea, and J. Cordon. Importance of Simulations for Nuclear and Aeronautical Inspections with Ultrasonic and Eddy Current Testing. Simulation in NDT, Online Workshop in `www.ndt.net`, September 2010.
13. J. B. Dennis, G. R. Gao, and K. W. Todd. Modeling The Weather With a Data Flow Supercomputer. *IEEE Trans. Computers*, pages 592–603, 1984.
14. D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin. SPIRE: A Sequential to Parallel Intermediate Representation Extension. Technical Report CRI/A-487 (Submitted to CGO'13), MINES ParisTech, 2012.
15. S. Kumar, D. Kim, M. Smelyanskiy, Y.-K. Chen, J. Chhugani, C. J. Hughes, C. Kim, V. W. Lee, and A. D. Nguyen. Atomic Vector Operations on Chip Multiprocessors. *SIGARCH Comput. Archit. News*, 36(3):441–452, June 2008.
16. J. Larus and C. Kozyrakis. Transactional Memory. *Commun. ACM*, 51:80–88, July 2008.
17. D. A. Padua, editor. *Encyclopedia of Parallel Computing*. Springer, 2011.
18. V. Sarkar. Synchronization Using Counting Semaphores. In *ICS'88*, pages 627–637, 1988.
19. J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: A Unified Deadlock-Free Construct for Collective and Point-To-Point Synchronization. In *ICS'08*, pages 277–288, New York, NY, USA, 2008. ACM.