Mixed-Language Programming with XcalableMP

Masahiro Nakao

Abstract This chapter presents the mixed-language programming with XcalableMP and other programming languages. It is supported by the linkage functions between XcalableMP and MPI library. We also demonstrate how to call XcalableMP program from Python program (M. Nakao et al., Linkage of XcalableMP and Python languages for high productivity on HPC cluster system, Proceedings of Workshops of HPC Asia, No .9, pp.39–47, 2018).

1 Background

To develop applications on high-performance computing (HPC) cluster systems, Partitioned Global Address Space (PGAS) languages that can demonstrate high productivity are used [2–5]. Because the use of PGAS languages is familiar in one-sided communication, applications in PGAS languages can sometimes exhibit higher performance than those using MPI library by directly using a communication layer close to hardware [5, 6]. Examples of PGAS languages include XcalableMP (XMP) [5, 7, 8]; XcalableACC [9–11]; Coarray Fortran [12], PCJ [13], Unified Parallel C,¹ UPC++ [14], HabaneroUPC++ [15], X10 [16], Chapel [17], and DASH [18].

Although PGAS languages have many advantages, re-implementing an existing MPI application using a PGAS language is often not realistic for the following reasons: (1) since the number of lines of a real-world application code may reach several million, the programming cost for re-implementing is excessive, and (2) cases where productivity and performance have been improved by PGAS languages are limited. Moreover, since each programming language generally has its own

¹<https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.3.pdf>

M. Nakao (✉)
RIKEN Center for Computational Science, Kobe, Hyogo, Japan
e-mail: masahiro.nakao@riken.jp

strong and weak points, it is difficult to develop all parallel applications using just one programming language.

In order to exploit the advantages of a PGAS language, it is important to consider the linkage between the PGAS language and other languages. For example, modifying only a part where the performance or code outlook will be better using a PGAS language has the potential to partially alleviate the two problems listed in the previous paragraph because the programming cost of partial re-implementation is smaller than that of whole re-implementation.

We have designed an XMP language, and developed Omni compiler described in Chapter 2. This chapter describes the development of linkage functions between XMP and MPI library for Omni compiler. Moreover, it also describes how to call XMP program from Python program. Especially, since Python has numerous scientific computing libraries, we believe that linking Python and XMP will lead to a significant reduction in programming cost when developing HPC applications.

2 Translation by Omni Compiler

Figure 1 shows an example of a code translation for XMP in C language, where the code of the left figure is translated into that of the right figure.² Omni compiler inserts `xmp_init_all()` and `xmp_finalize_all()` automatically to perform the initialization and finalization processes, respectively. These functions are defined in Omni compiler runtime library. Since `xmp_init_all()` calls `MPI_Init()` and `MPI_Comm_dup()` internally to duplicate `MPI_COMM_WORLD`. Since the newly duplicated communicator is used to perform XMP communication, user MPI communication does not conflict with XMP communication. Additionally, `xmp_finalize_all()` calls `MPI_Finalize()` function internally.

In an XMP program, the **task** directive can divide a node set. The implementation of the **task** directive creates a new communicator based on the duplicated communicator by `MPI_Comm_split()`. If communication occurs within the range of the **task** directive, then the new communicator is used to perform the communication.

Omni compiler renames a user `main()` as `xmpc_main()` in order to place the `xmpc_main()` between `xmp_init_all()` and `xmp_finalize_all()`. New `main()` calls `xmpc_main()`. The above special translation is performed only for `main()`. For other functions (such as `foo()`), renaming is not performed.

²Since Omni compiler performs almost the same operation for an XMP in Fortran code, these examples are omitted in this chapter.

Fig. 1 Example of translation in Omni compiler [19]

3 Functions for Mixed-Language

For a mixed-language programming with XMP, Omni compiler has the following functions.

