DE LA RECHERCHE À L'INDUSTRIE

www.cea.fr

PGAS models and programming languages

Marc Tajchman

CEA/DEN/DM2S/STMF/LMES

Nov. 10, 2015

Outline

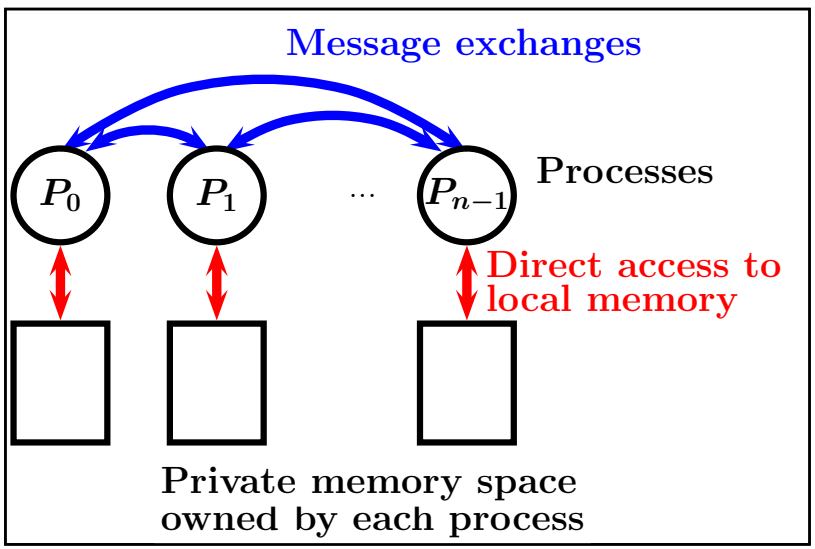
Outline

Main characteristics of the programming models considered here:

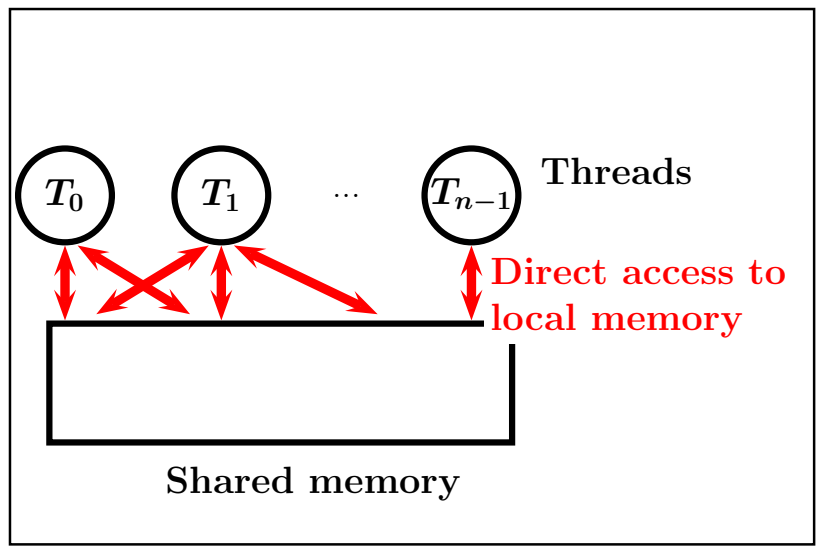
- how to define data placement and access pattern
- how to organize the computations (execution model)

