St. Petersburg State University of Aerospace Instrumentation

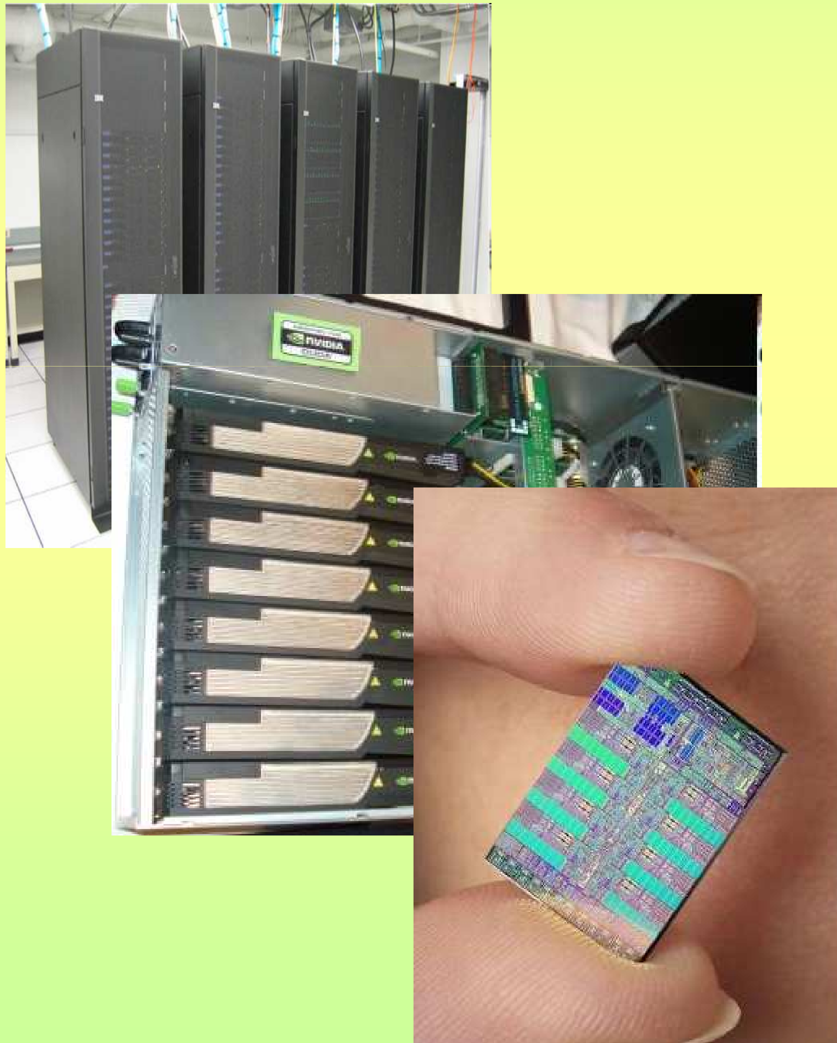
Institute of High-Performance Computer and Network Technologies

Data allocation for parallel processing in distributed computing systems

Alexey Syschikov
Researcher
Parallel programming Lab
Bolshaya Morskaya, No 67
190 000 St. Petersburg, Russia
E-mail: alexey.syschikov@guap.ru

Denis Rutkov
Post-graduate student
Parallel programming Lab
Bolshaya Morskaya, No 67
190 000 St. Petersburg, Russia
E-mail: dendron2000@mail.ru

Introduction



- Many computations have a potential for parallelization
- Vector computational model is not sufficient to cover all algorithms
- Methods of data allocation to local memory of processor elements is a very important question in distributed computing system

Concept of data allocation

Data allocation

Type of allocation

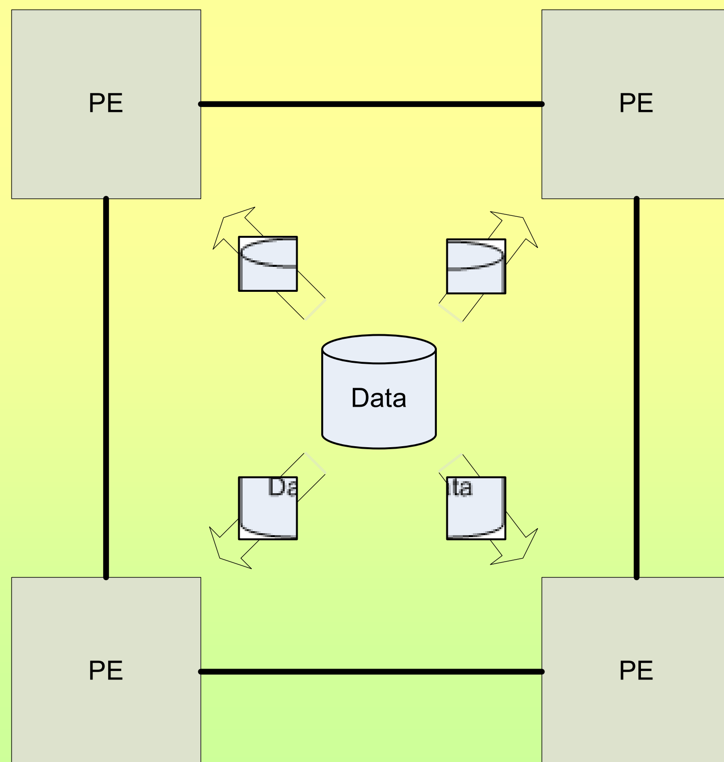
How data are allocated on PE's of a distributed system.

