Programming paradigms using PGAS-based languages

Marc Tajchman

CEA - DEN/DM2S/SFME/LGLS

Monday, June 9th 2011

Outline

energie atomique • energies alternatives

General considerations
- PGAS definition
- MPI and multithreads models
- PGAS models

Langages
- UPC
- Co-Array Fortran
- X10
- Chapel
- XcalableMP

Outline

energie atomique • energies alternatives

PGAS

PGAS (Partitioned Global Address Space) is a parallel programming model.

This model defines:

- execution contexts, with separated memory spaces,

  Execution context ≈ MPI process

- threads running inside an execution context.

  PGAS thread ≈ OpenMP thread, pthread, ...
  (PGAS threads are often light threads)

PGAS

- direct access from one context to data managed by another context,

  Data structures can be distributed in several contexts, with a global addressing scheme (more or less transparent, depending on the programming language).

- higher-level operations on distributed data structures, *e.g. "for each"-type operations on arrays*

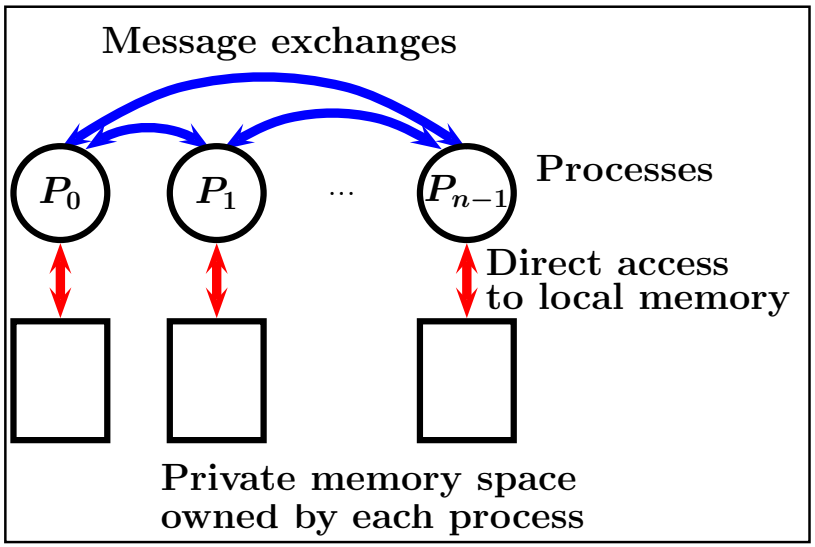
  These operations may create threads implicitely (e.g. on multicore computing nodes), and do implicit data copy between contexts. The available set depends on the programming language.

Outline

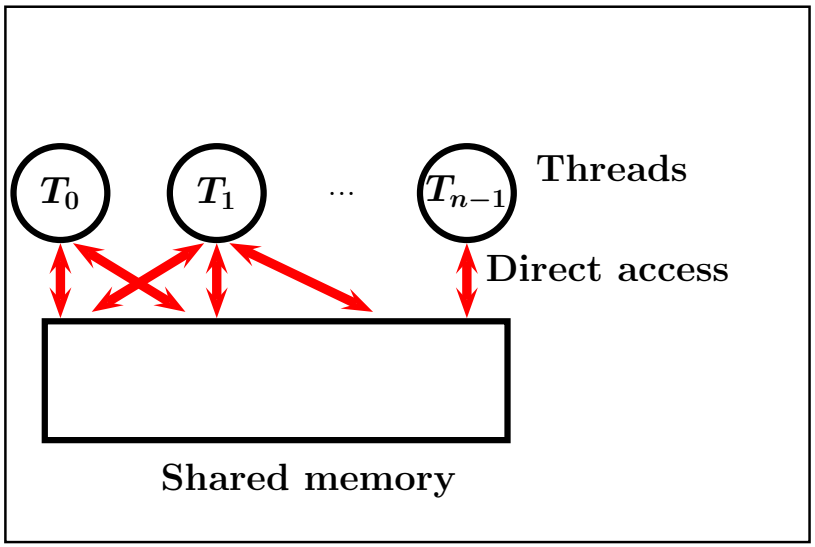
"Standard Models"

energie atomique • energies alternatives

"Message passing" model (e.g. MPI)

