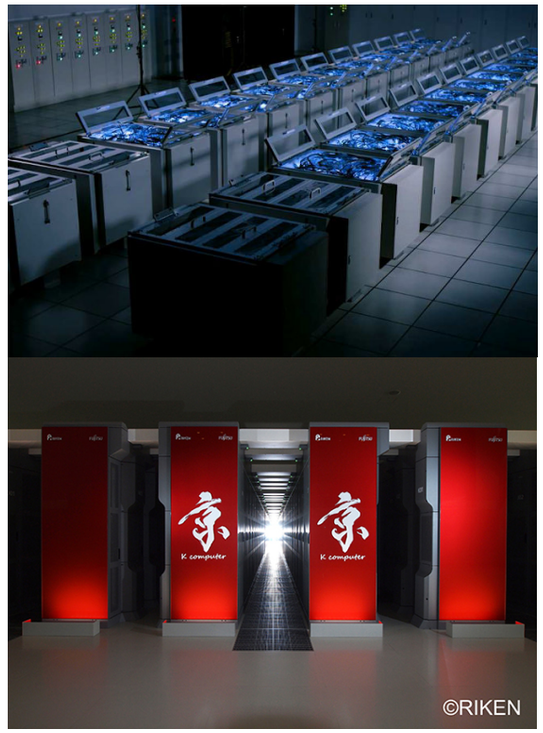
Linkage of XcalableMP and Python languages for high productivity on HPC cluster system

Masahiro Nakao (RIKEN Center for Computational Science)

Background

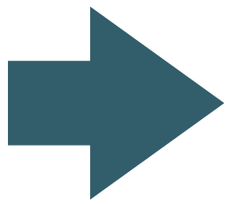
- **XcalableMP (XMP)** is a directive-based language extension for HPC cluster systems
  - Provide directives for PGAS programming
  - Based on C and Fortran (C++ on the table)
  - Designed by PC Cluster consortium
  - http://xcalablemp.org



- **Omni compiler**
  - Reference implementation for XMP
  - Developed by R-CCS and University of Tsukuba
  - Source-to-Source compiler
    - Support : The K computer, Intel Xeon Phi Cluster, Cray machines, ...
  - Open source software
  - http://omni-compiler.org

Objective

- The purpose of a PGAS language is to develop parallel applications with both **high productivity and performance**
- To do this, we think that a linking of a PGAS language and other languages is very important
  - **Different programming languages are good at different things**



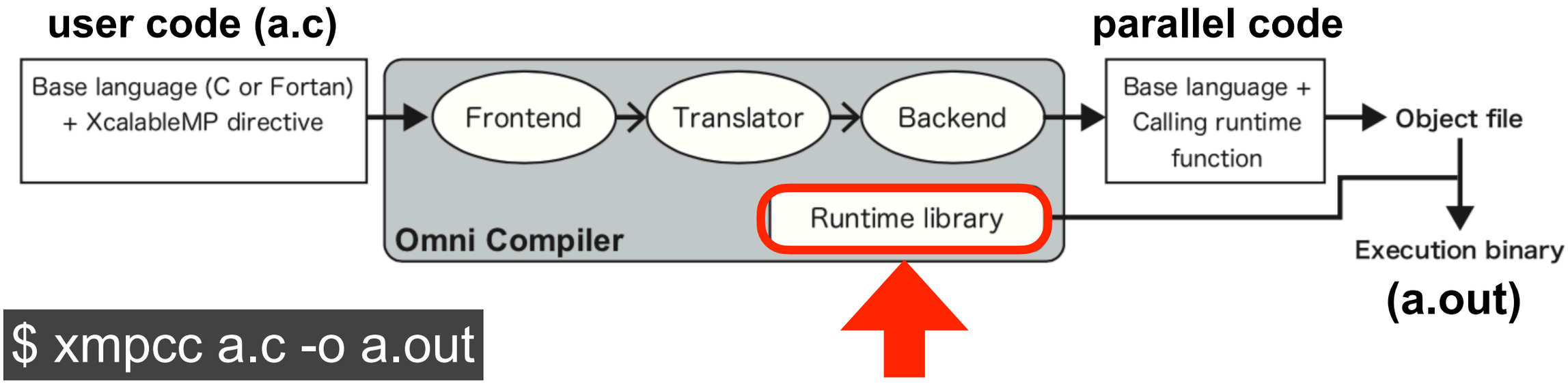
Development function for a linkage of XMP and Python

- Why do we choice Python ?
  - Python has a lot of packages for fields of science (e.g. SciPy and NumPy)
  - There are many users of python

Agenda from this slide

- Linkage of XMP and an MPI library
- Linkage of XMP and Python
  - Application to Order/degree Problem in Graph Theory
- Summary

Omni compiler



$ xmpcc a.c -o a.out

Add linkage functions

- A user code with XMP directives is translated to a parallel code with runtime calls of Omni compiler's runtime library
- The runtime library is implemented in C and MPI
- The parallel code is compiled by a native compiler (e.g. GNU, Intel, ...) to create an execution binary

Call an XMP program from an MPI program

- Implement following functions (which are defined in xmp.h)

Language	Return Value Type	Function	Description
XMP/C XMP/F	void (None)	xmp_init(MPI_Comm) xmp_init(Integer)	Initialize XMP environment
XMP/C XMP/F	void (None)	xmp_finalize(void) xmp_finalize()	Finalize XMP environment