Specification method

What language instruments are presented for programmer to define and control data allocation

Data distribution

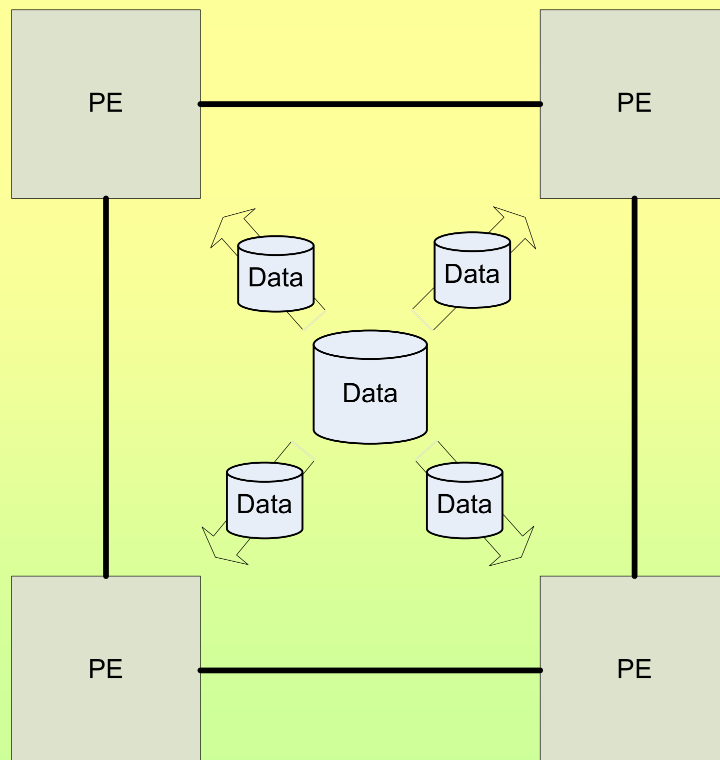
PEs receive just a part of data that is required, remaining data should be requested from other PEs



- **Pros:**
  - Parallel computations with complex data dependences
  - Process data without increasing of data in system
- **Cons:**
  - Reduced computation effectiveness due to communications
  - Necessity for synchronization of inter-processor communications
  - Data communications in program code

Data localization

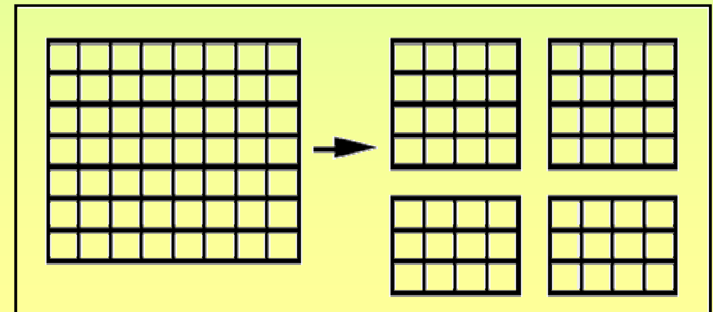
PEs receive sufficient data for computation, no requests to other PEs can and should be made



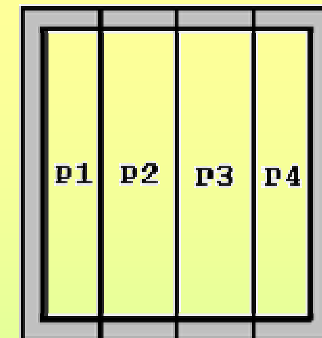
- **Pros:**
  - PEs are completely independent
  - No communications during the computation process
- **Cons:**
  - Increased data amount in the system
  - Requirements for centralization and preparing of data for localization
  - Impossible to parallelize computations that have non-localizable data

Allocation specification methods

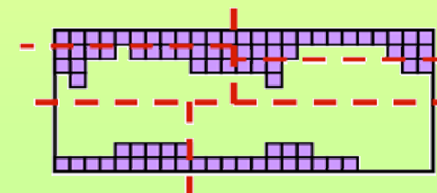
- **Automatic allocation**
  - Data will be allocated without the involvement of programmer
  - Data allocation task must be fully solved by a compiler