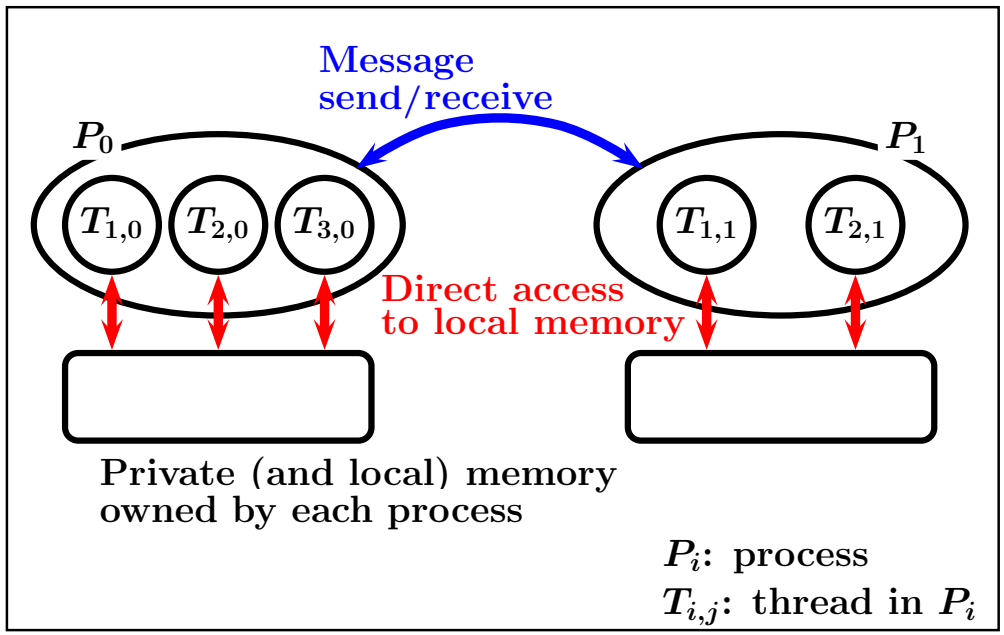


"Shared memory" model (e.g. OpenMP)



Hybrid programming (e.g. MPI-OpenMP) :

- One or more threads in each process.
- A thread has direct access to the private memory owned by its process.
- Inter-processes data communications handled by messages.



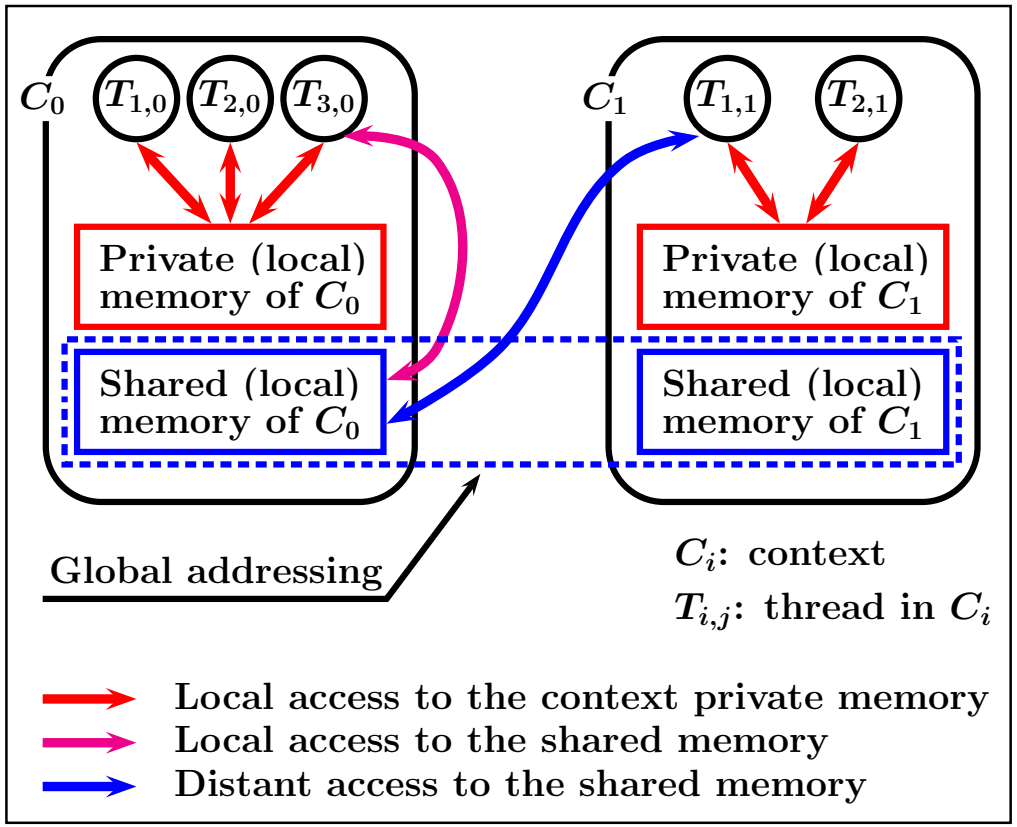
Outline

energie atomique • energies alternatives

PGAS: Execution and memory models

Execution model depends on the language (see next chapter).

