Fundamentals of High-Performance Computing for Finite Element Analysis

Hiroshi Kawai, Masao Ogino, Ryuji Shioya, and Shinobu Yoshimura

Abstract As introduction, high-performance computing (HPC) technology for finite element analysis is explained. First, general notions, tips and techniques which are useful for the programming on modern HPC and supercomputing environments are introduced. Here, as a supercomputer, the authors assume mainly a distributed memory parallel computer with each computational node having one or more multi-core scalar processors. Both hardware and software aspects are covered, with important concepts such as MPI, OpenMP and other compiler directives, as well as network interconnect, cache, memory bandwidth, SIMD and peak performance ratio. There are also some tips and advices about the software development process specific for the supercomputers, such as software design, testing and debugging, and profiling. Then, some important ideas and techniques for the development of FE-based simulation code running on supercomputers, such as the selection of linear algebraic solvers, domain decomposition method (DDM), element-by-element (EBE), as well as mesh generation and visualization are explained. This chapter

11th World Congress on Computational Mechanics (WCCM XI)
5th European Conference on Computational Mechanics (ECCM V)
6th European Conference on Computational Fluid Dynamics (ECFD VI)
20–25 July 2014, Barcelona, Spain

H. Kawai (✉)
Tokyo University of Science-Suwa, 5000-1 Toyohira, Chino-shi, Nagano 391-0292, Japan
e-mail: kawai@rs.tus.ac.jp

M. Ogino
Nagoya University, Furo-cho, Chikusa-ku, Nagoya-shi, Aichi 464-8601, Japan
e-mail: masao.ogino@cc.nagoya-u.ac.jp

R. Shioya
Toyo University, 2100 Kujirai, Kawagoe-shi, Saitama 350-8585, Japan
e-mail: shioya@toyo.jp

S. Yoshimura
The University of Tokyo, 7-3-1 Hongo, Bunkyo, Tokyo 113-8656, Japan
e-mail: yoshi@sys.t.u-tokyo.ac.jp

S. Yoshimura et al. (eds.), *High-Performance Computing for Structural Mechanics and Earthquake/Tsunami Engineering*, Springer Tracts in Mechanical Engineering,
DOI 10.1007/978-3-319-21048-3_1

could also serve as the introduction of HPC and supercomputing to the following chapters dealing with more specific problems and schemes.

1 Introduction

Welcome to the supercomputing world! Here in this chapter, we explain various issues and techniques related to high-performance computing (HPC), and especially, how to write your finite element code running on a modern supercomputer.

Before understanding the current modern HPC technology, it is beneficial to know what had already happened in the past. Here we explain a brief history of supercomputing. By the way, the keyword "supercomputer" is generally defined as "a computer which is at least 1000 times faster than an ordinary computer like note PC and tablet."

About three decades ago, a supercomputer at that moment was a vector computer, which equips a vector-processing unit. To take advantage of the vector supercomputer, we had to vectorize our program. Although it forced us to modify some portions of our code, still it was a good old times because we needed to focus on just only one thing, "vectorization." Later on, shared memory parallel processing was added to the vector supercomputer. And compiler directive approaches for parallelization, which eventually evolved into OpenMP, also appeared at that time. On the other hand, although not successful, another type of supercomputers based on single instruction multiple data (SIMD) architecture was tried also. The usage of these SIMD machines was almost similar to that of the vector processor, based on the parallelization of loops.

Then, the 2nd wave of supercomputing, so-called massively parallel processors (MPP) arrived. It was a distributed memory parallel computer, and classified as multiple instruction multiple data (MIMD). Various kinds of message passing approaches had been appearing, but finally they were merged into the MPI standard. It actually had changed the fundamental design of our finite element code drastically. The notion of domain decomposition became MUST to exploit the potential power of MPP. Later on, MPP was taken over by so-called attack of killer micro, PC cluster. This means, a supercomputer became just a bunch of PC boxes connected each other by high-speed network. Each "box," a computational node of the supercomputer, contained one or a few scalar processors and their own memory space. Still, it was relatively easier than now to write code because we needed to think about MPI communication and domain decomposition only.

Currently, we are just in the midst of the 3rd on-going wave of supercomputing. Some people call it the era of accelerator in general, or GPU in more specific. Other people say multi-core or many-core age. In either way, while the number of computational nodes in one supercomputer system had gradually grown to hundreds of thousands, each computational node itself had become a kind of "beast."

So now, we are in the truly massively parallel era. Here, "massively parallel" means so many that it is virtually uncountable, while at the moment of the 2nd wave

a typical MPP system had processors of only a thousand or less. The number of cores in a current peta-scale supercomputer is already reaching near 1 million, and it could increase to 1 billion in an exa-scale system which will soon appear in very near future.

On the other hand, readers should not forget that your desktop PC or workstation has also become very, very powerful. A typical workstation, which equips GPU or many-core accelerator, enjoys more than 1 TFLOPS. Welcome to Desktop Supercomputing! Unfortunately, something wrong about this machine is that a typical simulation code developed by a researcher runs only 1 GFLOPS on the machine, if without any performance tuning effort. It is simply because the clock frequency of these processors itself is only around a few GHz. Anyway, between Giga and Tera, 1000 times of difference! Is it a kind of cheat or FLOPS fraud? That's why we call it a "beast," which cannot be tamed easily. The same thing applies to supercomputing because currently the architecture of each computational node of a supercomputer is virtually the same as that of a desktop PC or a workstation. Thus, a bunch of "beasts" forms a "monster," the massively parallel many-core supercomputer.