Standard Models

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

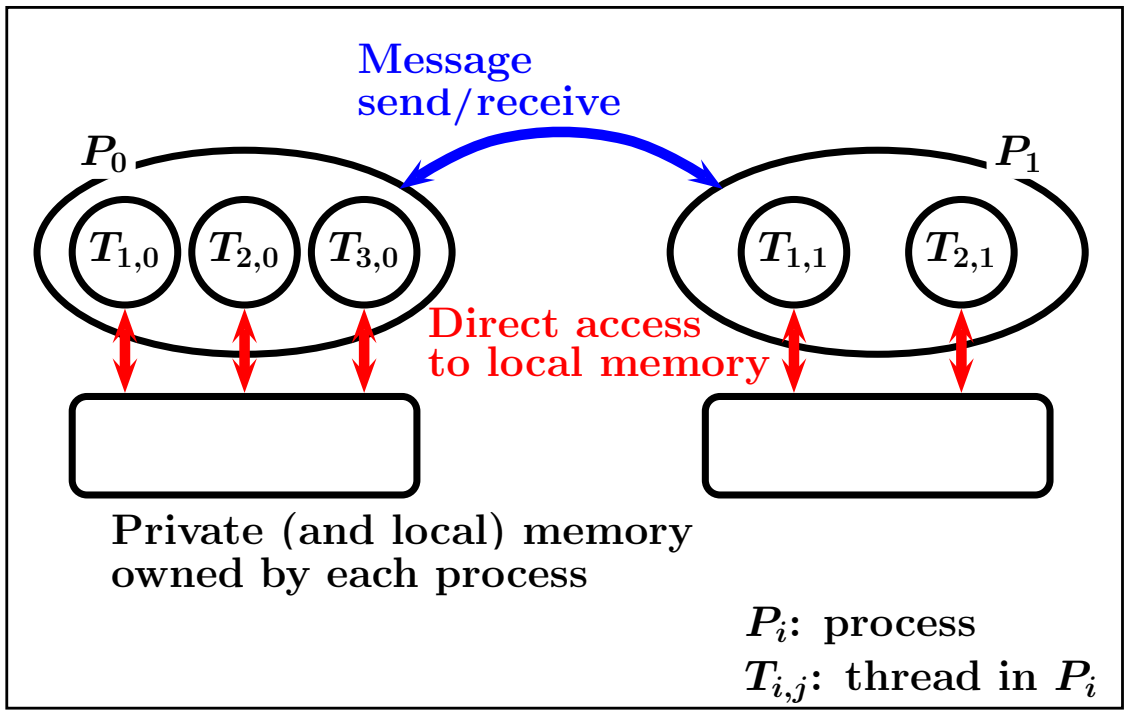


"Shared memory" model (e.g. OpenMP)



Standard Models - 2

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



PGAS

PGAS (Partitioned Global Address Space) is a parallel programming model, where you can use:

- multiple execution contexts, with one execution thread, and separated memory spaces,