MPI program (mpi.c)

```
#include <xmp.h>
#include <mpi.h>

int main(int argc, char **argv){
  MPI_Init(&argc, &argv);

  xmp_init(MPI_COMM_WORLD);
  call_xmp();
  xmp_finalize();
```

XMP program (xmp.c)

```
void call_xmp(){
#pragma xmp nodes p[3]
      :
```

```
$ xmpcc xmp.c -c
$ mpicc mpi.c -c
$ xmpcc xmp.o mpi.o -o a.out
$ mpirun -np 3 a.out
```

Call an MPI program from an XMP program

- Implement following functions (which are defined in xmp.h)

Language	Return Value Type	Function	Description
XMP/C	void	xmp_init_mpi(int*, char***)	Initialize MPI environment
XMP/F	(None)	xmp_init_mpi()	
XMP/C	MPI_Comm	xmp_get_mpi_comm(void)	Create MPI communicator from XMP node set
XMP/F	Integer	xmp_get_mpi_comm()	
XMP/C	void	xmp_finalize_mpi(void)	Finalize MPI environment
XMP/F	(None)	xmp_finalize_mpi()	

XMP program (xmp.c)

```
#include <xmp.h>
#include <mpi.h>
#pragma xmp nodes p[3]

int main(int argc, char **argv){
  xmp_init_mpi(&argc, &argv);
  MPI_Comm comm = xmp_get_mpi_comm();
  call_mpi(comm);
  xmp_finalize_mpi();
```

MPI program (mpi.c)

```
#include <mpi.h>

void call_mpi(MPI_Comm comm){
  int rank, size;
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);
```

```
$ xmpcc xmp.c -c
$ mpicc mpi.c -c
$ xmpcc xmp.o mpi.o -o a.out
$ mpirun -np 3 a.out
```

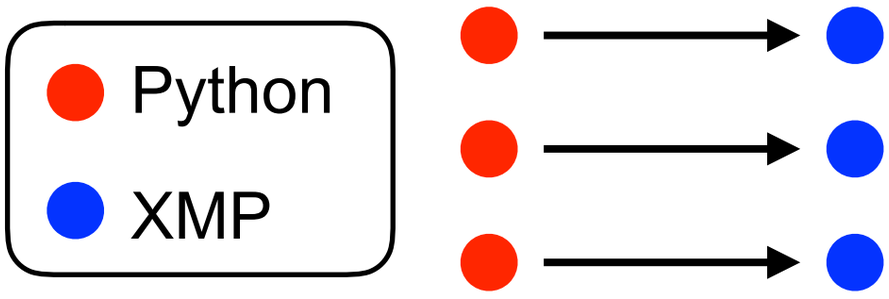
Agenda from this slide

- Linkage of XMP and an MPI library
- Linkage of XMP and Python
  - Application to Order/degree Problem in Graph Theory
- Summary

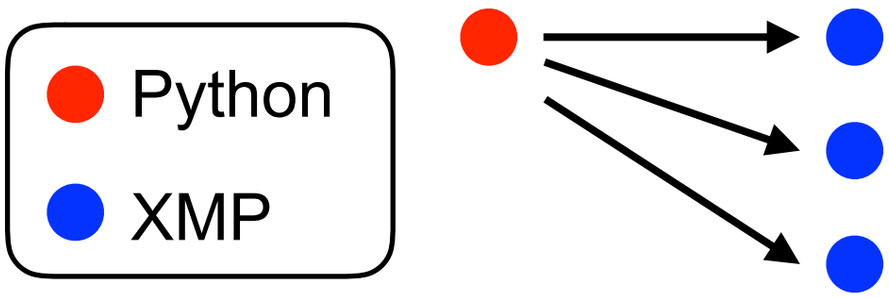
Linkage of XMP and Python

1 : Parallel Python program calls a parallel XMP program

- This concept is the same as that of XMP and MPI



2 : Serial Python program calls a parallel XMP program



Parallel Python program calls a parallel XMP program

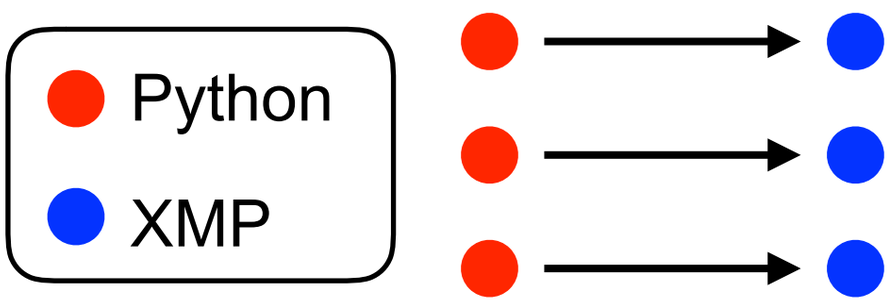
Python program (a.py)

```
import xmp
from mpi4py import MPI

lib = xmp.Lib("xmp.so")
args = ([1,2,3], [4,5,6])
job  = lib.call(MPI.COMM_WORLD, "call_xmp", args)
```

XMP program (xmp.c)

```
void call_xmp(long a1[3],
              long a2[3]){
#pragma xmp nodes p[3]
    :
```