Here in this chapter, first, supercomputing hardware and software technologies are explained from programmer's point of view. Then, the more specific topic, "how can I tune up my finite element code?" is explained.

Because of this limited space, each topic is explained only briefly, without enough illustrations and examples. Through this chapter, however, readers can find several important keywords, which are typically related to frequently occurring troubles and issues during the development of simulation code. Starting from these keywords, you can understand the challenge and also find the solution to them. Hopefully, this chapter could serve you as a survival guide in the supercomputing jungle.

2 Hardware

In this section, we start from hardware issues first. Here, we focus on three hardware elements: processor, memory, and network interconnect. To highlight the potential implication of these components to the performance design of simulation code, we try to explain them from programmer's point of view as much as possible.

A typical modern supercomputer consists of many computational nodes. They are connected each other by high-speed network interconnect. Each computational node holds one or a few processor chips and its associated memory chips, thus forming a distributed memory parallel computer. As a result, both intra-node and inter-node performance are important for the performance design of simulation code.

2.1 Processor

Nowadays, a processor can issue multiple instructions and execute multiple floating-point operations for each clock cycle. For example, a pair of multiplication and addition, or a set of multiply-add pairs packed in one SIMD instruction. It is called instruction-level parallelism (ILP). For example, if a processor can handle eight double precision (DP) floating-point operations per clock, and it runs with 2 GHz, its DP peak performance is $8 \times 2 = 16$ GFLOPS. Theoretically, it is nice, if it always works so. The reality, however, is not so simple. How can we write code which keeps handling four add and four multiply operations in every clock? Anywhere branch or procedure call appears, it doesn't work so. It is so-called ILP wall problem. This is one of the major issues to widen the gap between the ideal, peak performance of the hardware and the actual, sustained performance when running real application code on the hardware.

For example, let's think about SIMD instruction. The SIMD mechanism allows multiple operations of the same type to be packed in one instruction and executed at once. The operation can be addition, subtraction, multiplication or division, or even a pair of multiply and add/subtract, so-called fused multiply-add (FMA). If a processor can handle a set of four FMA operations, which is called 4-way SIMD, it sums up to eight operations per clock. A modern processor can handle 4, 8 or 16 operations in each clock. This number tends to grow up further as the density of transistors in semiconductor design increases (Moore's law).

Moreover, the above fact is merely for one core in a processor chip. In recent years, the number of cores in a chip had also increased. Here, "core" means "processor" in a traditional sense. Until recently, one processor corresponds to just one chip, or sometimes to multiple chips in case of very complicated processor design. But nowadays, because of Moore's law, one processor chip can hold multiple processors. To avoid ambiguity, the keyword "processor" is renamed to "core." Thus, it is called "multi-core." Currently, a typical scalar processor has 2, 4, 6 or 8 cores in one chip, although there are some "many-core" chips having more than ten cores. They can be regarded as a shared memory computer in one chip. Either MPI or OpenMP can be used for parallel programming. However, multi-core has more serious implication to memory access, which will be explained soon in the next section.

2.2 Memory

Even if the processor can perform multiple operations per clock, in reality, it is no meaning if data cannot be supplied from the memory system into the processor in enough speed.

Let's start from a simple example of adding two vectors, like "$c(i) = a(i) + b(i)$." Also assume that a processor chip has 8 cores, and each core equips 4-way SIMD

and perform four add operations per clock. If the chip runs in 3 GHz, it can perform 4 operations $\times$ 8 cores $\times$ 3 G times per second = 96 GFLOPS. Not so bad. Then, think about data also. Data I/O required to read/write is, assuming double precision, 8 bytes per number, 8 bytes $\times$ 3 I/O (2 read, 1 write) $\times$ 96 G FP operations per second = 2304 GB/s. Currently, no memory system can supply data in the speed of such vast amount. Usually, only from 30 to 300 GB/s at most. It is so-called memory wall problem.

Memory access pattern is the important keyword to understand the performance characteristics of the memory system. Let's imagine the memory access patterns in a loop. It can be sequential, stride or random. In case of random access, actually it may be indirect index access. For example, in the context of finite element code, touching nodal data in each element loop is a typical example of this case. For each finite element of the loop, its associated nodal data must be accessed through the element connectivity data, in a form like "data(index(i))." Of these access patterns, typically, the sequential access pattern is the fastest. On the other hand, stride and indirect index accesses have some problems. Therefore, if possible, data should be rearranged so that the sequential access pattern dominates.

For example, let's consider an element-by-element (EBE) loop. In this loop, many floating-point operations have to be performed for each finite element. However, its dominant memory access pattern is indirect index access. This memory access pattern may prohibit efficient execution of the loop body. To overcome the issue, an extra element loop is added just before the main loop, and in this loop, nodal data are first copied into another element-wise array. Then, in the main element loop, using this extra element-wise array, all the data access can be done in element-wise manner, thus, sequentially. It can allow compiler to vectorize the main loop. It is a useful technique if EBE involves relatively heavy calculation.