- Calling an MPI program from an XMP program
- Calling an XMP program from an MPI program
- Calling an XMP program from a Python program

3.1 Function to Call MPI Program from XMP Program

Table 1 shows the functions to call an MPI program from an XMP program. These functions are defined in Appendix A.1 of the XMP specification version 1.4.³ Figure 2 also shows an example of how to use these functions. In line 1 of the left figure, an XMP header file (xmp.h) is included to use the functions in Table 1. In line 2, an MPI header file (mpi.h) is included to obtain the information of the MPI communicator type that is used in line 7. In line 6, `xmp_init_mpi()` initializes an MPI environment. In line 8, `xmp_get_mpi_comm()` returns an MPI communicator from the information of the executing XMP node set. In line 9, `foo()` which is defined in right figure is called. In line 10, `xmp_finalize_mpi()` finalizes the MPI

³<https://xcalablemp.org/download/spec/xmp-spec-1.4.pdf>

Table 1 Functions to call MPI program from XMP program [19]

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/F	Integer	xmp_get_mpi_comm()	XMP node set
XMP/C	Void	xmp_finalize_mpi(void)	Finalize MPI environment
XMP/F	(None)	xmp_finalize_mpi()	

```

1 #include <xmp.h>
2 #include <mpi.h>
3 #pragma xmp nodes p[*]
4
5 int main(int argc, char **argv){
6   xmp_init_mpi(&argc, &argv);
7   MPI_Comm comm;
8   comm = xmp_get_mpi_comm();
9   foo(comm);
10  xmp_finalize_mpi();
11
12  return 0;
13 }

```

```

#include <mpi.h>

void foo(MPI_Comm comm){
  int rank, size;
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);
  :
}

```

Fig. 2 Example of calling MPI program (mpi.c) from XMP program (xmp.c) [19]

environment. Note that **xmp_get_mpi_comm()** and other MPI functions must be placed between **xmp_init_mpi()** and **xmp_finalize_mpi()**.

The implementations of **xmp_init_mpi()** and **xmp_finalize_mpi()** are empty because, as shown in Fig. 1, the **xmp_init_all()** and **xmp_finalize_all()** are always invoked at the beginning and the end of the program, and these functions initialize and finalize the MPI environment. Next, an implementation of **xmp_get_mpi_comm()** will be described. As shown in Sect. 2, the **task** directive creates a new MPI communicator. Thus, an MPI communicator is stored at a stack data architecture in Omni compiler runtime library. The **xmp_get_mpi_comm()** returns an MPI communicator at the top of the stack.

An example of compilation using Omni compiler is as follows:

```

$ mpicc mpi.c -c -o mpi.o
$ xmpcc xmp.c -c -o xmp.o
$ xmpcc mpi.o xmp.o -o a.out

```

Since Omni compiler uses an MPI compiler as a native compiler internally, MPI programs can also be compiled with the “xmpcc” command as follows:

```
$ xmpcc mpi.c -o mpi.o
```

Thus, it is also possible to execute all the compilation work using a single command as follows:

```
$ xmpcc mpi.c xmp.c -o a.out
```

The execution binary is executed using the execution command provided by user’s MPI environment as follows:

```
$ mpirun -np 4 a.out
```

3.2 *Function to Call XMP Program from MPI Program*

Table 2 shows the functions to call an XMP program from an MPI program. These functions are defined in Appendix A.2 of the XMP specification version 1.4. Figure 3 shows an example of how to use these functions. In line 1 of the left figure, an XMP header file is included to use functions in Table 2. In line 7, **xmp_init()** initializes an XMP environment, and creates the XMP node set based on the communicator specified in its argument. In line 8, **foo()** in the right figure is called. In line 9, **xmp_finalize()** finalizes the XMP environment. Note that the XMP functions must be placed between **xmp_init()** and **xmp_finalize()**.

Though the **xmp_init()** is implemented to call the **xmp_init_all()** function described in Sect. 2, it also performs the following ingenuities:

Table 2 Functions to call XMP program from MPI program [19]

Language	Return value type	Function	Description
XMP/C	Void	xmp_init(MPI_Comm)	Initialize XMP environment
XMP/F	(None)	xmp_init(Integer)	
XMP/C	Void	xmp_finalize(void)	Finalize XMP environment
XMP/F	(None)	xmp_finalize()

```

1 #include <xmp.h>
2 #include <mpi.h>
3
4 int main(int argc, char **argv){
5     MPI_Init(&argc, &argv);
6
7     xmp_init(MPI_COMM_WORLD);
8     foo();
9     xmp_finalize();
10
11 MPI_Finalize();
12 return 0;
13 }