- xmp.Lib() sets "shared library" which is developed in XMP program
- xmp.Lib.call() executes an XMP program
  - xmp.Lib.call() calls "xmp_init()" and "xmp_finalize()" internally




```
$ xmpcc -fPIC -shared xmp.c -o xmp.so
$ mpirun -np 3 python a.py
```

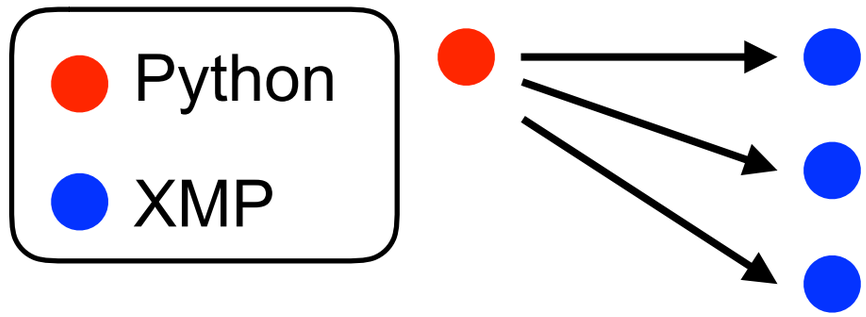
Serial Python program calls a parallel XMP program

Python program (a.py)

```
import xmp

lib = xmp.Lib("xmp.so")
args = ([1,2,3], [3,4,5])
job = lib.spawn(3, "call_xmp", args)
```

XMP program (xmp.c)

```
void call_xmp(long a1[3],
              long a2[3]){
#pragma xmp nodes p[3]
    :
```



- xmp.Lib.spawn() method creates new processes and they work as an XMP program in parallel

```
$ xmpcc -fPIC -shared xmp.c -o xmp.so
$ mpirun -np 1 python a.py
```

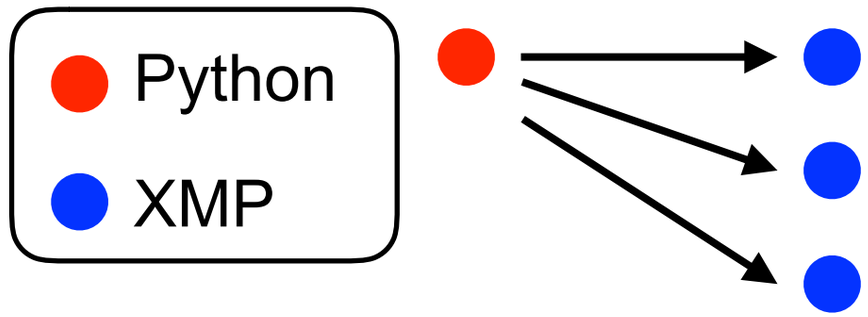
Serial Python program calls a parallel XMP program

Python program (a.py)

```
import xmp

lib = xmp.Lib("xmp.so")
args = ([1,2,3], [3,4,5])
job = lib.spawn(3, "call_xmp", args, async=True)
// other work
job.wait()
```

XMP program (xmp.c)

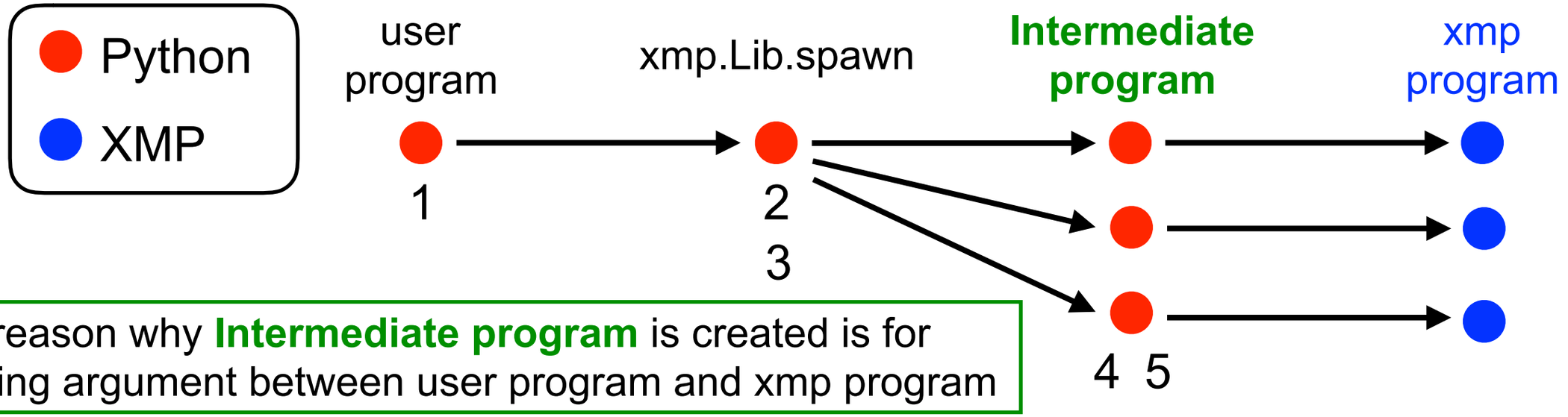
```
void call_xmp(long a1[3],
              long a2[3]){
#pragma xmp nodes p[3]
    :
```