Cache memory is very important in the design of modern scalar processors, and it is related to a memory access pattern called data locality. In short, the cache memory is a fast but small special memory region, and it is separated from the main memory. When the processor tries to read a small amount of data from the main memory, the fetched data is once copied into the cache memory automatically. Then, suppose this small data becomes necessary again. If the data still remains on the cache memory, instead of reading the original data again from the slow main memory, it is sufficient to access this copy from the fast cache. This means, it is better to keep using this small amount of data as much as possible. If the size of frequently used data is larger than the size of the cache, however, it doesn't work in this way. While reading it, most of data are kicked out of the cache, and it ends up virtually accessing to the main memory. Thus, the cache mechanism works only if "a relatively small amount of data is accessed repeatedly, repeatedly and repeatedly."

Then, how can we utilize such tricky cache mechanism? There is a well-known technique called blocking or tiling. In cache blocking, a big data region is divided into many small data blocks first. For example, in case of a matrix, it is decomposed into multiple sub-matrices. They look like tiles. Using associated blocking algorithm, once a small block is read from the main memory, it is utilized repeatedly before the next one is needed.

From programmer's point of view, there is one easy way to implement this cache blocking. First, prepare explicitly small arrays as working area. The total size of these arrays must fit on the cache memory. Then, copy data subset from the big array into these small arrays. Using only this working area, perform relatively heavy calculation. And finally, move the result back to the big array. Was it easy? If this way of coding style works fine with your case, you are lucky to obtain a blocking version of your algorithm.

However, if any blocking algorithm cannot be devised, how come? In this case, sadly, you have to directly tackle against the slowness of the main memory. Now, memory bandwidth becomes a serious issue.

To consider the memory bandwidth problem, it is useful to understand the keyword, B/F ratio. B/F ratio measures how much data can be read/write from/to the main memory for each execution of floating-point operation. This ratio typically assumes a specific numerical algorithm. If the actual B/F ratio value is no less than the value originally required by the algorithm, it is OK. Otherwise, there is memory bandwidth bottleneck. You may switch to much better supercomputer, if possible. Or you might be forced to modify your algorithm.

For example, let's think about matrix–vector product. Suppose the size of matrix and vector as N. Code can be written as "$c(i) = c(i) + a(i, j) * b(j)$," with "$i$" as outer loop and "$j$" as inner loop. Assuming "$c(i)$" fits on processor register, there are two read accesses, "$a(i, j)$" and "$b(j)$." Therefore, assuming double precision, 8 bytes per number, the total amount of data I/O is $16\,N^2$ bytes, while that of floating-point operations is $2\,N^2$ operations. In this case, B/F ratio becomes 8. Furthermore, a blocking algorithm can be applied to the case of this matrix–vector product. Because vector "$b(j)$" can be moved on cache, so B/F ratio drops to 4. Still, a supercomputer with a very strong memory system, having B/F ratio 4, is required to perform this algorithm efficiently. Otherwise, the plenty of extra floating-point hardware resources available in processor side is merely wasted.

In reality, the current situation is very bad. Most of scalar processors can supply B/F ratio of only from 0.1 to 0.5 at most. This means, when the matrix–vector product is performed, peak performance ratio becomes only a few percent! Is it a kind of cheat or FLOPS fraud? That's why it is so-called the issue of "memory wall."

The recent rapid growth of the raw computational power of a processor chip in terms of floating-point operations, caused by the increase of both the number of processor cores and the number of SIMD ways, has made this memory bandwidth issue more serious and desperate. Thanks to Moore's law, the growth pace of the FP capability is far exceeding that of the memory bandwidth. While each core has its own cache memory (roughly speaking), the path to the main memory is basically shared among all the cores, because it is "shared memory." As a result, only the algorithms which can take advantage of cache mechanism can keep up with the growth of the computational power, while B/F ratio will drop further.

2.3 Network Interconnect

Of the three key hardware components, processor, memory and network interconnect, the former two, explained already, are also related to the performance design of simulation software running on a desktop PC and a workstation. In this sense, the last one, network interconnect, is the only key component unique in supercomputing. In modern supercomputing, distributed memory parallel architecture is the primary hardware architecture. A supercomputer is composed of multiple computational nodes. Each of them has its own memory space. A computational node cannot directly access memory owned by other nodes. Instead, it has to send/receive data to/from others through this high-speed network interconnect.

As the architecture of the network interconnect, nowadays torus or fat tree is utilized for a high-end supercomputer, while for low-end HPC environment like a PC cluster, switching hardware is used. The fat tree can be said to be cascade of switches. On the other hand, in case of the torus architecture, each node has very high bandwidth connection directly, but to only a few neighbouring nodes. To communicate to any node other than these neighbour nodes, data has to be relayed through one or a few intermediate nodes. It is typically used to connect a relatively large number of nodes.

To use such a distributed memory parallel computer, care must be taken about communication patterns. Because some of the communication patterns are either inherently efficient or strengthened by additional hardware mechanism, the use of these special patterns should be considered first in the design of parallel software.

Global communication, such as barrier, broadcast and reduce operations are frequently used in various kinds of parallel programs. High-speed interconnect supports those patterns directly in hardware level. Therefore, if these communication patterns are recognized in your code, instead of ordinary send/receive protocol, the corresponding special communication API or directives should be used. They are the only routes to take advantage of the special network hardware mechanism.