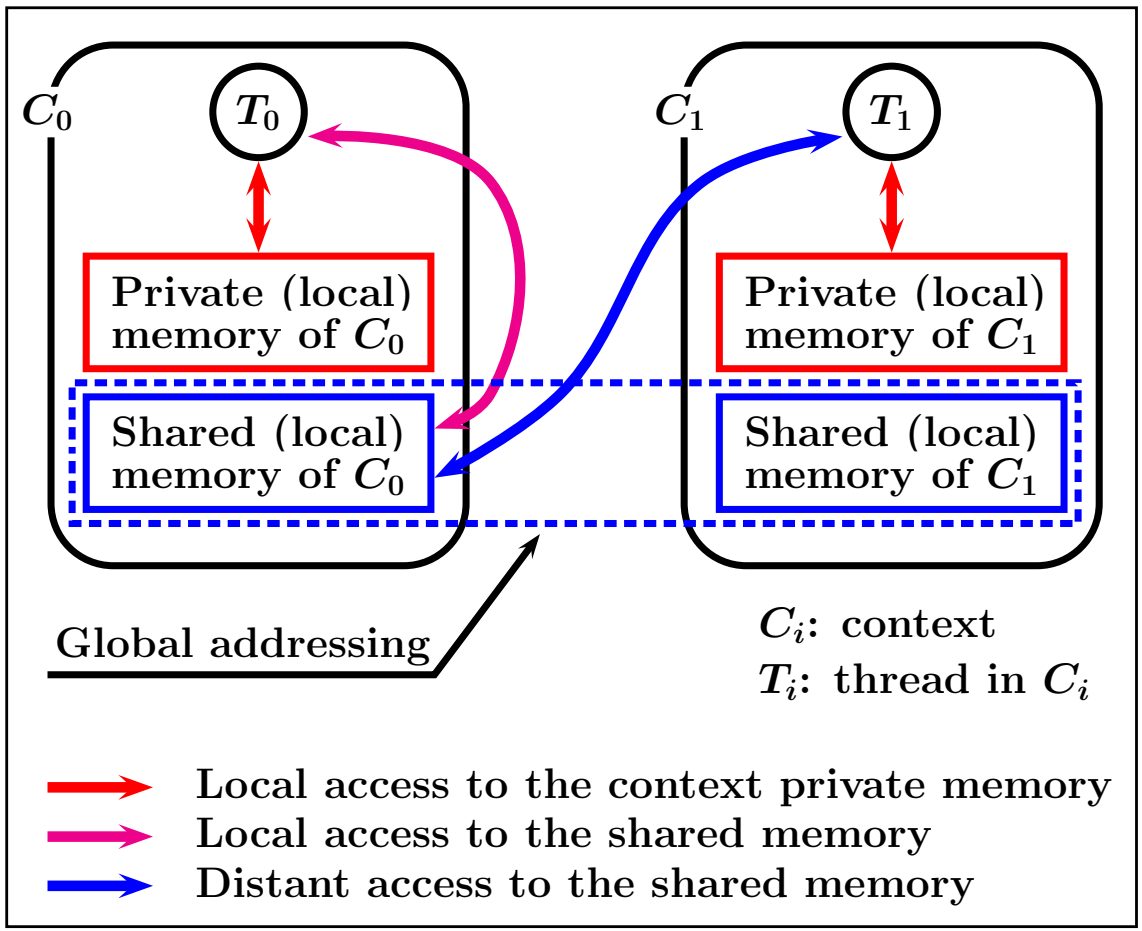
  Execution context ≈ MPI process

  If you want to use "hybrid programming", you need to add something else (e.g. PGAS + OpenMP).

- direct access from one context to (some) 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).

Data placement and accesses - Execution model depends on the language (see later)



APGAS

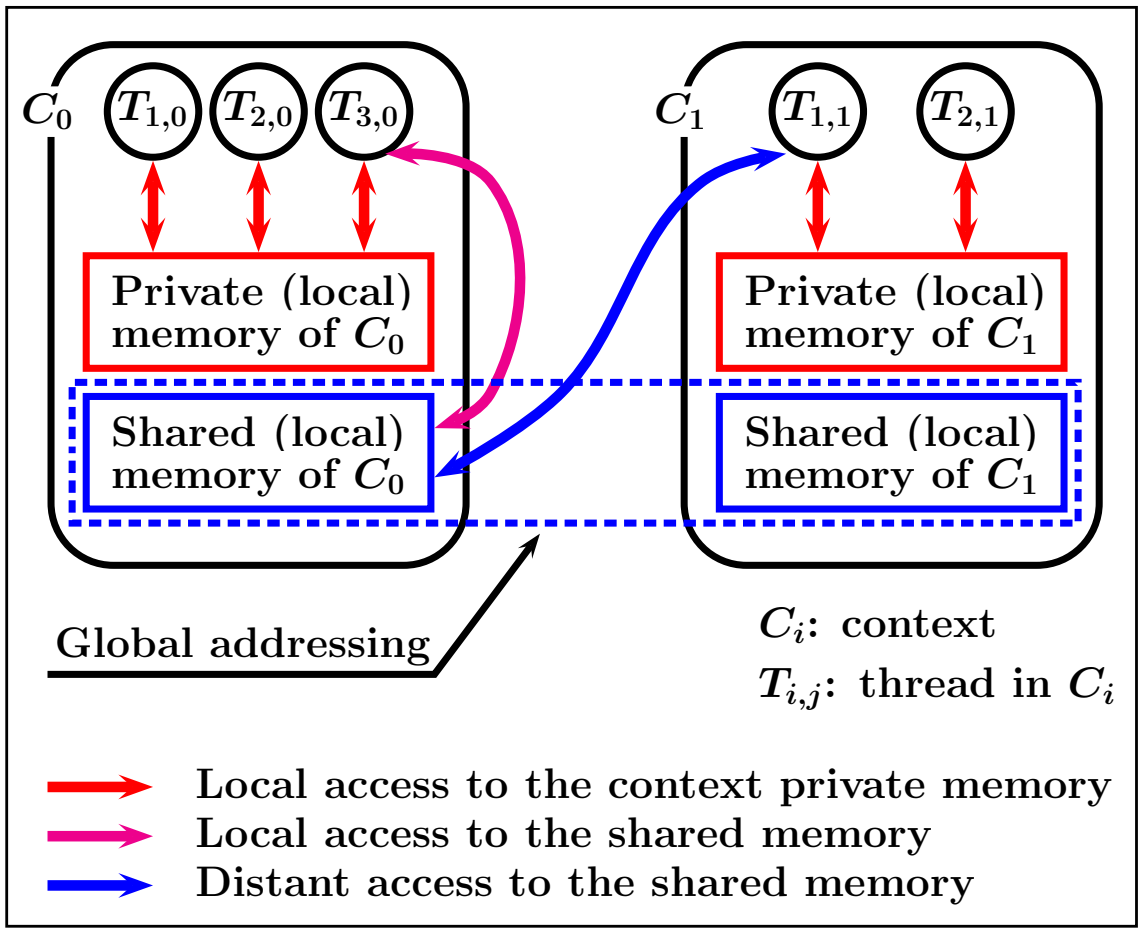
APGAS (Asynchronous Partitioned Global Address Space) adds to the PGAS programming model:

- the possibility to run more then one "thread" inside an execution context

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

  A thread can be started from the same context or (remotely) from another context.
  This operation may implicitly copy data between contexts.

APGAS - 2


C_0
$T_{1,0}$
$T_{2,0}$
$T_{3,0}$
C_1
$T_{1,1}$
$T_{2,1}$
Private (local) memory of C_0
Private (local) memory of C_1
Shared (local) memory of C_0
Shared (local) memory of C_1
Global addressing
C_i: context
$T_{i,j}$: thread in C_i
Local access to the context private memory
Local access to the shared memory
Distant access to the shared memory

(A)PGAS execution model

Each PGAS implementation may offer a more or less complete set of high-level operations, such that:

- creating and controlling a second level of parallelism (i.e. threads into processes)
- parallel control commands (e.g. "`for all`" loops)
- ...

In the next slides, we will show, for each presented tool, its main high-level features.

Remote memory accesses

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.

$\Rightarrow$ **notion of affinity**:

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

$\Rightarrow$ 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

PGAS tools and languages

Several types of tools are available to program using the PGAS model:

- libraries to use in a standard parallel program
- extensions to existing programming languages
- dedicated programming languages

Outline

PGAS libraries

These libraries contains a set of functions that implement distributed arrays and a more or less transparent global addressing scheme.

Examples :

- GlobalArrays (`http://hpc.pnl.gov/globalarrays`)
- OpenSHMEM (`http://openshmem.org/site`)
- GASPI (`http://www.gaspi.de/en/project.html`
- GasNet (`http://gasnet.lbl.gov`)