- xmp.Lib.spawn() method creates new processes and they work as an XMP program in parallel

```
$ xmpcc -fPIC -shared xmp.c -o xmp.so
$ mpirun -np 1 python a.py
```

Implementation of xmp.Lib.spawn()



The reason why **Intermediate program** is created is for passing argument between user program and xmp program

1. User program calls xmp.Lib.spawn()
2. xmp.Lib.spawn() creates **Intermediate program** automatically and executes **Intermediate program** using the spawn method of mpi4py
3. xmp.Lib.spawn() broadcasts the arguments to **Intermediate program** using mpi4py.Bcast()
4. **Intermediate program** receives the arguments
5. **Intermediate program** executes an XMP program (This procedure is the same as parallel run style)

Intermediate program

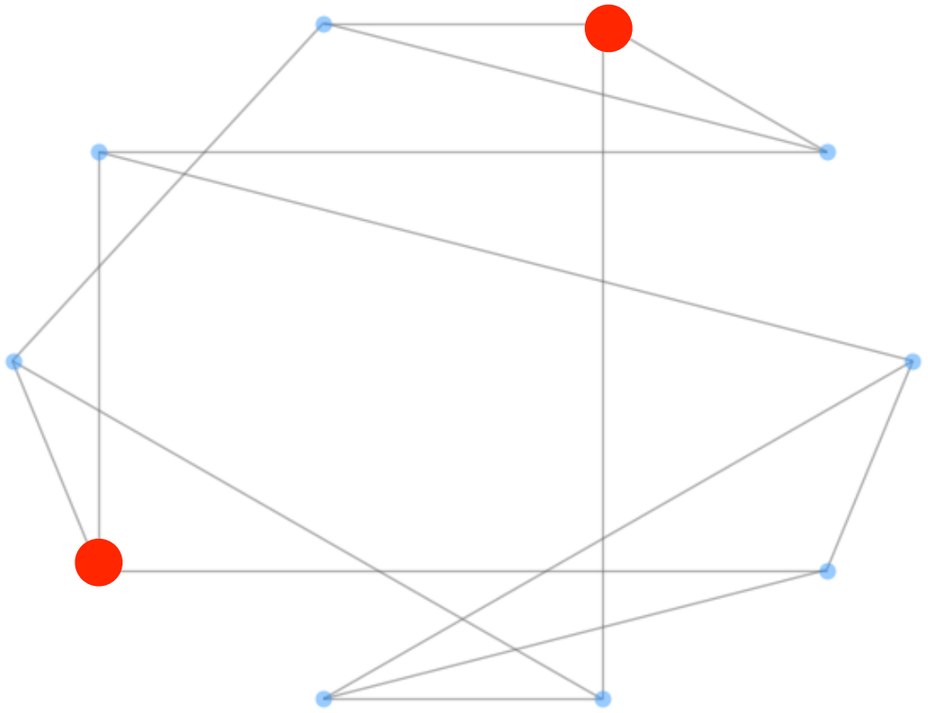
```
from mpi4py import MPI
import numpy
from ctypes import *

comm = MPI.Comm.Get_parent()
arg0 = numpy.zeros(2)
comm.Bcast(arg0, root=0)
lib = CDLL("xmp.so")
lib.xmp_init_py(comm.py2f())
lib.call_xmp(arg0.ctypes)
lib.xmp_finalize()
comm.Disconnect()
```

Agenda from this slide

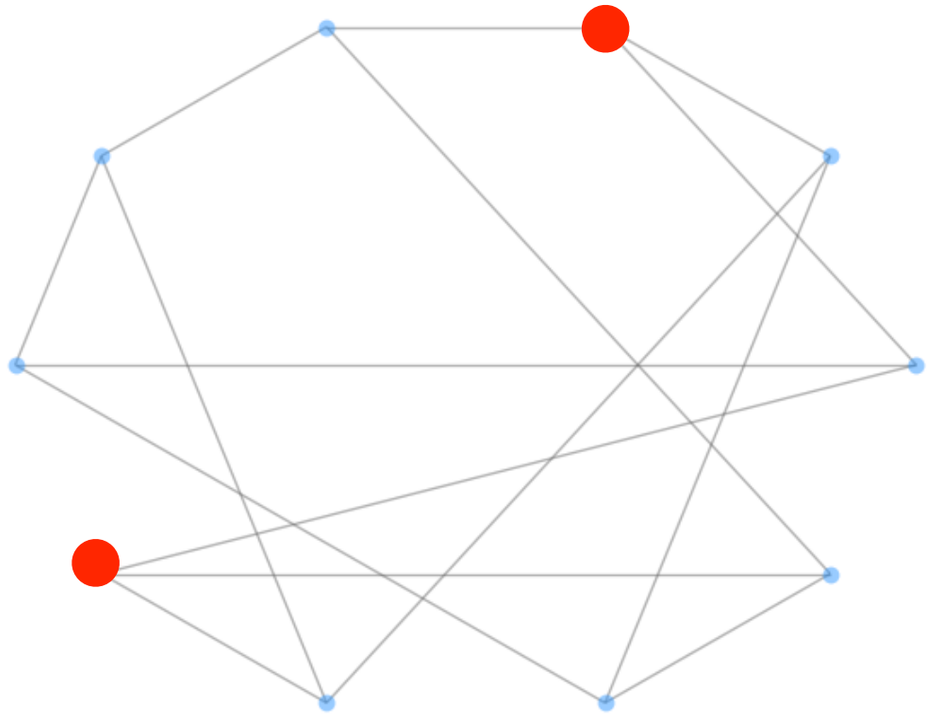
- Linkage of XMP and an MPI library
- Linkage of XMP and Python
  - Application to Order/degree Problem in Graph Theory
- Summary

What is Order/degree problem ?