On the other hand, neighbour communication is another important pattern. To understand it, let's consider the difference between bus and switch. Suppose there are four nodes, A, B, C and D. In case of the bus architecture, nodes C and D cannot communicate while nodes A and B are talking. In case of the switching architecture, however, communication of A–B and C–D can work simultaneously. It can be easily extended to the case of many, many nodes: A–B, C–D, E–F, G–H, and so on. All of them can be invoked at once. Instead, if node A wants to receive data from more than one, for example, nodes B and C, there will be conflict.

The neighbour communication pattern is very important in case of macro-scale, continuum mechanics-based simulation. In this type of simulation, a whole analysis domain can be decomposed into multiple subdomains. Typically, communication between neighbouring subdomains frequently occurs. It can be represented as the neighbour communication pattern. Thus, well-designed code for the continuum mechanics field can easily scale on a supercomputer.

As for the neighbour communication pattern, it is better to understand the keyword, volume to area ratio. This means the ratio between amount of calculation per subdomain as a volume, and that of the associated interface boundaries as an area. As the problem size grows, this ratio is expected to grow as well, because of comparison between volume N^3 and area N^2. That means, the bigger the problem size is, the easier to parallelize. You can always enjoy good parallel efficiency, if you specify the problem size big enough. Of course, there are some issues, too long execution time and lack of memory.

In reality, however, you cannot wait so long. And more often, you may want to solve the problem just faster, with the problem size fixed. In this case, further intensive performance optimization of the communication routines will be required, such as overlap of communication and computation. There is a notorious saying that, "Using a supercomputer, you cannot shrink time. Instead, you can grow problem size."

3 Software

In the previous section, the hardware architecture of modern supercomputers is described. It is also necessary to mention about more software-related topics, especially, the software development process and environment of supercomputing applications. In this section, starting from software design principle, basic programming models, programming languages, compilers, libraries and tools are explained.

3.1 Performance-Centric Software Design

Anyway, why using a supercomputer? Of course, you want to run your simulation code faster, and also you may want to make the problem size much larger, with a finer mesh, more detailed geometry, much wider coverage of the analysis domain, and more accurate modelling. If you can just use your original, un-tuned program as is without modification, you will be very happy. The reality is much harder, however, and you will be forced to optimize your code, to take advantage of the potential power of the supercomputer.

Usually, the performance optimization task is only a small portion of the whole software development cycle. Testing, debugging, trying to find much better design for maintenance and extension in the future, and moreover, capturing true user requirement (for example, in case of simulation, appropriate modelling of target physical phenomena) are more important and essential tasks. As a result, the performance tuning is often ignored. Once you have decided to use a supercomputer, however, this task suddenly becomes more than essential, and the first priority. It is simply because, if you failed to gain performance boost from the use of the

supercomputer, it would be just a waste of time and money. The sole purpose of using the supercomputer is, to make your code faster.

In the good old days, money could buy everything. All you need to do was, just to prepare budget for renting a supercomputer. With the same, original code, you could have enjoyed much faster simulation. Nowadays, however, money alone is not enough. You need also time to modify your code. To do so, the design of the code should be re-considered. "Why does this code run so slowly?" Then, here comes the performance-centric software design.

To incorporate the performance tuning tasks into the software design process up-front, there is an established systematic approach. It consists of three tasks, finding hot spot, scaling test and unit node test.

First, hot spot has to be identified. Here, the hot spot means routines or portion of the code which consume the majority of execution time. In most of simulation programs, it could easily happen that only a few routines or loops spend more than 99 % of time. Then, it is good enough to focus your effort only on these hot spot routines and loops. They may also be called as performance kernels.

Next, in case of parallel processing, scaling test is performed to check the scalability of the code and also to identify the performance bottleneck if there exists. There are two types of tests, strong scaling and weak scaling. The former test keeps the problem size unchanged, and it can be used to make sure the computational time shrinks as the number of processors increases. On the other hand, in case of the latter one, both the number of processors and the problem size are increased. In an ideal case, the execution time does not change. Instead, if the execution time also increases, there would be the bottleneck against parallelization.

Recently, not only the scalability, but also the concept of peak performance ratio becomes more and more important. This is the ratio between the peak performance of the supercomputer as hardware and the actual speed of application code, both measured in FLOPS. For example, if the simulation code runs with the speed of 100 TFLOPS on the machine with the peak performance 1 PFLOPS, the ratio is 10 %. Usually, the scalability is pre-requisite for the high peak performance ratio, but more is required to obtain good peak ratio. The unit node performance, the performance measured on a single computational node, cannot be ignored as well. If the peak ratio on a single node is less than 5 %, the ratio value using the whole system can be also 5 % in the best case, and it is usually less than this value. Thus, the unit node performance is another important measure, and the unit node test has to be carried out.

Anyway, what the unit node performance actually means? In case of a modern supercomputer, the hardware architecture of each computational node is almost the same as that of a brand new PC on your desktop. Therefore, before porting your code onto the supercomputer, it is a good idea to tune the code on your desktop PC first. However, this task itself is getting harder and harder nowadays.

3.2 Programming Model

To consider the software design for modern supercomputing applications, there are some special programming models. Each model covers a specific HPC hardware architecture. Here, we introduce two important programming models, hybrid parallel and data parallel.