```

```

void foo(){
    #pragma xmp nodes p[*]
    :
}

```

Fig. 3 Example of calling XMP program from MPI program [19]

- Before calling **xmp_init()**, **MPI_Init()** must be called in the MPI program. Therefore, we added a procedure that ensures **MPI_Init()** is not called in **xmp_init_all()**.
- As shown in Sect. 2, **xmp_init_all()** duplicates **MPI_COMM_WORLD** to perform XMP communication. We also added a procedure that ensures the communicator specified in **xmp_init()** is duplicated instead of **MPI_COMM_WORLD**.

Basically, **xmp_finalize()** is also implemented to call **xmp_finalize_all()** in Sect. 2. As with **xmp_init()**, after calling **xmp_finalize()**, **MPI_Finalize()** is called in an MPI program. Therefore, we added a procedure to ensure that **MPI_Finalize()** is not called in **xmp_finalize_all()**.

The method to compile and execute is the same as Sect. 3.1.

3.3 Function to Call XMP Program from Python Program

There are two types of calling an XMP program from a Python program in Omni compiler; the one is “calling from a parallel Python program,” the other is “calling from a sequential Python program.”

3.3.1 From Parallel Python Program

Figure 4 shows an example of calling an XMP program from a parallel Python program. We assume the use of “mpi4py” for a Python MPI environment. In line 2 of the left figure, an XMP package is imported which is in Omni compiler. In line 4, the shared library (**bar.so**) created from the right figure is read. In line 8,

```

1 from mpi4py import MPI
2 import xmp
3
4 lib = xmp.Lib("bar.so")
5 comm = MPI.COMM_WORLD
6 args = ([1,2,3], [4,5,6])
7
8 job = lib.call(comm, "foo", args)

```

```

void foo(long a[3], long b[3]){
    #pragma xmp nodes p[*]
    :
}

```

Fig. 4 Example of calling XMP program (bar.c) from parallel Python program (bar.py) [19]

xmp.Lib.call() calls a parallel XMP program. This function performs initialization and finalization for an XMP environment internally.

To use the features, the Omni compiler runtime library must be a shared library. Therefore, we develop a new compile process to create a shared library for Omni compiler. When adding an option “`--enable-shared`” to “`./configure`” as follows, shared libraries are created.

```
$ ./configure --enable-shared
```

An example of compilation and execution using Omni compiler is as follows. A shared library is created by “`xmpcc`” command from a user program. Compile options used to create the shared library depend on a native compiler (e.g., “`-fPIC -shared`” if the native compiler is `gcc`). The execution binary is executed via Python.

```
$ xmpcc -fPIC -shared bar.c -o bar.so
$ mpirun -np 4 python bar.py
```

3.3.2 From Sequential Python Program

Figure 5 shows an example of calling an XMP program from a sequential Python program. In line 6, **xmp.Lib.spawn()** calls a parallel XMP program. The first argument is number of nodes in XMP. The last argument is an option for asynchronous operation. If it is true, processing may return to python before “**foo()**” completes. In line 7, **xmp.Lib.wait()** waits until “**foo()**” completes. In line 9, **xmp.Lib.elapsed_time()** returns processing time for “**foo()**”.

```

1 import xmp
2
3 lib = xmp.Lib("test.so")
4 args = ([1,2,3], [4,5,6])
5
6 job = lib.spawn(4, "foo", args, async = True)
7 job.wait()
8
9 print ("elapsed_time:{0}".format(job.elapsed_time()) + "[sec]")

```

Fig. 5 Example of spawning XMP program from sequential Python program [19]

An example of execution using Omni compiler is as follows. Note that a python program executes with one process, but an XMP program is executed with number of processes specified in code.

```
$ mpirun -np 1 python bar.py
```

4 Application to Order/Degree Problem

4.1 What Is Order/Degree Program

The order/degree problem is a problem that 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 (<http://research.nii.ac.jp/graphgolf>).