- Minimizes the "diameter" and "average shortest path length (ASPL)" among vertices in an undirected graph with "a given number of vertices and degrees".
  - The problem is useful for designing low latency interconnection networks
  - Although the smallest diameter and ASPL can be calculated from the given vertices and degrees [V.G.Cerf 1974], we don't know how edges and vertices are connected
- Examples of the graph with vertices = 10 and degree = 3



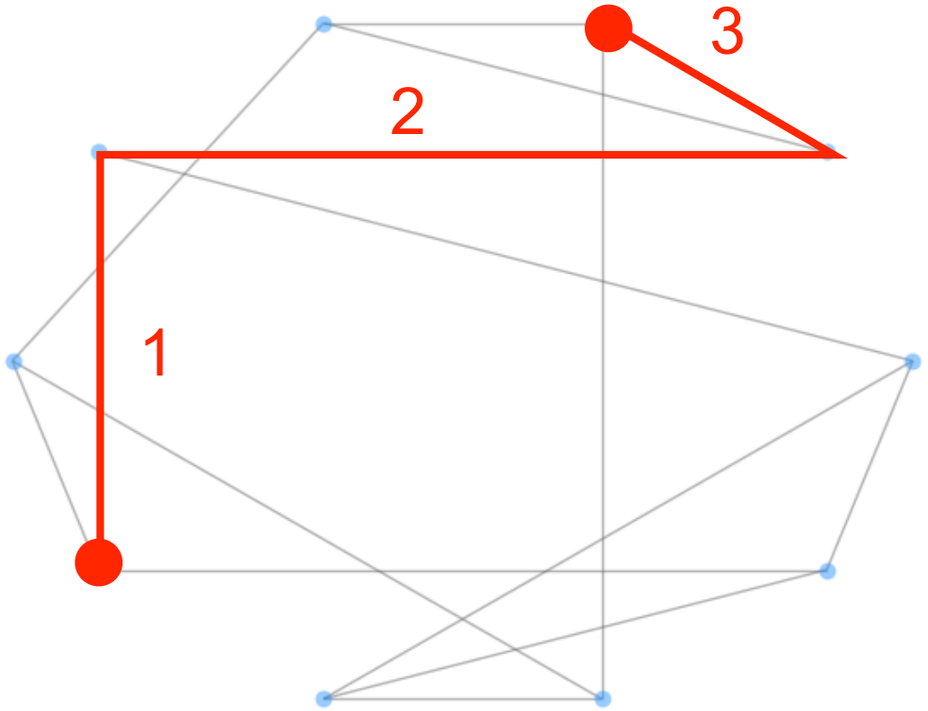
Diameter=3, ASPL=1.89 (Random)



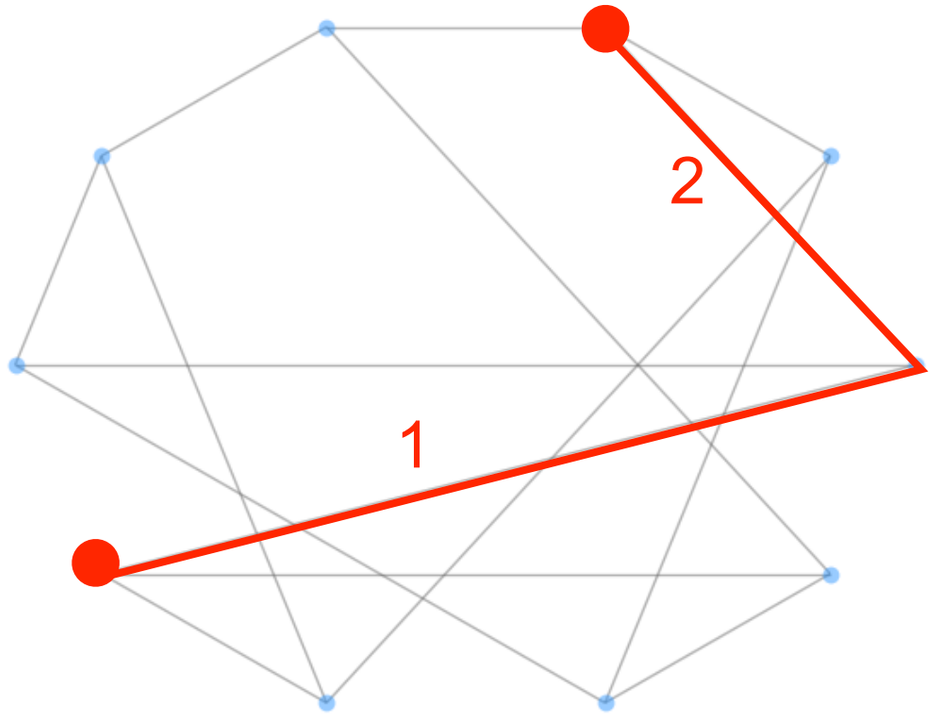
Diameter=2, ASPL=1.67 (Optimal)

What is Graph Order/degree problem ?