Suppose if a supercomputer is based on multi-core scalar processors, there are two choices in programming models, flat MPI and hybrid parallel. In case of flat MPI, it is just sufficient to re-use your MPI-based code without modification, but you need to set the number of MPI ranks as same as the number of total cores using. On the other hand, the hybrid parallel programming model is needed mainly for two reasons. One is, to obtain the maximum parallel efficiency. The other one is, to utilize processor cores of sub-million order or more.

Theoretically, it is better to utilize OpenMP for intra-node communication, while MPI is used for inter-node one. In reality, it depends on applications and problems. Typical trend is, when the problem size is relatively small while the number of cores used is many, hybrid parallel programming model works well. If the parallel efficiency is already good enough with the current flat MPI implementation, however, the additional gain obtained from the hybrid parallel approach is marginal. In case of domain decomposition-based approaches, because of the volume to area ratio, explained before, the bigger the problem size is, the better the parallel efficiency is.

On the other hand, if luckily you have a chance to utilize a world-class supercomputer, then unluckily you will be forced to adopt the hybrid parallel programming model, simply because the flat MPI model does not work in this environment. Currently, no MPI implementation seems to work well more than ten thousand MPI processes. The combination of MPI and OpenMP is the only way to utilize millions of cores available on such a top-end supercomputer.

Data parallel programming model, or SIMD, is the programming model useful for GPU, and also for SIMD instruction in modern scalar processors. SIMD is single instruction multiple data. It usually involves the use of either compiler directives, such as SIMD vectorization and OpenACC, or extension to existing programming languages, such as CUDA and OpenCL. In the former cases, a loop is annotated by compiler directives with additional information necessary to parallelize. In case of CUDA and OpenCL, the loop body of a loop is extracted as a GPU-local routine, which is associated to a thread when running on the GPU.

3.3 Programming Language and Compiler

To port your simulation code onto a supercomputer, more or less some amount of code modification effort will be required. It typically involves inserting special compiler directives such as vectorization, OpenMP and OpenACC into the code,

and calling functions/subroutines of HPC-related libraries such as MPI and BLAS. In the worst case, instead of keep using ordinary programming languages such as Fortran, C and C++, special programming languages dedicated for specific HPC environments might have to be employed, and the code would be re-written completely from scratch.

If the hybrid parallel programming model, explained before, is employed, OpenMP is the primary choice for intra-node parallelization. OpenMP is mainly a set of compiler directives designed for thread-level parallelization on shared memory parallel environment. It fits well on modern multi-core or many-core scalar processors. Programmer specifies OpenMP compiler directives on each loop which can be parallelized. Instead, if your code is relatively simple and you are lucky, the automatic parallelization capability of compiler may also work well.

OpenMP can be utilized if a computational node contains multiple processor chips, each of which may further be multi-core. In this case, non-uniform memory access (NUMA) memory architecture should be considered. Each processor chip has direct connection to memory chips. That means, each processor has its own memory. Access to own memory is fast, while access to other processor's memory tends to be much slower. Actually, it can be considered as distributed memory. From programmer's point of view, first touch principle can be applied. Data region "touched" first by a thread running on a processor is owned and managed by the processor, and it is allocated on its memory. Through the capability of OpenMP, thread ID information is required to handle this situation.

Recently, the impact of SIMD instructions is becoming more and more important for the performance design in supercomputing. To utilize the SIMD instructions, there are three ways. One is, of course, directly use assembly language. Most of people, including us, would want to ignore the first choice. Then, the second option is, to use SIMD intrinsic functions/subroutines. Still, this option is very tedious. Therefore, the third option, compiler-driven vectorization, is more practical for the most of programmers.

Compiler-driven vectorization technique for SIMD is very similar to the so-called "vectorization" in the vector supercomputer age. The idea itself has been very simple, inserting compiler directives just before loops which can be vectorized. However, there is one big difference between the current SIMD vectorization (also called, short vectorization) and the old predecessor. While the vector processor can aggressively read/write data from/to main memory, the SIMD mechanism of the modern scalar processor is effective only to data on cache memory. That means, before considering the use of SIMD instructions, first, you need to put your data on cache.

So, how to put your data on cache? It depends on code design and numerical algorithm itself. We have already explained that small amount of data with the size fitting on the cache has to be accessed repeatedly. Some algorithms fit well on cache architecture using the tiling/blocking technique, while others do not. For example, addition of vector and vector cannot utilize cache at all. Then, in case of matrix–vector product, at least the vector data can be placed on cache, while the matrix side cannot be. Still by applying these cache-aware techniques, the code can

be significantly accelerated on a modern cache-based scalar processor, because the amount of data to be read from the main memory can be cut by half. Finally, in case of matrix–matrix product, with cache blocking, the majority of the working data set can be placed on cache, which makes further application of SIMD vectorization very effective. Thus, whether SIMD and cache blocking are effective or not heavily depends on the nature of the numerical algorithm employed and higher level design.

We need to mention about accelerators. The use of accelerators such as GPU and many-core chip is gradually starting. On these environments, special programming languages or extension to existing languages, such as CUDA and OpenCL, may have to be utilized. Directive-based programming models such as OpenACC, which require much less work, are also becoming available.

3.4 Library

Some HPC-related libraries are available. They may have to be employed into your code if necessary.