Memory model:



PGAS: Execution and memory models

Distant memory accesses are (or should be):

- of RDMA-type (remote direct memory access),
- handled by one-sided communication functions (like `MPI_Put, MPI_Get` in MPI middleware).

So, PGAS models need efficient implementation of these operations.

That's why PGAS implementations are typically build on a few low-level communication layers, like GASNet or MPI-LAPI (on IBM machines).

Notion of affinity

PGAS models consider several memory access types, by increasing speed:

- shared memory location, on a different context,
- shared memory location, on the same context,
- private memory location, on the same context.

⇒ **notion of affinity**:

logical association between shared data and contexts. Each element of shared data storage has affinity to exactly one context.

⇒ PGAS languages propose mechanisms to take a better account of affinity

i.e. to distribute data and threads to perform as many local accesses as possible, instead of distant accesses.

Outline

energie atomique • energies alternatives

Languages

Several PGAS programming environments exist (language definition + compilation/execution tools) :

- UPC (Unified Parallel C), a superset of C
- CAF (Co-Array Fortran), syntax based on fortran 95
- Titanium , a superset of java
- X10, syntax based on java
- Chapel, new language (various influences)
- XcalableMP, set of pragma's added to C/C++/fortran

Compilers = "Intermediate source" front-end generators + C/C++/fortran back-end compiler.

Intermediate source code generation in C (Chapel, UPC, Titanium, XcalableMP), C++ (X10), fortran (CAF, XcalableMP), or java (X10).

Languages

energie atomique • energies alternatives

Remote communications and data distribution handled by external tools/libraries :

- MPI (proposed by most implementations)
- GASNet (proposed by most implementations)

  `http://gasnet.cs.berkeley.edu`
- OpenSHMEM

  `http://www2.cs.uh.edu/~hpctools/research/OpenSHMEM`
- GPI

  `http://www.gpi-site.com`
- ...

Outline

energie atomique • energies alternatives

Langage UPC

UPC (**`http://upc.gwu.edu`**) is a superset of the C language. It's one of the first languages that use a PGAS model, and also one of the most stable.

UPC extends the C norm with the following features:

- a parallel execution model of SPMD type,
- distributed data structures with a global addressing scheme, and static or dynamic allocation
- operators on these structures, with affinity control,
- copy operators between private, local shared, and distant shared memories,
- 2 levels of memory coherence checking (strict for computation safety and relaxed for performance),

UPC proposes only one level of task parallelism (only processes, no threads).

Langage UPC

Several "open-source" implementations exist, the most active are:

- Berkeley UPC (v 2.12.2, may 2011),

  `http://upc.lbl.gov`

- GCC/UPC (v 4.5.1.2, october 2010),

  `http://www.gccupc.org`

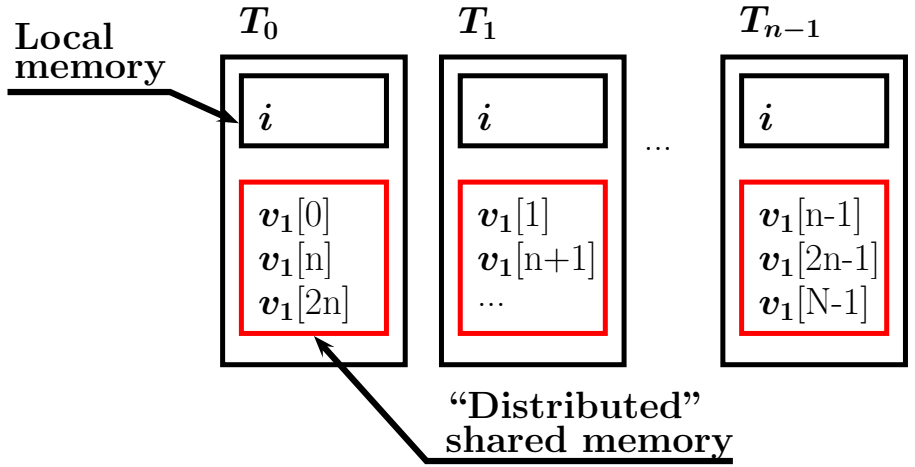
Several US computer manufacturers propose UPC compilers : IBM, HP, Cray

(there was apparently some incentive from the US administration to provide a UPC compiler along with C/C++/fortran compilers for new machines).