- Minimizes the "diameter" and "average shortest path length (ASPL)" among vertices in an undirected graph with "a given number of vertices and degrees".
  - The problem is useful for designing low latency interconnection networks
  - Although the smallest diameter and ASPL can be calculated from the given vertices and degrees [V.G.Cerf 1974], we don't know how edges and vertices are connected
- Examples of the graph with vertices = 10 and degree = 3



Diameter=3, ASPL=1.89 (Random)



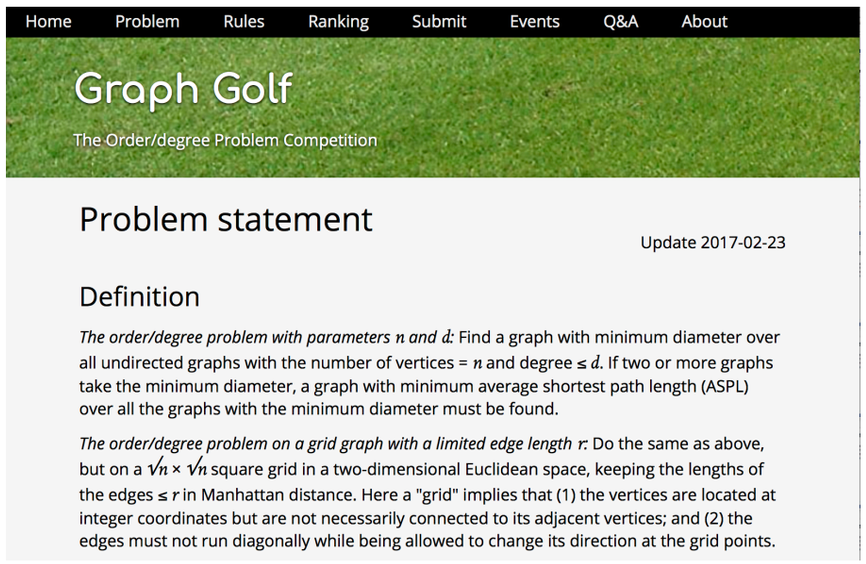
Diameter=2, ASPL=1.67 (Optimal)

Graph Golf@National Institute of Informatics

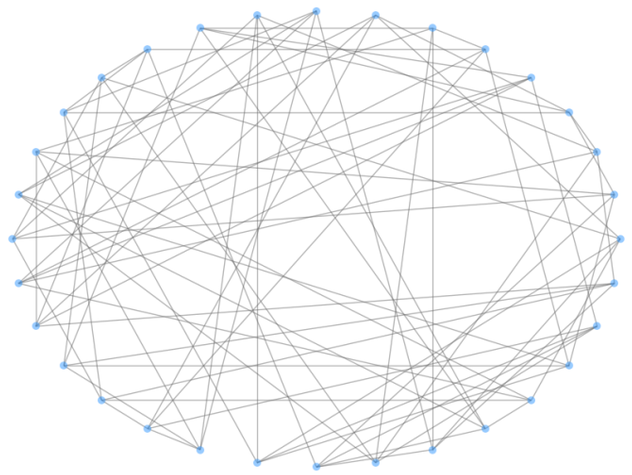
- National Institute of Informatics has held **the Graph Golf competition** since 2015 for Order/degree problem.
  - http://research.nii.ac.jp/graphgolf
  - Some combinations of vertices and degrees are provided

Combination of vertices and degrees of General Graph Category in 2017

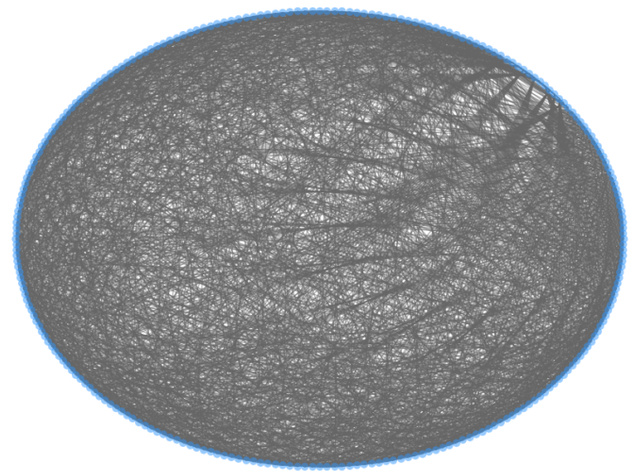
Number of vertices (n)	32	256	576	1344	4896	9344	88128	98304	100000	100000
Number of degrees (d)	5	18	30	30	24	10	12	10	32	64