From the number of vertices (n) and degrees (d), the theoretical lower bounds of the diameter ($K_{n,d}$) and the ASPL ($L_{n,d}$) are calculated as follows [20]:

$$K_{n,d} = \begin{cases} \lceil \frac{n-1}{2} \rceil & \text{if } d = 2 \\ \lceil \log_{d-1}(\frac{(n-1)(d-2)}{d}) + 1 \rceil & \text{if } d > 2 \end{cases}$$

$$L_{n,d} = \begin{cases} 1 & \text{if } K_{n,d} = 1 \\ \frac{S_{n,d} + K_{n,d} R_{n,d}}{n-1} & \text{if } K_{n,d} \geq 2 \end{cases}$$

$$S_{n,d} = \sum_{i=1}^{K_{n,d}-1} id(d-1)^{i-1}$$

$$R_{n,d} = n - 1 - \sum_{i=1}^{K_{n,d}-1} d(d-1)^{i-1}$$

Figures 6 and 7 show examples of graphs with $n = 10$ and $d = 3$ and their distance matrices. The distance matrix indicates the shortest hop count between vertices. The diameter is the maximum value of the elements in the distance matrix. ASPL is the value obtained by dividing the total value of all elements by the number of elements $(n^2 - n)/2$. While the graph in Fig. 6 has random edges, the graph in Fig. 7 has edges optimized by our algorithm, which is described in Sect. 4.2. The diameter and ASPL of Fig. 7 are theoretical lower bounds.

In an effort to expand the order/degree problem into open science, the National Institute of Informatics has held a “Graph Golf” competition every year since 2015 to search for the smallest diameter and ASPL. A combination of several vertices and degrees is provided in each of those events. The competition has two categories: one is the General Graph Category, where vertices are placed freely, and the other is the Grid Graph Category, where vertices are placed on a two-dimensional grid. This section deals with the General Graph Category. Table 3 shows the combination of vertices and degrees used in 2017.

A python program “create-random.py” for the order/degree problem is available on the official Graph Golf website. The program outputs follow from the number of vertices and degrees. These calculations use the Python networkx package (<https://networkx.github.io>).

- Initial graph with random edges (the graph of Fig. 6 is created by this function)
- Calculation of diameter and ASPL
- Graph figure in Portable Network Graphics (PNG) format (the graphs shown Figs. 6 and 7 are created by this function)

4.2 Implementation

The “create-random.py” does not search out the smallest diameter and ASPL. Moreover, to obtain diameter and ASPL of a graph, it is necessary to calculate all of the shortest paths among its vertices. Although the “create-random.py” calculates the shortest paths using the **shortest_path_length** method of the networkx package, this method requires a significant amount of time.

To search for the smallest diameter and ASPL, we developed a GraphGolf code in both Python and XMP/C based on “create-random.py.” A Simulated Annealing (SA) [21, 22] algorithm is used for optimization. The shortest paths calculation is parallelized by XMP directives. Figure 8 shows a flow chart for the algorithm. While Python is used to create initial graph and output the figure, XMP/C is used to implement other parts.

Fig. 6 Diameter = 3, ASPL = 1.89

Figures 9 and 10 show a portion of our codes. In lines 6–10 of Fig. 9, the number of vertices and degrees are transformed to allow their use in the program. In lines 12–17, an initial graph is created and set a variable *arr*. The first element of the variable *arr* stores the number of vertices and the degree. In lines 19–20, the initial graph is passed to the **xmp_graphgolf** function of Fig. 10, which optimizes it. The result is saved in a variable *arr*, which is the same as one of the arguments. After line 22, the result is transformed into a figure. In lines 1–3 of Fig. 10, the XMP directives declare the template and node set, and then distributes the template onto

Fig. 7 Diameter = 2, ASPL = 1.67

	0	1	2	3	4	5	6	7	8	9
0		2	2	2	2	1	2	1	2	1
1			1	1	2	1	2	2	2	2
2				2	2	2	1	2	2	1
3					2	2	2	1	1	2
4						1	1	2	1	2
5							2	2	2	2
6								1	2	2
7									2	2
8										1
9										

Table 3 Combination of vertices and degrees of General Graph Category in 2017

Number of vertices	32	256	576	1344	4896	9344	88,128	98,304	100,000	100,000
Number of degrees	5	18	30	30	24	10	12	10	32	64