UPC Example (1)

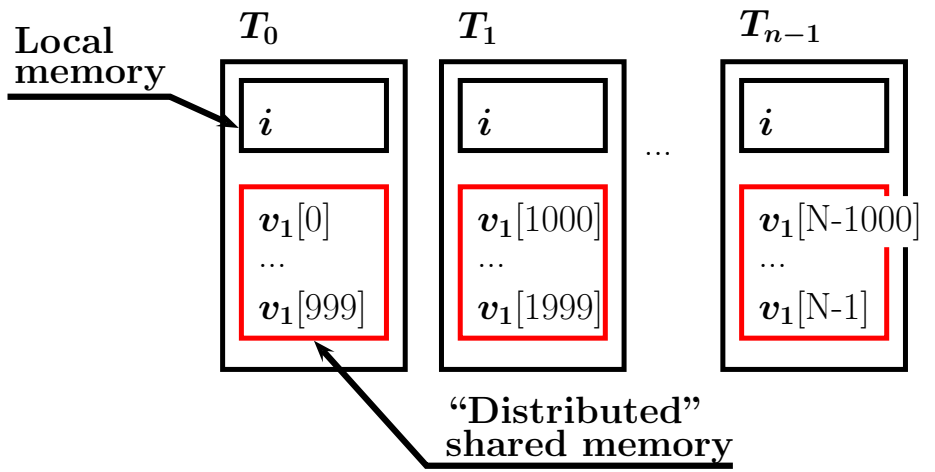
A (static) distributed data structure can be defined by:

```
1  #define N 1000*THREADS
2  int i;
3  shared int v1(N);
```



or, with a different distribution:

```
1  #define N 1000*THREADS
2  int i;
3  shared (1000) int v1(N);
```



UPC Example (1^a)

Definition and use of distributed vectors (1^{st} version):

```
1  #include <upc.h>
2  #define N 10000*THREADS
3
4  shared int v1(N), v2(N), v3(N);
5  int main()
6  {
7      int i;
8      for(i=1; i<N-1; i++)
9          v3(i)=0.5*(v1(i+1)-v1(i-1))+v2(i);
10
11     upc_barrier;
12     return 0;
13 }
```

Test with 2 processes (on 2 different machines):

793,1 s (10000 loops)

UPC Example (1^b)

Definition and use of distributed vectors (2^{nd} version, using affinity information):

```
1  #include <upc_relaxed.h>
2  #define N 10000*THREADS
3
4  shared int v1(N), v2(N), v3(N);
5  int main()
6  {
7      int i;
8      for(i=0; i<N; i++)
9          if (MYTHREAD == upc_threadof(&(v3(i))))
10             v3(i)=0.5*(v1(i+1)-v1(i-1))+v2(i);
11     upc_barrier;
12     return 0;
13 }
```

Test with 2 processes (on 2 different machines):

307,0 s (10000 loops)

UPC Example (1/c)

Definition and use of distributed vectors (3rd version, using an "upc loop"):

```
1  #include <upc_relaxed.h>
2  #define N 10000*THREADS
3
4  shared int v1(N), v2(N), v3(N);
5  int main()
6  {
7      int i;
8      upc_forall(i=0; i<N; i++; &(v3(i)))
9          v3(i)=0.5*(v1(i+1)-v1(i-1))+v2(i);
10
11     upc_barrier;
12     return 0;
13 }
```

Test with 2 processes (on 2 different machines):

301,5 s (10000 loops)

UPC Example (1^d)

Definition and use of distributed vectors (4^{th} version, using a different distribution):

```
1  #include <upc_relaxed.h>
2  #define N 10000*THREADS
3
4  shared (1000) int v1(N), v2(N), v3(N);
5  int main()
6  {
7      int i;
8      upc_forall(i=0; i<N; i++; &(v3(i)))
9          v3(i)=0.5*(v1(i+1)-v1(i-1))+v2(i);
10
11     upc_barrier;
12     return 0;
13 }
```

Test with 2 processes (on 2 different machines):

13,7 s (10000 loops)

Remote data access optimization

Distant accesses imply data (transparent) transferts between processes.

To improve the efficiency, UPC proposes a set of bloc-copy functions between:

- shared memories of 2 different processes: `upc_memcpy`,
- private memory of one process, and shared memory of the same or another process: `upc_memget` and `upc_memput`.

With these operators, the code will be more efficient, but may be more complicated to write.

Sample Comparison of data accesses types

energie atomique • energies alternatives

Extract of the upc test code:

```
1  #define N  10000*THREADS
2  #define M  10000
3  #define NLocal N/THREADS
4  #define NLast N-1
5  #define NDummy 0
6
7  shared (1000) int v(N);
8  int * vLocal = (int *) malloc(NLocal * sizeof(int));
9
10 for (j=0; j<M; j++)
11   for(i=0; i<NLocal; i++) vLocal(i+NDummy) += 1;
12
13 for (j=0; j<M; j++)
14   upc_forall(i=0; i<N; i++; i) v(i+NDummy) += 1;
15
16 for (j=0; j<M; j++)
17   upc_forall(i=0; i<N; i++; i) v(NLast-i) += 1;
```

Sample Comparison of data accesses types

Running times obtained with Berkeley UPC (similar results with GCCUPC)

On a 32-core (8×4) machine with shared memory:

Memory type	n^o of threads at compile time	n^o of threads at run time
local private	0.085 s	0.088 s
local shared	2.43 s	1.96 s
distant shared	44.0 s	18.2 s

On a 2-core machine (this laptop):

Memory type	n^o of threads at compile time	n^o of threads at run time
local private	0.071 s	0.067 s
local shared	1.95 s	1.09 s
distant shared	2.97 s	1.20 s

Expect more differences on a distributed memory machine.