Assuming the use of a modern distributed memory parallel supercomputer, for inter-node parallelization, message passing interface (MPI) is the primary choice. It is a message passing-based communication library among computational nodes. In case of macro-scale simulation, domain decomposition is required. Especially in case of unstructured grid or mesh-free/particle, identifying the boundary region between subdomains is a bit complicated task. Instead of this really tedious approach, some people have started using PGAS languages. Using these special languages, programmer can parallelize code in much similar way as shared memory environments like OpenMP.

In addition, the knowledge of linear algebra is often required. It is very useful if matrix and vector-related libraries are available. Linear algebraic solver and eigenvalue solver libraries are also important.

Basic linear algebraic subroutines (BLAS) is one of the famous libraries for matrix and vector operations in the basic and fundamental level, such as various kinds of vector–vector, matrix–vector and matrix–matrix operations. The scene behind the fact that recently this library becomes very important is, however, because nowadays highly tuned versions of this library prepared by hardware vendors themselves are available. Especially, level-3 BLAS routines, which are related to matrix–matrix operations, are really important. Usually, inside the vendor-provided library, these routines are implemented as cache-aware and SIMD-vectorized. And, by using these optimized routines, high peak ratio can be easily obtained. Although BLAS is designed for dense matrix, recently, its sparse matrix version becomes available.

As for the linear algebraic solver, a variety of libraries are available. LAPACK contains direct solvers for dense and banded matrices [1]. ScaLAPACK is also available for MPI-parallel environments. In case of sparse solvers, some are direct and others are iterative. SuperLU, MUMPS, PARDISO and WSMP are

examples of sparse direct solvers, while PETSc is an example of iterative solver libraries.

3.5 Supporting Environment and Tools

Other than compiler, the most important programming tool for supercomputing is profiler. This tool can be used to identify hot spot of the code. Finding the hot spot is vital in performance-centric software design, described before. Moreover, the profiler can measure how fast your code runs on the supercomputer. That means, measuring FLOPS values.

They say that, "without measurement, no improvement." This is absolutely right. On a supercomputer, the profiler is a survival tool. The concept of peak performance ratio has no meaning if one cannot measure FLOPS accurately. But if there is no such profiler available in your environment, how can you survive? In theory, it is very simple. First, have your program count the number of total floating-point operations executed inside your code. Then, divide this number by the execution time measured in seconds. That's it. Yes, this is floating-point operation per second (FLOPS)! Actually, this task is very tedious as you imagine. That's why the profiler is essential in HPC.

Compared to the profiler, debugger, which is usually another essential tool in more general software development environment, seems to be of little use, at least for us. We mean, although we can often find the situation where the debugger is useful in this environment, it is far from enough. To supplement the debugger, we can suggest two ways for debugging.

One is, to test/debug your code on a single processor and single core environment first, and keep doing it until all the bugs disappear. Virtually, invoking the debugger on the supercomputer is too late to find a bug.

The other is, to prepare additional debugging mechanism embedded in your code explicitly. For example, it is convenient to add the functionality to compare internal data set between single version and parallel one. While it is also a tedious task and even the meaning of data comparison itself depends heavily on each problem, still it is a great help if bugs are somewhere in your code and also such functionality is ready. A cool and fancy parallel debugger would not help you so much.

In addition to these headaches, you might face more mysterious bugs on the supercomputer. Although thread-level parallelization like OpenMP and CUDA helps you write parallel code much easier, it could be often the source of serious bugs, such as synchronization and subtle timing issues. You might also encounter the bug of compiler, profiler, libraries and OS itself. For the most of the cases, however, it will end up just caused by your mere misunderstanding about the actual behaviour of these tools. But, based on our experiences, in very rare occasions, you might actually see these bugs!

Anyway, even on ordinary, single core desktop PC environment, just using the debugger, can you actually find the bug of a minus operator, which is wrong and

should be plus instead, from a bunch of formulas in your code? Probably, intensive and thorough code review, and verification test of the code against theoretical or experimental data are required. Thus, in the development of simulation code, the debugging and testing task itself has been very important and critical from long, long time ago. It will be so in near future also, and perhaps forever. It is simply because the debugger does not work well. It is noticed that, on the supercomputer, without thorough and deliberate preparation for debugging, you could see the real hell of debugging. Please be aware of this fact.

3.6 Other Issues

Here, we mention about some issues, advices and programming tips to take advantage of these modern technologies in HPC and supercomputing.

3.6.1 Estimation and Measurement

For any scientific activity, measurement is an important task. Also, prediction and estimation of the behaviour of the target object is useful to establish background theory behind the phenomena. Such kind of so-called Plan (prediction and estimation)—Do (experiment)—See (measurement) cycle is also effective for the performance optimization process of supercomputing applications.

For example, have you ever estimate and measure actually about the following things? How much is the calculation cost and how much is the communication cost? How much data is sent to/received from neighbour nodes? In this loop, how much data is read/written from main memory, while how many FP operations are carried out? Without estimation, there can be no reasoning and theory. And also, without measurement, no justification or correction of them.

In HPC and supercomputing, “how many” is very important. How many times the code becomes faster than before? How large problem the code can handle in this supercomputer? And, in the end, how long does it take to run this job? It is quite different from using other hardware devices and computer environments. For example, to utilize a 3-D graphics device, it is anyway fine if you can draw lines and triangles. The performance is a secondary issue, and it should be handled only if necessary. That means, in 3-D graphics, “can do” is more important than “how many.” However, in case of a supercomputer, merely using this expensive machine has virtually no meaning. As we mentioned before, the sole purpose of using the supercomputer is, making your code faster.