Pro.: You can use these libraries in a standard (legacy) MPI parallel program,

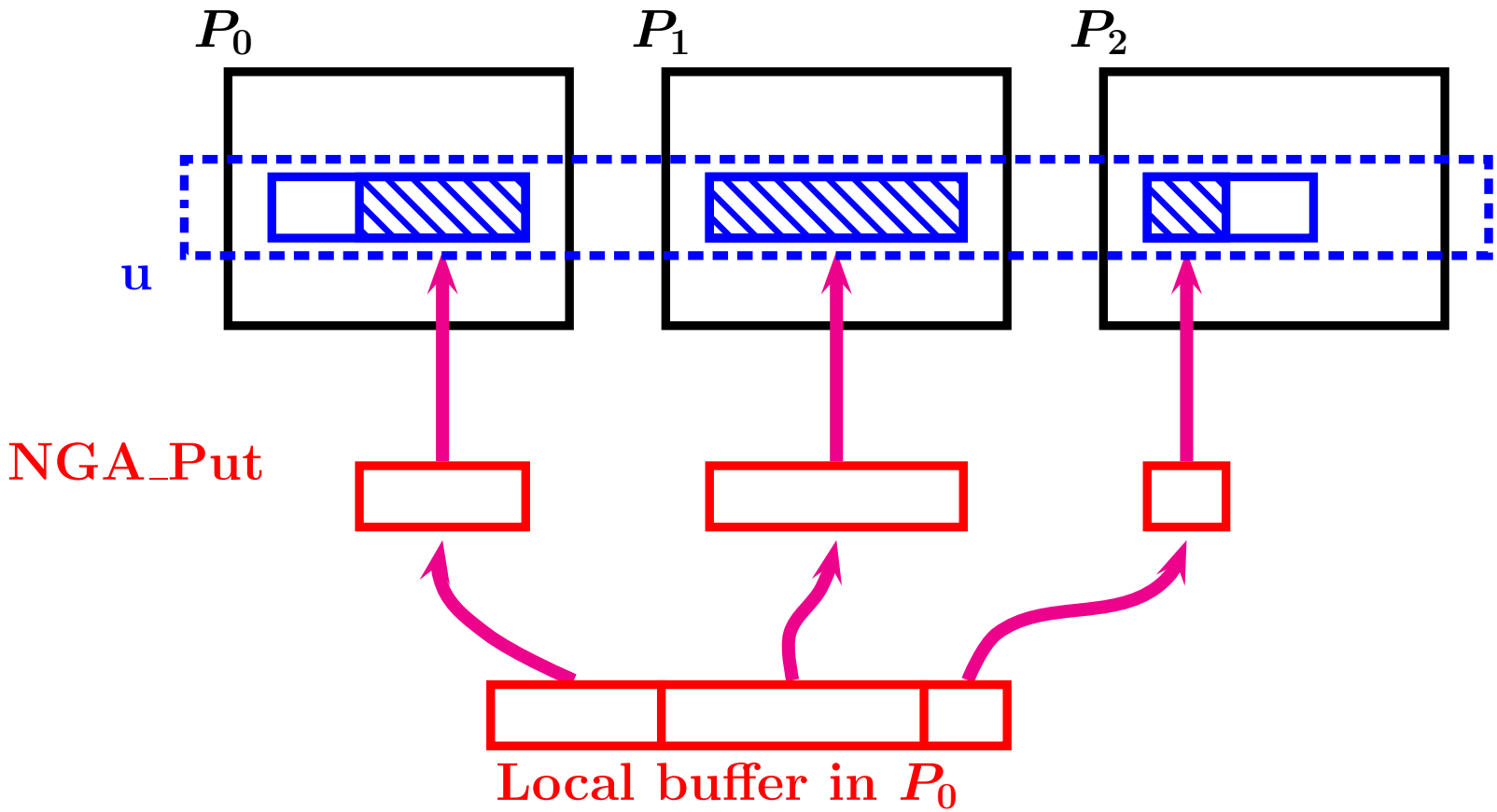
Cons.: You have to use a separate tool to define the execution model (e.g MPI).

Simple example using GlobalArrays

```
 1  double buffer[...];
 2
 3  MPI_Init(&argc,&argv);
 4  GA_Initialize();
 5
 6  u = NGA_Create(C_DBL, 1, &dim, "U", &chunk);
 7
 8  rank = GA_Nodeid();
 9  if (rank == 0)
10    NGA_Put(u, &imin, &imax, buffer, NULL);
11
12  GA_Sync();
13  GA_Print(u);
14  GA_Destroy(u);
15
16  GA_Terminate();
17  MPI_Finalize();
```

(click here for a working example)