Vertices=32, Degrees=5

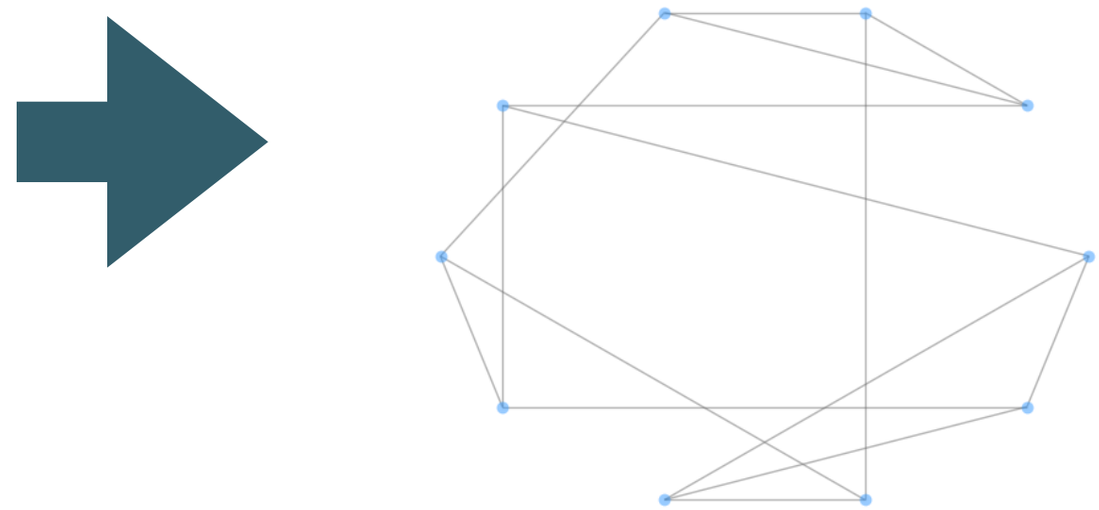


Vertices=256, Degrees=18



Usage of XMP

- A python program for the Graph Golf competition is available on the official website
  - http://research.nii.ac.jp/graphgolf/py/create-random.py (About 100 lines)
  - The python program outputs follow from the number of vertices and degrees.
    - Initial graph with random edges
    - Calculation of diameter and ASPL
    - The graph is saved in PNG format
  - Python "networkx" package is used
    - https://github.com/networkx/



Diameter=3, ASPL=1.89 (Random)

- Issues
  - The python program doesn't search the smallest diameter and ASPL
  - To calculate diameter and ASPL, it requires a significant amount of time

We create an XMP program to solve the issues.

Graph Order/degree solver in Python + XMP

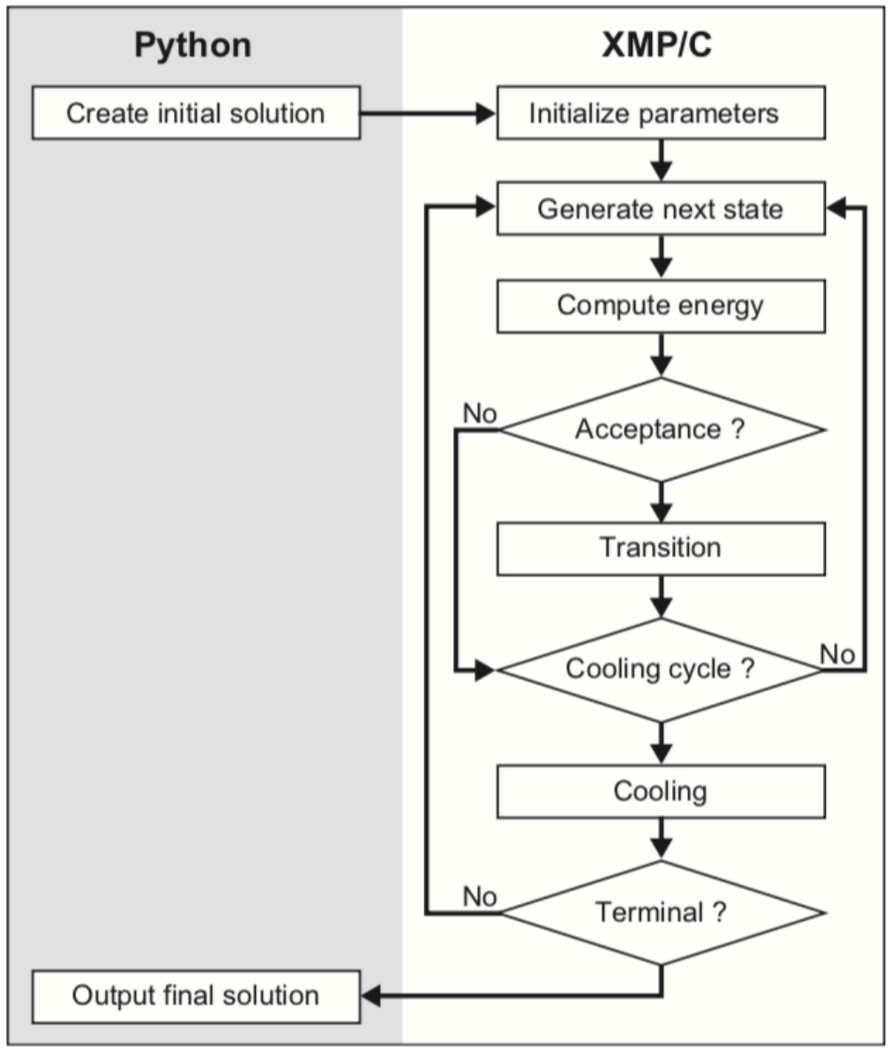
- **Implementation policy**
  - The existing python program generates an initial solution and saves a final solution
  - XMP program searches an optimal solution (includes calculates diameter and ASPL)
    - Optimization algorithm in XMP uses Simulated Annealing

Python

```
import xmp
  :
lib = xmp.Lib("xmp.so")
args = (vertices, degrees, edge)
lib.call(MPI.COMM_WORLD, "xmp_sa", args)
```