3.6.2 Memory Access Patterns

In modern HPC environments, memory is one of the slowest components, and it can easily become the performance bottleneck. Therefore, it is useful to design your code considering memory access pattern. Nowadays, the execution time of the code can be roughly estimated from memory access cost, rather than the number of FP operations.

Whether to main memory, cache, SSD or hard disk, data access pattern to these memory devices virtually defines the performance. If possible, sequential access is the best. Otherwise, certain amount of penalty should be expected, depending on the type of devices.

Because of memory hierarchy mechanism, automatic data movement between a slow device and a fast one may occur also. Data access patterns considering data locality control the actual behaviour of this mechanism, and you can identify "where is this data actually placed now?"

3.6.3 Implicit or Explicit?

When discussing about issues of any programming language and compiler in HPC and supercomputing, there is the argument of so-called implicit or explicit. In case of implicit, the behaviour of the code is implicitly defined in a certain, hopefully convenient for the programmer, way. On the other hand, in case of explicit, the programmer has to explicitly define the behaviour of the code one by one, line by line.

At a glance, the former one is easier and better. The latter one is usually so tedious and error-prone, people tend to choose the former implicit or automatic approach. The question is, "in the end, did your code actually get faster?" As we mentioned before, using a supercomputer is not just using it, but also, "how fast" your code runs on the supercomputer. In this sense, the implicit approach is something in the middle. Your code may become faster, or not so much. It actually depends. It depends on the design of your code.

It is our opinion that, whether implicit or explicit, the code performance is virtually defined by the design of the code. Sometimes, it heavily depends on the type of numerical algorithm adopted in the code. And the difference between implicit and explicit is, while deep understanding about the implication of the code design is required "before" using the explicit approach, in case of implicit, you can easily jump start into the coding phase without thinking about these design issues. As a result, if lucky, it works fine. Otherwise, and for the most of the cases in our experiences, you are forced to re-consider the design of your code thoroughly from scratch, or just give up and retreat from supercomputing.

Actually, in terms of the productivity, implicit approaches, such as OpenMP and OpenACC, are far more superior than corresponding explicit approaches of much lower level, such as p-thread and CUDA. Same is applied to SIMD vectorization driven by compiler directives, over assembly languages. As for the discussion

about MPI versus PGAS languages, we cannot say here, because we don't have so much experience of the latter approaches. In some HPC environments, only implicit approaches are available. In this sense, virtual memory, cache mechanism and NUMA are also classified into implicit approaches.

When an implicit approach is appropriate or the only one available, how to use it? The key point is the thorough understanding of the actual behaviour of this implicit approach, until the level of hardware layer. In case of NUMA, even if it looks like shared memory, it is actually a kind of distributed memory, with each processor having its own memory. In case of SIMD instruction, what kind of assembly code is actually generated by compiler through SIMD vectorization? And also, what kind of memory access pattern dominates the loop? Such level of understanding might be required, if you are unlucky and something is not going well.

3.6.4 Software Design and Performance Optimization

As already mentioned, through this chapter, the concept of performance-centric software design is introduced. The idea is, let's think about the performance tuning task up-front in the design phase of software, rather than mere hacking-like activity after everything finished. And also, we have mentioned just before that, it is necessary to consider some aspects of HPC hardware and software architecture in the design phase. Here in this section, some more know-hows and tips are shown for seeking the performance-centric design.

For example, portability. How to handle this issue in the code development for HPC? In the extreme case, it might be the best to write code in assembly language for each different HPC platform. In terms of portability, this is obviously the worst case. In the modern supercomputing environments, however, it seems to be unavoidable to modify the code for each specific hardware platform and its associated special software development environment. Even programming languages might have to be chosen.

Let's follow a well-known software design principle. The solution is modularization, as always. Identification of hot spot helps to isolate performance-sensitive portions of the code from the rest. Let's move them into the device-dependent part, and optimize only these portions thoroughly for each specific HPC environment. Whatever tools and programming languages, even assembly language can be used in these isolated portions. It is very similar to just maintaining portability among various kinds of OS, network and GUI environment, found usually in the scene of modern software development in other fields. Libraries such as MPI and BLAS can be considered as examples of this case. In the similar way, each application can prepare its own portability layer against platform diversity and future change of hardware.

4 Design and Implementation of Finite Element Code

The finite element method (FEM) is one of the famous approximation methods for solving partial differential equations. Starting from structural mechanics, it has mainly been applied to macro-scale problems in the continuum mechanics field.

As for the history of FEM on supercomputers, until recently, it used to be just enough to tune only the direct solver part of the FE code. With sufficiently large problem size, either band or skyline solver can be easily vectorized on vector processor, or parallelized on shared memory environment.

With the emergence of distributed memory parallel supercomputers, however, things have been drastically changed. Numerical schemes adopted in the applications running on PC cluster and MPP are mainly dominated by either static/implicit time marching schemes using iterative solver as the linear equation solver, or explicit time marching schemes. The whole analysis domain has to be domain-decomposed into multiple subdomains. The use of sparse direct solvers on such a distributed memory parallel environment has just begun recently with a relatively limited number of computational nodes.