Put operation in the GlobalArrays example

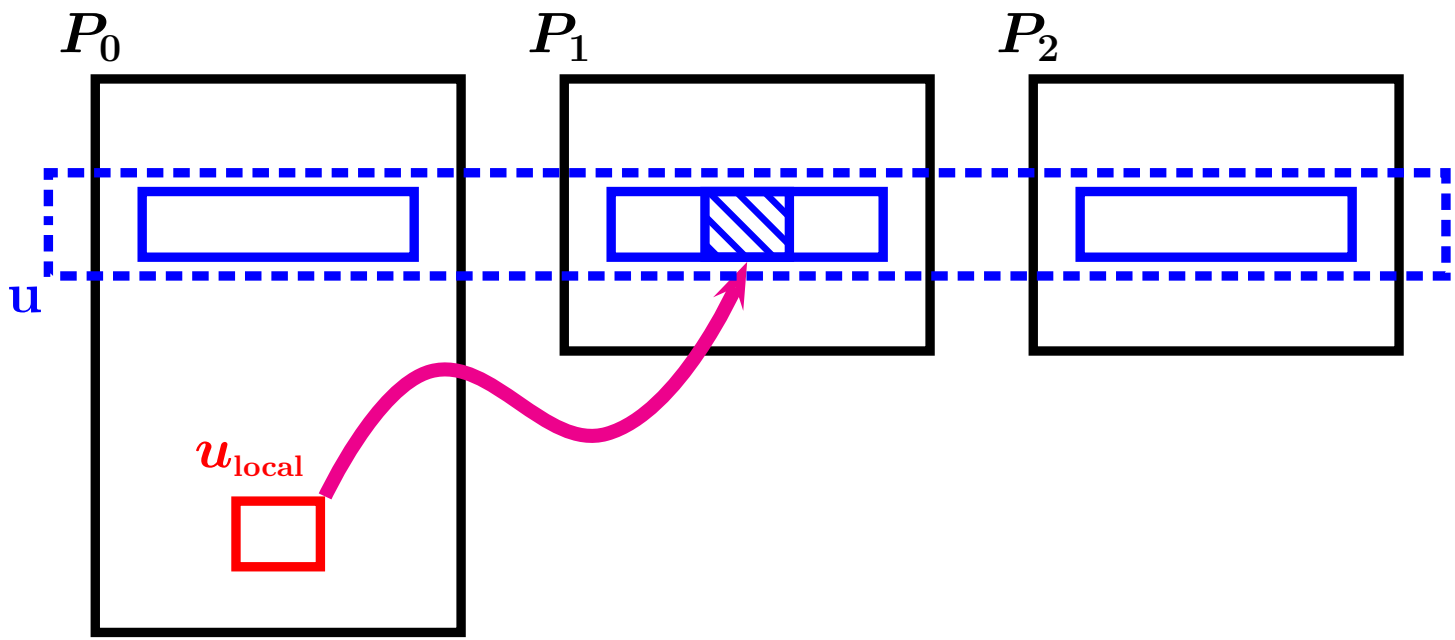


Simple example using OpenSHMEM

```
 1  double *u = NULL;
 2  int me;
 3
 4  shmem_init ();
 5  me = shmem_my_pe ();
 6  u = (double *) shmem_malloc(sizeof(double)*1024);
 7
 8  if (me == 0) {
 9    double u_local[10];
10    u_local[5] = 43.0;
11    shmem_double_put(&(u[11]), &(u_local[0]), 10, me+1);
12   }
13  shmem_barrier(0,1,1, pSync);
14
15  shmem_free (u);
16  shmem_finalize();
```

(click here for a working example)

Put operation in the OpenSHMEM example



shmem_double_put

- u is called a symmetric global variable (1024 doubles on each process memory space).

Outline

Extensions to existing languages

Several PGAS programming environments add new keywords/pragmas to an existing language (C, fortran, etc.).

- UPC (Unified Parallel C), a superset of C
  `https://upc-lang.org`
- CAF (Co-Array Fortran), integrated part of recent fortran norms (2008+)
  `ftp://ftp.nag.co.uk/sc22wg5/N2001-N2050/N2007.pdf`
- XcalableMP, set of pragma's added to C/C++/fortran
  `http://www.xcalablemp.org/`

These tools can usually be mixed with MPI and/or OpenMP programming.

UPC

UPC (`http://upc.gwu.edu`) is a superset of the C language.
It's one of the first languages that uses 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).

UPC implementations

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

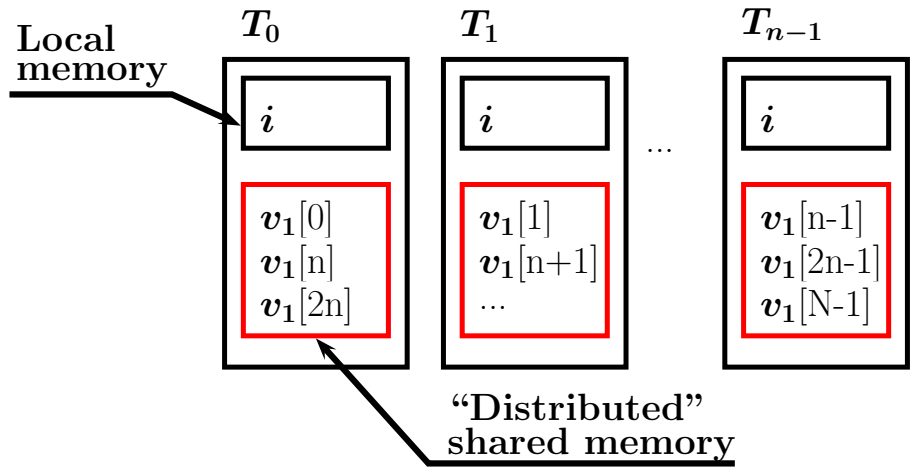
- Berkeley UPC (v 2.22.0, october 2015),
  `http://upc.lbl.gov`
- GCC/UPC (v 5.2.1.0, august 2015),
  `http://www.gccupc.org`