Outline

energie atomique • energies alternatives

Co-Array Fortran

Co-Array Fortran (`http://www.co-array.org`) is an extension of fortran95. Fortran 2008 norm includes some of the co-arrays features.

Co-Array Fortran provides:

- an explicit parallel execution model of SPMD-type,

  Co-Array Fortran use the name of images for processes.

- distributed arrays (co-array) with transparent access to coefficients,
- the extension of fortran matrix operations to co-array's,
- etc.

Like in UPC, there is only one level of parallelism in Co-Array fortran.

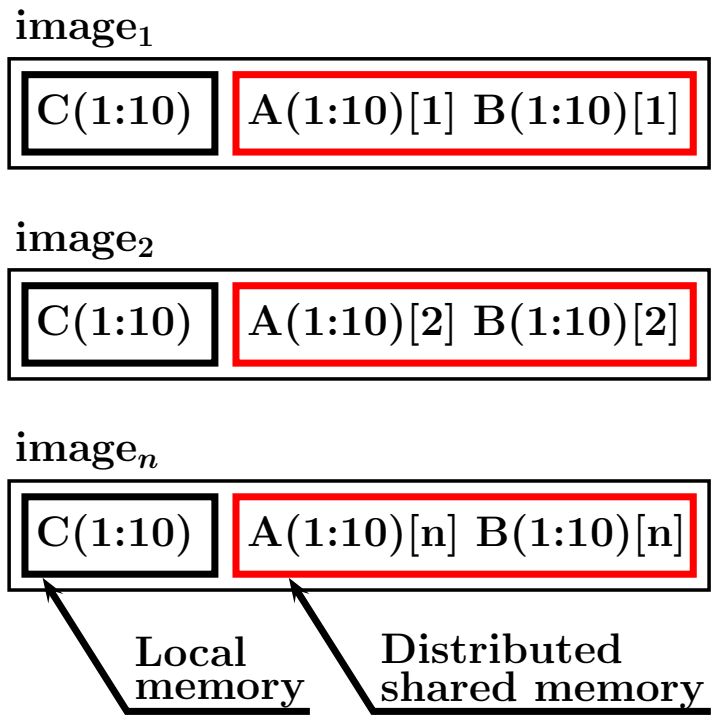
Co-Array Fortran

There are now relatively few implementations of Co-Array Fortran.

- Some commercial compilers provides partial versions of co-arrays (IBM CoArray Fortran, Intel Fortran Compiler XE 2011, etc).
- The only (as far as I know) open-source Co-Array fortran compilers are in development stage: a compiler from Rice University, or 4.6 and (experimental) 4.7 versions of GNU's gfortran.

Example in Co-Array Fortran


```
integer,codimension(*),dimension(10) :: A,B
integer size, rank, C(10)
size = num_images()
rank = this_image()

do  i=1,10
 A(i) = rank*10 + i
end do

if (rank .eq. 1) then
  do i=1,10
    B(i) = size*10 + i
  end do
end if

sync images(*)

if  (rank .eq. size) then
  A(2:9)(1) =A(2:9)
end if
```



Outline

energie atomique • energies alternatives

X10 language

X10 (`http://x10.codehaus.org`) is a language defined and developped at IBM Research. It's the IBM proposal to DARPA's HPCS program (High Productivity Computer System). Development is very active (new version every 2-3 months).

A context (resp. thread) is called a place (resp. activity) in X10.

X10 main features (for parallel programing):

- a specific execution model:

  an initial activity starts at place 0, from that activity the user can launch "child" activities on the same or other places,

- tasks parallelism

  activities are synchronous or asynchronous, syncronization barriers can be activated between activities (not necessarily on the same place)

X10 language

- data parallelism

  data can be distributed on a (sub)set of places (see examples)

- low-level operators:

  interaction between data and task parallelism can be specified very precisely by the programmer

X10 : data parallelism

To define a distributed array, one proceeds in 3 steps, building:

- a region (set of points or valid indexes):

```
R:Region(2) = (0..n)*(0..n);
```

- a distribution (partition scheme between places)

```
D:Dist(2) = Dist.makeBlock(R, 0);
```

- the array itself:

```
u:DistArray(double)(2) =
    DistArray.make(double)(D);
```

To read/write a coefficient in a distributed array:

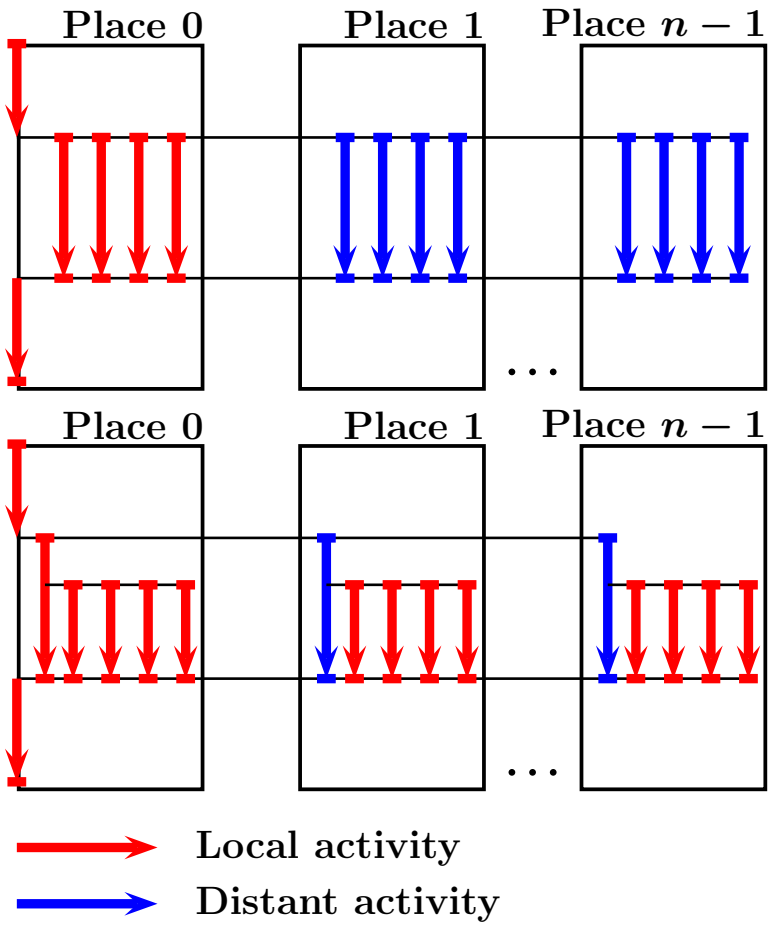
```
at (A.dist(2,2))
    A(2,2) = (at(A.dist(3,0)) A(3,0)) + 4.5;

at (A.dist(2,2))
    A(2,2) = at(A.dist(3,0)) (A(3,0) + 4.5);
```

X10 : task parallelism

At first, one activity (thread) only start in place 0.
Then this activity can start other activities in the same or other places. These activities can themselves launch local or remote activities.

```
finish
for (p in u)
    async at (u.dist(p))
        S(u(p));
```

```
finish
for (p in D.places())
  async at (p)
    for (q in u.dist|here) {
      async S(u(q));
    }
```



X10 : task parallelism


```
var u : DistArray(double)(3);

n : int = 100;
R : Region(3) = (1..n)*(1..n)*(1..n);
D : Dist(3) = Dist.makeBlock(R, 0);
u = DistArray.make(double)(D, (p:Point) => 0.0);
```

```
// 1 level of threads
finish
for (p in u)
   async at (u.dist(p))
      S(u(p));
```

```
// 2 levels of threads
finish
for (pl in u.dist.places())
   async at (pl)
      for (q in D | here)
         S(u(q));
```

On a 2-core machine (this laptop) with 2 places (processes):

level of threads	
1	9.46 s
2	0.665 s

X10 : some guideline for performance

X10 is a very rich language, advanced features are very powerful, but add additional execution time cost.
So, as (non definitive) guidelines for performance:

- try to launch as many local activities as possible (vs. distant ones)
- try to put as many barriers between colocalized activities as possible (vs barriers between activities on different places).
- activities are light threads but their creation take some time, so put enough work into each activities
- if you know that a region is cartesian, specify it explicitely (for the current version, the compiler cannot always detect it)
- ...