XMP/C

```
void xmp_sa(int vertices, int degrees,
                    int edge[vertices*degrees]){
   :
#pragma xmp loop on t[i]
  for(int i=0;i<vertices;i++){
      : // Calculate diameter and ASPL
  }
#pragma xmp reduction(max:diameter)
#pragma xmp reduction(+:ASPL)
```



Evaluation

- COMA cluster system at University of Tsukuba

CPU	Intel Xeon-E5 2670v2 2.8 GHz x 2 Sockets
Memory	DDR3 1866MHz 59.7GB/s 64GB
Network	InfiniBand FDR 7GB/s
Software	intel/16.0.2, intelmpi/5.1.1, Omni Compiler 1.2.1 Python 2.7.9, networkx 1.9

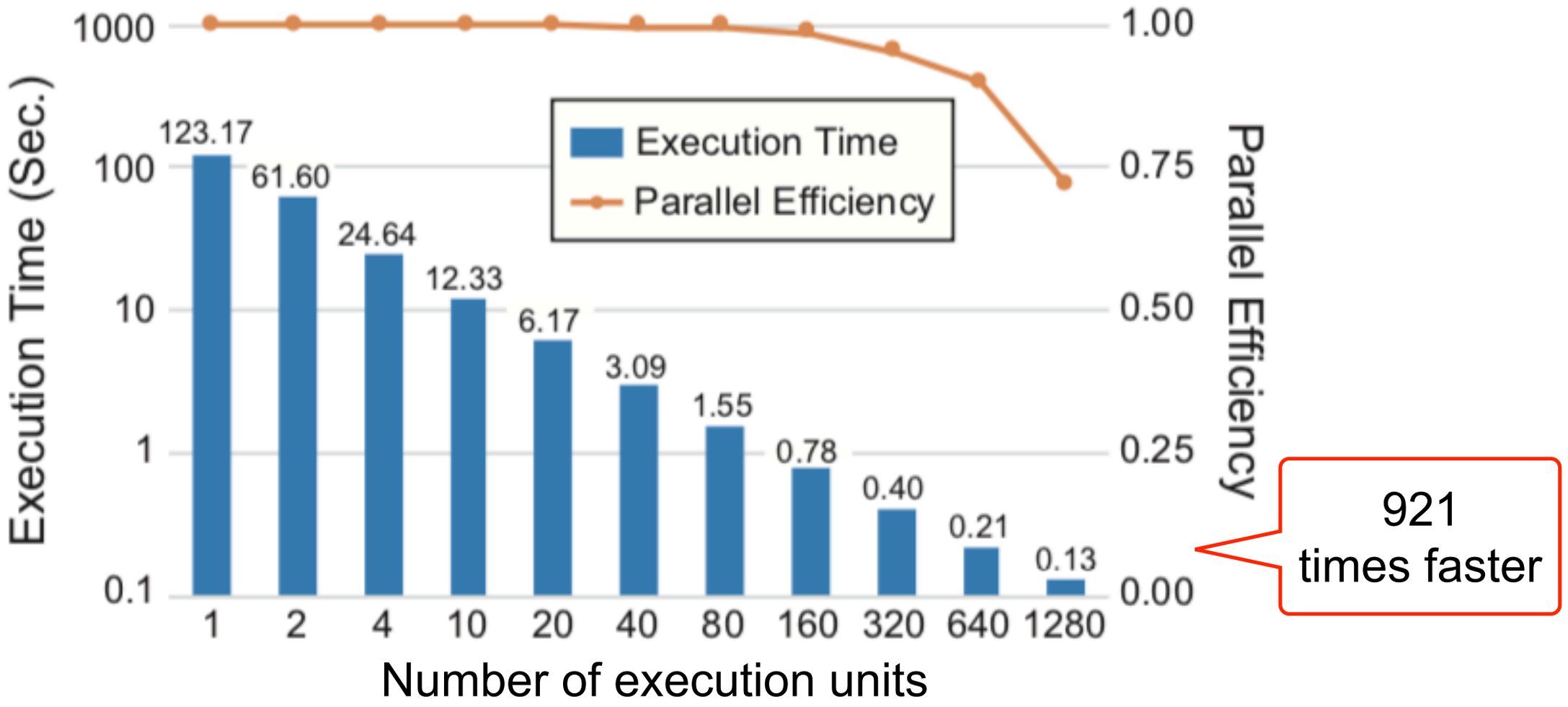
  - Elapse time to calculate diameter and ASPL
  - Problem size is vertices=9344, degrees=10

Combination of vertices and degrees of General Graph Category in 2017

Number of vertices (n)	32	256	576	1344	4896	9344	88128	98304	100000	100000
Number of degrees (d)	5	18	30	30	24	10	12	10	32	64

Performance results

- flat-MPI
- 20 processes in a single compute node
- The original python code using networkx package requires 148.83 sec.



Conclusion

- Since different programming languages are good at different things, we developed the linkage function for XMP and Python
  - Parallel python program calls a parallel XMP program
  - Serial python program spawns new processes which executes a parallel XMP program
- Development of an application of the Order/degree problem using the linkage function
  - As a result of using 1280 CPU cores, it achieved 921 times faster than using 1 CPU core.
  - Python networkx package is used to create an initial graph and save a final graph
  - By mixing Python and XMP, the parallel application is developed easily.