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

UPC Example (1)

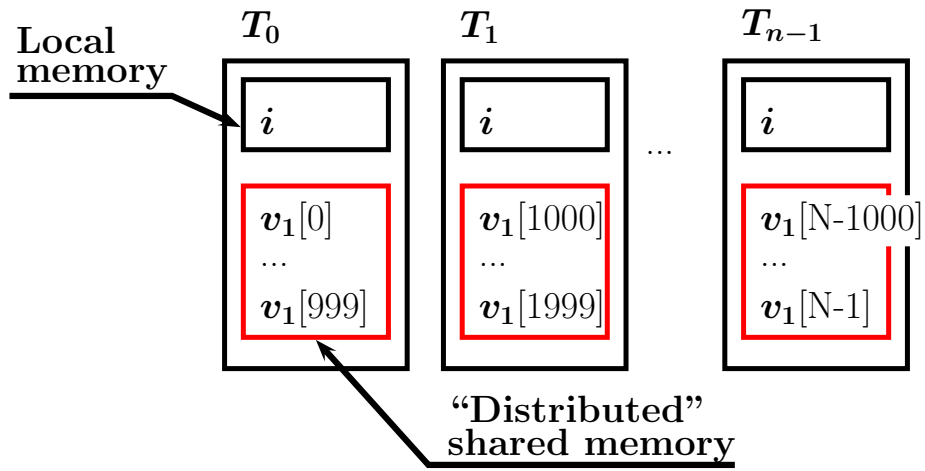
A (static) distributed data structure can be defined by:

```
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 }
```

(click here for a working example)

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
(3^{rd} 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):

1,37 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.

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.

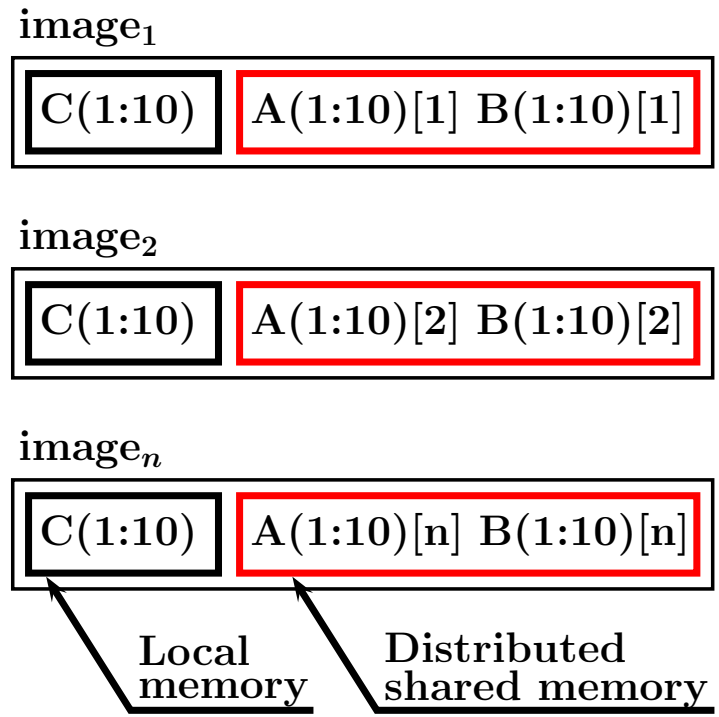
Co-Array Fortran implementations

There is now a reasonable choice of Co-Array Fortran implementations (even if co-arrays are integrated in the fortran norm from some time, real implementations were not available until recently).