Fig. 8 Flow chart of an algorithm in Python and XMP/C

the node set in a block manner. In line 1, the “[:]” means that the template size is not fixed at this point. In line 9, the **template_fix** directive determines the template size *vertices*, which is the number of vertices. In lines 11–13, a diameter and ASPL are calculated. The calculation is a state of the “Compute energy” shown in Fig. 8. This calculation uses the top-down approach of the breadth-first search. In line 10, the **loop** directive parallelizes the loop statement to calculate each shortest path in parallel. In lines 14–15, the **reduction** directives aggregate a diameter and ASPL stored at each node.

```

1  import ctypes
2  import xmp
3  import networkx as nx
4  import argparse
5
6  argumentparser = argparse.ArgumentParser()
7  argumentparser.add_argument('vertex', type=int)
8  argumentparser.add_argument('degree', type=int)
9  vertex = args.vertex
10 degree = args.degree
11
12 arr = ((c_int * 2) * (vertex * degree / 2 + 1))()
13 arr[i][0] = vertex
14 arr[i][1] = degree
15 for i, l in enumerate(nx.generate_edgelist(g, data=False)):
16     arr[i+1][0] = int(l.split()[0])
17     arr[i+1][1] = int(l.split()[1])
18
19 lib = xmp.Lib("xmp_graphgolf.so")
20 lib.spawn(4, "xmp_graphgolf", arr, async = False)
21
22 image_name = "n"+str(vertex)+"d"+str(degree)+".png"
23 lines = []
24 for line in arr:
25     lines.append(str(line[0]) + "_" + str(line[1]))
26     G = nx.parse_edgelist(lines, nodetype = int)
27     save_image(G, image_name)
28
29 def save_image(g, filepath):
30     import matplotlib as mpl
31     mpl.use('Agg')
32     import matplotlib.pyplot as plt
33     layout = nx.circular_layout(g)
34     nx.draw(g, with_labels=False, node_size=50, linewidths=0, alpha=0.5, node_color='
        #3399ff', edge_color=' #666666', pos=layout)
35     plt.draw()
36     plt.savefig(filepath)

```

Fig. 9 Code in Python

4.3 Evaluation

To evaluate the performance of the Graph Golf application, we used the “coma” system located in University of Tsukuba, the specifications of which are shown in Table 4.

We measured the time required to calculate all of the shortest paths once by changing the number of nodes. We assigned 20 XMP nodes to each compute node, and used a medium-sized number of vertices and degrees ($n = 9344$, $d = 10$), as shown in Table 3.

```

1  #pragma xmp template t[:]
2  #pragma xmp nodes p[*]
3  #pragma xmp distribute t[block] onto p
4
5  void xmp_graphgolf(int *edge)
6  {
7      :
8      int vertices = edge[0];
9      #pragma xmp template_fix t[vertices]
10     #pragma xmp loop on t[i]
11     for(int i=0;i<vertices;i++){
12         : // Calculate diameter and ASPL
13     }
14     #pragma xmp reduction(+:ASPL)
15     #pragma xmp reduction(max:diameter)
16     :
17 }

```

Fig. 10 Code in XMP/C

Table 4 Coma system specifications

CPU	Intel Xeon-E5 2670v2 2.5 GHz 10 Cores × 2 Sockets
Memory	DDR3 1866 MHz 59.7 GB/s 64 GB
Network	InfiniBand FDR 7 GB/s
Software	intel/16.0.2, intelmpi/5.1.1, Omni compiler 1.2.1 Python 2.7.9, networkx 1.9

Figure 11 shows performance results where the bar graph shows the time measurements, and the line graph shows the parallel efficiency of one XMP node. The time for one XMP node is 123.17 s, while the time for 1280 XMP nodes (using 64 compute nodes) is 0.13 s, which is 921 times faster. Since the time using the Python networkx package is 148.83 s in one CPU core, XMP achieved a performance improvement of 21%.