- **Predefined allocation**
  - Programmer has a parameterized templates to allocate data to virtual processors
  - Mapping of virtual processors to physical processors is performed by a compiler or in runtime



- **Custom allocations**
  - Custom allocations are completely specified by the programmer



Reviewed languages and system

- **Message Passing Interface (MPI) / MPI Forum**
  - MPI is a communication routines between library (C/C++/Fortran). Currently MPI is a de-facto standard for message passing.
- **High Performance Fortran (HPF) / HPF Forum**
  - HPF is a set of extensions to Fortran 90 that provide access to high-performance features with maintaining portability across platforms.
- **Co-array Fortran (Fortran 2008) / Minnesota&RICE universities**
  - Co-Array Fortran is a set of extensions to Fortran 95 for SPMD parallel processing on any kind of parallel architecture.
- **Distributed Virtual Machine (DVM) / Keldysh Institute, RAS**
  - DVM allows to develop parallel programs in C-DVM and Fortran-DVM languages for different architectures computers and computer networks.
- **ZPL / University of Washington**
  - ZPL is an array programming language without explicit parallel instructions and automatic data distribution.
- **Chapel / CRAY**
  - Chapel is a new imperative block-structured high-performance programming language.

Data allocation types

- **Message Passing Interface (MPI)**
  - MPI 1.x standard defines **data localization**
  - MPI 2.x standard adds **data distribution** by direct read/write
- **High Performance Fortran (HPF)**
  - HPF uses **data localization**. It doesn't allow defining explicit inter-processor communications or access data located on other PE
- **Co-array Fortran (Fortran 2008)**
  - Co-array Fortran uses **data distribution**. In fact, Co-array Fortran is a high-level superstructure over traditional communication routines (MPI).
- **Distributed Virtual Machine (DVM)**
  - DVM uses both types: **data localization** (directive DISTRIBUTE), **data distribution** (directives SHADOW, REMOTE, REDUCTION).
- **ZPL**
  - ZPL execution model uses **data distribution**.
- **Chapel**
  - Chapel uses **data distribution**.

Allocation specification methods

- **Message Passing Interface (MPI)**
  - MPI functions organize **predefined allocations** with blocks: Functions SEND, SCATTER, GATHER etc.
- **High Performance Fortran (HPF)**
  - HPF 1.x uses **predefined allocations** of equally-sized blocks (BLOCK and CYCLIC directives).
- **Co-array Fortran (Fortran 2008)**
  - Co-array Fortran uses the **predefined allocation** with fixed-size blocks.
- **Distributed Virtual Machine (DVM)**
  - DVM uses **predefined allocations** with blocks (BLOCK/GEN_BLOCK) or over alignment (ALIGN WITH)
- **ZPL**
  - ZPL execution model supports **automatic allocation** with blocks only
- **Chapel**
  - Chapel is oriented towards **custom allocations**. However, it will also support a set of **predefined distributions** (Block, Cyclic, Cut…)

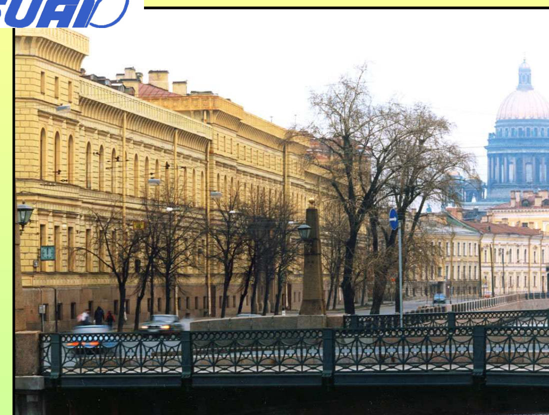
Summary of features

Language or system	Allocation types		Allocation specification		
	Locali-zation	Distri-bution	Auto-matic	Pre-defined	Cus-tom
MPI	+	+ (2.x)	–	+	~
HPF	+	–	~	+	–
Co-array Fortran	–	+	–	+	–
DVM	+	+	–	+	–
ZPL	–	+	+	–	–
Chapel	–	+	–	+	+

Summary

- Old problem of compromise between programmability and functionality also appeared in data allocation
- Most approaches use the predefined allocations with just several predefined allocations: about 5 templates.
- Limited amount of allocation templates significantly restrict abilities of a programmer to implement his task conveniently.
- Selection between data localization and data distribution is always a compromise.
- **The language should contain small set of predefined allocations and ability to construct custom allocations.**
- **Clear choice must be done between data localization and data allocation according to tasks specifics.**