- Intel Fortran in Intel Parallel Studio 2015 (Cluster edition for Linux),
- Cray Fortran in Cray Compiling Environment 8.4, sept 2015
- GCC/GFortran 5.2.0, july 2015 (distributed coarray provided by a separate library from OpenCoarrays `http://opencoarrays.org`),
- OpenUH 3.1 ()`http://web.cs.uh.edu/~openuh`), nov. 2015

Example in Co-Array Fortran

```
program write_test
integer,allocatable :: A(:)[*], B(:)[*]
integer i,rank
integer C(10)

allocate(A(10)[@team_world])
allocate(B(10)[@team_world])

rank = team_rank(team_world)
do i=1,10
    B(i)=i*rank
enddo
call barrier(team_world)

if (rank > 1) then
  A(:)= B(:)[rank-1]+C(:)
endif
end
```



XcalableMP

XcalableMP (`http://www.xcalablemp.org`) was designed and developped at 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).

An XcalableMP program can be compiled by the Omni compiler (`http://omni-compiler.org`). The most recent version is 0.9.1, and was released in april 2015.

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

XcalableMP

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

```
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];
    }
}
```

Outline

Specific languages

Lastly, some PGAS programming environments define a new language.
They provide an integrated way to program using the hybrid model without calling explicitely MPI or OpenMP (specifically, they use the APGAS programming model).

- X10

  `http://x10-lang.org`

- Chapel

  `http://chapel.cray.com`

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

Intermediate source code generation in C (Chapel), C++ (X10), or java (X10).

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.

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

- creation of threads in the same or 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

Chapel: 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;
...
```

Chapel: Task Parallelism, global data access

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();
}
```

X10 language

X10 (`http://x10.codehaus.org`) is a language defined and developped at IBM Research.

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)

- 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:

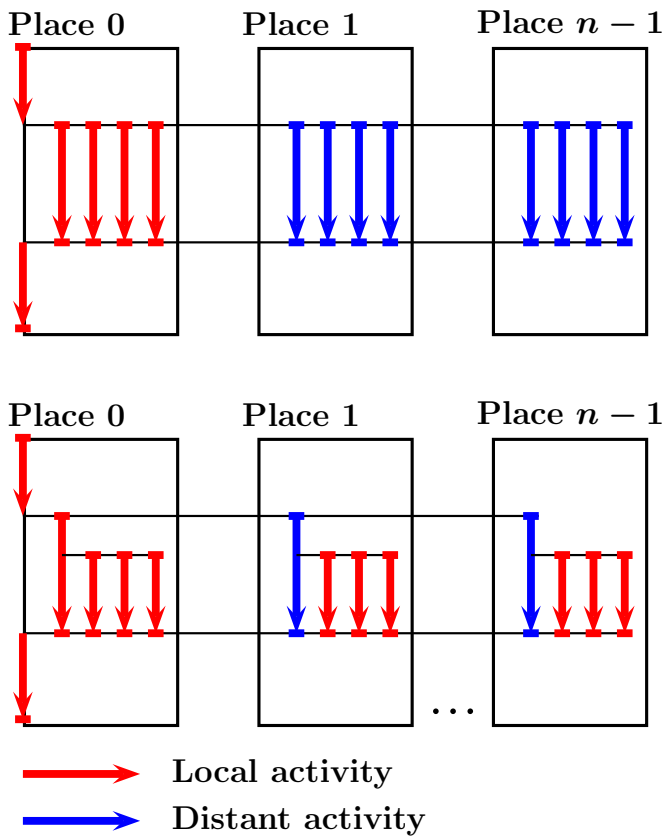
```
at (A.dist(2,2))
    A(2,2) = (at(A.dist(3,0)) A(3,0)) + 4.5;

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 starts in place 0.
Then this activity can start other activities at the same or other places. These activities can themselves launch local or remote activities.

```
finish
for (q in A)
 async at (q) {
  S(A(q));
}
```

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



APGAS languages
some guidelines for performance

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

- try to launch as many local threads as possible (vs. distant ones)
- try to put as many barriers between colocalized threads as possible (vs barriers between threads on different contexts).
- X10 and Chapel use light threads but their creation take some time, so put enough work into each thread
- if you know that a region is cartesian, specify it explicitely (the compilers cannot always detect it)
- ...