From an examination of Fig. 11, we found that the parallel efficiency decreases as the number of nodes increases. We consider that the following reasons are responsible for the decrease:

- The ratio of communication time increases. Figure 12 shows the ratio of communication time and calculation time to the total time. As the number of nodes increases, the proportion of communication time also increases.
- The parallelized loop lengths are non-uniform. The length of the loop statement in line 11 of Fig. 10 is 9344, which is the same as the number of vertices. Since the length is divided by the number of nodes, the length non-uniformity increases as the number of nodes increases.

Fig. 11 Performance evaluation

Fig. 12 Calculation and communication ratio

5 Conclusion

This chapter describes how to use linkage functions between XMP and MPI library in Omni compiler. Moreover, it also describes how to call an XMP program from a Python program. Users can call functions written in these languages with a simple

procedure. Since many existing parallel applications are written in MPI library, these functions can be useful for extending existing applications. Furthermore, it will be possible to effectively use the high functionality of Python.

References

1. M. Nakao et al., Linkage of XcalableMP and Python languages for high productivity on HPC cluster system, in *Proceedings of Workshops of HPC Asia, No. 9* (2018), pp.39–47
2. F. Cantonnnet et al., Productivity analysis of the UPC language, in *18th International Parallel and Distributed Processing Symposium* (2004), pp.254–260
3. K. Yelick et al., Productivity and performance using partitioned global address space languages, in *Proceedings of the International Workshop on Parallel Symbolic Computation* (2007)
4. A.I. Stone et al., Evaluating coarray Fortran with the CGPOP Miniapp, in *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models* (2011)
5. M. Nakao et al., Implementation and evaluation of the HPC challenge benchmark in the XcalableMP PGAS language. *Int. J. High Perform. Comput. Appl.* **33**, 110–123 (2017)
6. J. Jithin et al., Unifying UPC and MPI runtimes: experience with MVAPICH, in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model* (2010), pp. 5:1–5:10
7. M. Nakao et al., Productivity and performance of the HPC challenge benchmarks with the XcalableMP PGAS language, in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model* (2013), pp. 157–171
8. M. Nakao et al., Productivity and performance of global-view programming with XcalableMP PGAS language, in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2012), pp. 402–409
9. M. Nakao et al., XcalableACC: extension of XcalableMP PGAS language using OpenACC for accelerator clusters, in *Proceedings of the First Workshop on Accelerator Programming Using Directives* (2014), pp.27–36
10. M. Nakao et al., Implementing lattice QCD application with XcalableACC language on accelerated cluster, in *IEEE International Conference on Cluster Computing (CLUSTER)* (2017), pp. 429–438
11. M. Nakao et al., Evaluation of XcalableACC with tightly coupled accelerators/InfiniBand hybrid communication on accelerated cluster. *Int. J. High Perform. Comput. Appl.* **33**, 109434201882116 (2019)
12. R.W. Numrich et al., Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum* **17**(2), 1–31 (1998)
13. M. Nowicki et al., PCJ - Java library for high performance computing in PGAS model, in *International Conference on High Performance Computing & Simulation* (2014), pp. 202–209
14. Y. Zheng et al., UPC++: A PGAS Extension for C++, in *IEEE 28th International Parallel and Distributed Processing Symposium* (2014), pp. 1105–1114
15. K. Vivek et al., HabaneroUPC++: a compiler-free PGAS library, in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, No. 5* (2014), pp. 5:1–5:10
16. C. Philippe et al., X10: an object-oriented approach to non-uniform cluster computing, in *OOPSLA '05 Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, vol. 40, No. 10 (2005), pp. 519–538
17. B.L. Chamberlain et al., Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.* **21**(3), 291–312 (2007)

18. K. Furlinger et al., DASH: a C++ PGAS library for distributed data structures and parallel algorithms, in *IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)* (2016), pp. 983–990
19. M. Nakao et al., Linkage of XcalableMP and Python languages for high productivity on HPC cluster system, in *Workshop on PGAS Programming Models: Experiences and Implementations* (2018)
20. V.G. Cerf et al., A lower bound on the average shortest path length in regular graphs. *Networks* **4**(4), 335–342 (1974)
21. M. Nicholas et al., Equation of state calculations by fast computing machines. *J. Chem. Phys.* **21**(6), 1087–1092 (1953)
22. S. Kirkpatrick et al., Optimization by simulated annealing. *Science* **220**(4598), 671–680 (1983)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.