X10 : Laplace Equation (version 1)

energie atomique • energies alternatives

```
finish
  for ((i,j) in u.dist) {
    async at (u.dist(i,j))
        v(i, j) = (1-4*lambda) * u(i,j) + lambda *
        ( (at (u.dist(i+1,j)) u(i+1, j)) +
          (at (u.dist(i-1,j)) u(i-1, j)) +
          (at (u.dist(i,j-1)) u(i, j-1)) +
          (at (u.dist(i,j+1)) u(i, j+1)));
}
```

Lots of distant activities + scalar remote transferts : performs very badly

X10 : Laplace Equation (version 2)

energie atomique • energies alternatives

```
finish
  for (p in u.dist.places())
    async at (p)
      for ((i,j) in u.dist|here) {
        async
            v(i,j) = (1-4*lambda)*u(i,j) + lambda*
               ((at (u.dist(i+1,j)) u(i+1, j)) +
                (at (u.dist(i-1,j)) u(i-1, j)) +
                (at (u.dist(i,j-1)) u(i, j-1)) +
                (at (u.dist(i,j+1)) u(i, j+1)));
      }
```

A few distant activities + many local activities (and these activities do very little work) + scalar remote transferts : performs badly

X10 : Laplace Equation (version 3)


```
finish
for (p in u.dist.places()) async at (p) {

localRegion : Region(2) = u.dist | here;
innerRegion : Region(2)
    = (localRegion.min(0)+1 .. localRegion.max(0)-1) *
      (localRegion.min(1)+1 .. localRegion.max(1)-1);

boundaryRegion : new Array(Region(2))(4);
boundaryRegion(0)
    = (localRegion.min(0)   .. localRegion.min(0)) *
      (localRegion.min(1)+1 .. localRegion.max(1)-1)
...
async
  for ((i,j) in innerRegion)
    async
      v(i, j) = (1-4*lambda) * u(i,j) + lambda *
        ( u(i+1, j) + u(i-1, j) + u(i, j-1) + u(i, j+1));
```

X10 : Laplace Equation (version 3, cont'd)

energie atomique • energies alternatives

```
async
    for ((i,j) in boundaryRegion(0))
        v(i, j) = (1-4*lambda) * u(i,j) + lambda *
         ( u(i+1, j) +
           (at (u.dist(i-1,j)) u(i-1, j)) +
           u(i, j-1) +
           u(i, j+1));
async
    for ((i,j) in boundaryRegion(1))
        v(i, j) = (1-4*lambda) * u(i,j) + lambda *
           (at (u.dist(i+1,j)) u(i+1, j)) +
           u(i-1, j) +
           u(i, j-1) +
           u(i, j+1));
...
```

Much less activities : "false remote" activities dropped, scalar tranferts: not optimal but performs better

X10 : Laplace Equation (version 4)

Code extract for the interfaces between places:

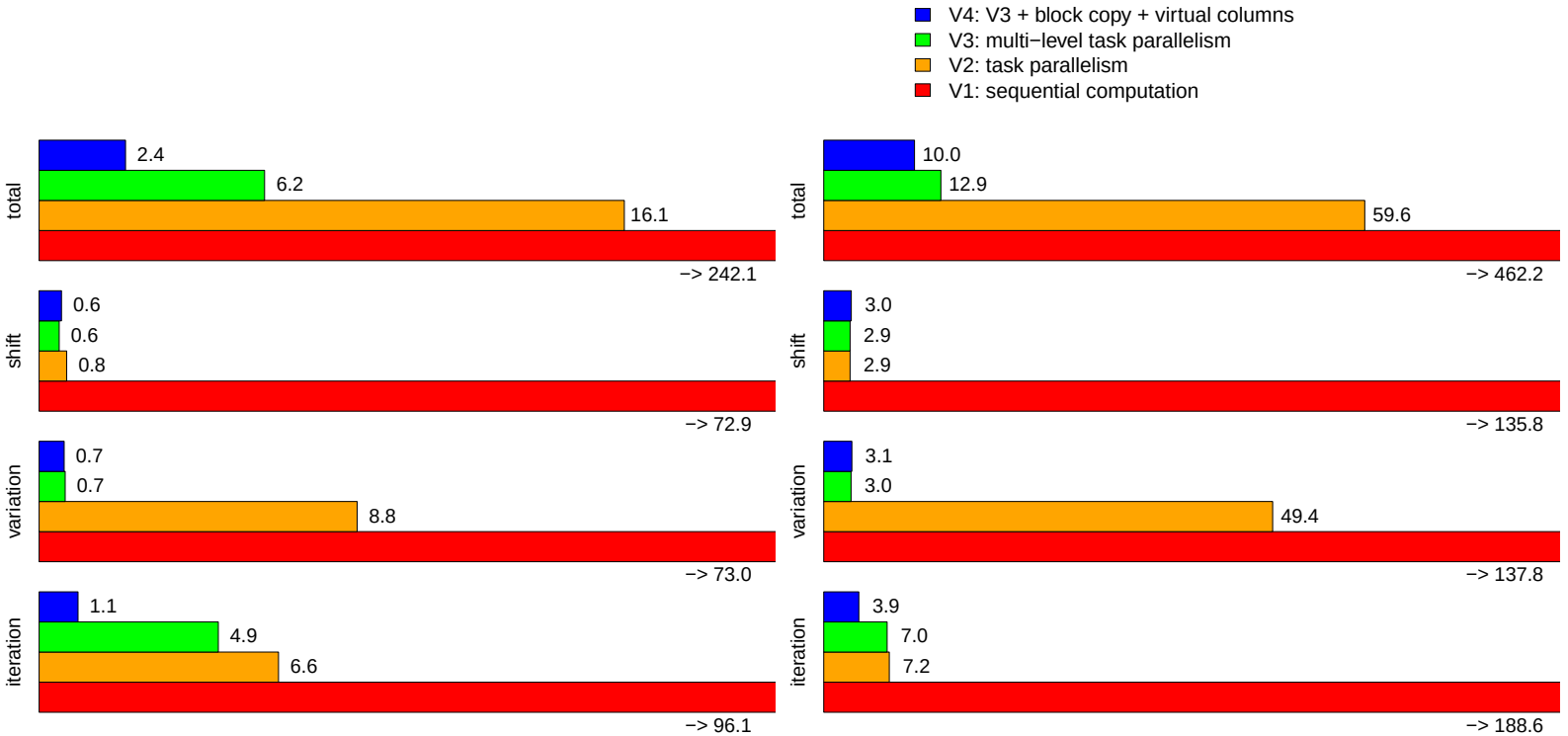
```
externalRegion = new Array(Region(2))(4);

externalRegion(0)
    = (localRegion.min(0)-1 .. localRegion.min(0)-1) *
      (localRegion.min(1)+1 .. localRegion.max(1)-1)
async {
   w : Array(2) = at (p) u(externalRegion(0));
   for ((i,j) in boundaryRegion(0))
     v(i, j) = (1-4*lambda) * u(i,j) + lambda *
       (u(i+1, j) + w(i-1, j) + u(i, j-1) + u(i, j+1));
}
...
```