The end



Backup slides

MPI functions organize **predefined allocations.**

1. With blocks. Functions SEND, SCATTER, GATHER etc.

   a. Equal blocks

```
MPI_Scatter( sendbuf, 15, MPI_INT, rbuf, 15, MPI_INT, root, comm);
// Where 15 - a block size for every destination process
```

/* Process root split source data (stored at address sendbuf) between all processes of communicator comm. with equal blocks of 15 integers for every process. */

   a. Unequal blocks

```
MPI_Comm comm;
int gsize,sbuf[1000];
int displ, rbuf[1000], i, disp[1000], cnt[1000];
...
MPI_Comm_size( comm, &gsize);
displ = 0;
for (i=0; i<gsize; ++i) {
   disp[i] = displ;
   cnt[i] = i*2;
   displ += cnt[i];
}
MPI_Scatterv( sbuf, cnt, disp, MPI_INT, rbuf, cnt[nproc], MPI_INT, 0, comm);
```

/* Where disp – array with displacements in the source array, cnt – array with block sizes
Process *root* split source data (stored at address *sbuf*) between all processes of communicator *comm.* with different blocks of 0, 2, 4, etc. integers for processes 0, 1, 2, etc. respectively. */

Allows to make a data duplication thus, in a common, this data allocation may be considered as the **custom allocation.**

In the Co-array Fortran language it is used the **predefined allocation** with fixed-size blocks. An array is extended with external dimensions which are allocated to virtual processors. Exact execution stream should obviously access either to its local copy of an array or to a remote one.

```
X     = Y[PE]     ! get from Y[PE]
Y[PE] = X         ! put into Y[PE]
Y[:]  = X         ! broadcast X
Y[L]  = X         ! broadcast X
over subset of PE’s in array L
Z(:)  = Y[:]      ! collect all Y
```

DVM-system uses **predefined allocations:**

1. With blocks

   a. Equal blocks (localization / distribution). Parameter – amount of data for block or for equal blocks between all processors.

```
CDVM$  PROCESSORS R( 4 )
REAL A(12)
CDVM$  DISTRIBUTE A (BLOCK) ONTO R
```

!Split the array A between amount of processors specified in R with equal blocks of |A|/|R| elements. For some values of |A| and |R| last processors in R may receive less array elements or even receive nothing.

   a. Unequal blocks (localization). Parameter – array with data amount for every processor or array with weights of source data elements.

```
CDVM$  PROCESSORS R( 4 )
INTEGER BS(4)
REAL A(12)
CDVM$   DISTRIBUTE A ( GEN_BLOCK( BS ) ) ONTO R
```

!Split the array A on |BS| blocks, where block *i* has size BS(*i*) and is placed on processor R(*i*),

1. Allocation over alignment.

```
REAL A(10), B(10,10), C(10)
CDVM$  DISTRIBUTE B ( BLOCK , BLOCK )
CDVM$  ALIGN A( I )      WITH B( 1, I )
```

!Alignment on array section (vector alignment over the first row of matrix A)

```
CDVM$  ALIGN C( I )      WITH B( *, I )
```

!Vector multiplication (alignment of vector over every rows of matrix B)

ZPL

```
---------- Declarations ----------
region
  R    = [1..n,   1..n  ]; -- problem region
  BigR = [0..n+1, 0..n+1]; -- with borders
direction
  north = [-1,  0]; -- cardinal directions
  east  = [ 0,  1];
  south = [ 1,  0];
  west  = [ 0, -1];
---------- Entry Procedure ----------
procedure jacobi();
var
  A, Temp : [BigR] float;
  delta   :        float;
[R] begin
  repeat
    Temp := (A@north + A@east + A@south + A@west) / 4.0;
    delta := max<< abs(A-Temp);
    A := Temp;
  until delta < epsilon;
end;
```

--Jacobi iteration. Given an array A, iteratively replace its elements with the average of their four nearest neighbours, until the largest change between two consecutive iterations is less than epsilon.

Chapel

```
const n1 = 1000000;
class MyC: Distribution {
  const z: integer; /* block size */
  const ntl: integer =... /* number of target locales */
function map(i:index(source)): locale { return
Locales(mod(ceil(i/z-1)+1,ntl));}
}
```

/*Global map for a simplified block-cyclic distribution with block size z≥1; the type of argument i is the type of the indices in the source domain: */

```
class MyB: Distribution {
const bl: integer =. . .; /* block length */
function map(i: index(source)): locale { return
Locales(ceil(i/bl));}
}
```

/*Global map for a simplified regular block distribution with block length bl: */

```
const D1C: domain (1) distributed (MyC(z=100))=[1..n1];
const D1B: domain (1) distributed (MyB)
   on Locales(1..num locales/10)=[1..n1];
var A1: [D1C] float;
var A2: [D1B] float;
```