Version 4 = Version 3 + vector tranferts: much better

X10 : Laplace Equation (version 4)

energie atomique • energies alternatives

Comparison of several results (old tests, must be updated)



8×4 cores,
shared memory

8-nodes, 4-cores,
dist. memory

Outline

energie atomique • energies alternatives

Langage Chapel

Chapel (Cascade High Productivity Language, `http://chapel.cray.com/index.html`) is a language designed by Cray, and selected by the HPCS project of DARPA like X10 of IBM.
It's a language built from scratch, with various influences.

Contexts (resp. threads) are called locales (resp. tasks) in Chapel. The main features are:

- a similar execution model as X10 (a unique thread starts in the first context, it can create other threads in the same of other contexts),
- distributed data structures
- tasks parallelism

  Several levels of abstraction : global operations (`forall`, `reduce`, etc.), finer control of tasks (`begin`, `cobegin`, etc.)

- simple language to learn

Data Parallelism

Distributed data definition in 3 steps, one has to build:

- a domain (set of valid indexes),
- a distribution (partition of a domain between `locales`),
- the array itself on this distribution.

Example:

```
use BlockDist;
...
  var D: domain(1) = (1..n) dmapped Block((1..n));
  var Din: domain(1) = (2..n-1);
  var a, b, f: (D) real;
...
```

Example (global operations):

```
do {
  forall i in Din
    b(i) = h2*f(i)+(a(i-1)+a(i+1))/2;

  diff= +reduce
    forall i in D do abs(b(i)-a(i));

  forall i in Din
    a(i) = b(i);
} while (diff > 1e-5);
```

Example (finer control of tasks):

```
cobegin {
  functionA();
  functionB();
  on Locales(2) functionC();
}
```

Outline

energie atomique • energies alternatives

XcalableMP

XcalableMP comes from the University of Tsukuba (Japan). It can be seen as an extension of C or fortran, using pragma's to express parallel and PGAS concepts (task parallelism and data distribution).

> *pragma's can be deactivated at compile-time, and the C/fortran source should be a valid sequential code (as in OpenMP).*

XcalableMP is a very new langage (first version available at the end of 2010). It's influenced by the HPF (high-performance fortran) and co-array fortran experiences.

As in X10 and Chapel, data distribution is done in 3 steps:

- defining a region (#pragma xmp template),
- a partition on contexts (#pragma xmp distribute),
- data array alignment (#pragma xmp align)

XcalableMP example

energie atomique • energies alternatives

```
int array(YMAX)(XMAX);
#pragma xmp nodes p(*)
#pragma xmp template t(YMAX)
#pragma xmp distribute t(block) on p
#pragma xmp align array(i)(*) with t(i)

main(){
  int i, j, res;
  res = 0;
#pragma xmp loop on t(i) reduction(+:res)
  for(i = 0; i < YMAX; i++)
    for(j = 0; j < XMAX; j++) {
      array(i)(j) = func(i, j);
      res += array(i)(j);
    }
}
```