Here in this section, related to the implementation of FE code on supercomputers, four topics are explained. Starting from EBE approaches, some issues about linear algebraic solver are described, followed by the domain decomposition method (DDM) as a parallel processing scheme. Finally, the issue of pre- and post-processing in supercomputing applications is briefly described.

4.1 Element-by-Element

Although the typical performance bottleneck, or the hot spot, of a FE code is its linear algebraic solver, the part of forming element-wise matrices and vectors can also be another weak point. In case of an explicit code, this part typically dominates. Even in an implicit or a static code, it can occupy a significant portion of execution time, if non-linearity is strong and stress integration involves relatively heavy calculation, or some of the terms in the weak form are explicitly evaluated. The performance tuning effort of EBE operations is at least not negligible.

As for the performance optimization of the EBE routines, it is easy to parallelize them. Because they are namely EBE operations, each of them can be done independently. The granularity of parallelization can be any, because there will be millions, or perhaps billions of elements to be processed.

Care should be taken for the assembly part, however. To assemble element-wise matrices into a global matrix, and element-wise vectors into a global vector, conflict occurs. Sorting of elements is required to avoid this conflict. In more general, this assembly process corresponds to so-called scatter operation, from elements to nodes. It distributes element data onto the corresponding nodes. The reverse one is "gather" operation, from nodes to elements. It collects nodal data for each element.

While the gather operation is read only operation from a nodal vector, the scatter operation updates a nodal vector. It involves both read and write. When multiple elements try to write their own data into the same nodal vector simultaneously, it easily causes data update conflict. In case of shared memory environment, this conflict must be suppressed by element reordering. Instead, if it is performed to domain-decomposed data structure in distributed memory parallel environment, data exchange occurs along the interface boundary between adjacent subdomains.

In addition to parallelization, what else? Intra-core optimization remains. For example, SIMD vectorization. Assuming that there is no major IF/THEN branch in forming an element-wise value, evaluation of this quantity in multiple elements or multiple integration points in an element can be performed not only in parallel, but also in exactly the same way. This fact naturally leads to SIMD. SIMD/short vectorization can be applied. SIMD implementation on GPU is also possible if a sufficient number of elements, for example, thousands or more, can be allocated in each GPU.

4.2 Linear Algebraic Solver

Unless the scheme of your FE code is purely explicit, it is necessary to prepare a linear algebraic solver to handle the matrix equation $[A]\ \{x\} = \{b\}$. The matrix $[A]$ may be represented as band, skyline or sparse. Here, sparse means storing only non-zero components in a certain way, such as compressed row storage (CRS) format or block CRS one.

If no iterative solver can be employed to solve the linear equation, because the matrix [A] is highly ill-conditioned or some other reasons, direct solver is the only choice. In this case, however, you'd better give up using a supercomputer. Even a small-scale PC cluster may have difficulty in scaling, unless the problem size is really huge. Currently, it can be said that it is very difficult or even impossible for direct solver to take advantage of distributed memory parallel architecture. As for a typical modern supercomputer with hundreds or thousands of computational nodes, either implicit code employing iterative solver or explicit code using EBE operations predominantly is the choice.

It can be said that iterative solver works fine on massively parallel environment, if domain decomposition is properly performed [2]. At least, the matrix–vector product, which is the main body of iterative solver, can be easily parallelized on any parallel environment, from SIMD, share memory to MPP. However, this is the case without pre-conditioner. In some problems, without efficient pre-conditioner, the convergence ratio is very bad. Then, the question is, are there any parallel pre-conditioners available? Currently, not so many pre-conditioners are ready. It is because, the stronger and the more effective the pre-conditioner is, the more it looks like a kind of direct solver. As we explained just before, it is very difficult to parallelize direct solver. Currently, finding a good parallel pre-conditioner is on-going challenge in this research field. Some parallel pre-conditioners based on

hierarchical domain decomposition, in a similar way as that of sparse direct solver, are available. Also, multi-grid approaches are effective. Either geometrical multi-grid or algebraic multi-grid (AMG) solver may be used. The third way is, DDM, described in the next section.

Although direct solver alone is almost useless on massively parallel environment, it is still useful for two cases. One is, for a component of much more complicated numerical software stack. The other is, on a desktop workstation with accelerator like GPU or many-core. Basically, direct solver works very well with cache mechanism. It is quite different from iterative solver, which is usually memory bound. Especially, the factorization phase of the direct solver can easily achieve near the peak performance on a single computational node. It also works well on GPU and many-core accelerators. Therefore, it can be utilized as a sub-component inside more sophisticated and complicated solver. Direct solver can be used, for example, as the linear algebraic solver for the coarsest grid in multi-grid solver, as the solver of the local models in non-linear homogenization code, and as the local subdomain and the coarse grid solvers in DDM.

4.3 Domain Decomposition Method

The DDM is a way to solve partial differential equation, by solving each subdomain problem separately [3]. The solution of each subdomain problem requires assumed boundary condition. With the repetition of this process, this, initially assumed boundary condition along subdomain interface gradually converges to the true one. In this sense, it is a kind of iterative method. DDM is inherently parallel because each subdomain local problem can be solved independently [4, 5]. To solve the subdomain local problem, either iterative or direct solver may be used.

Anyway, DDM is different from just decomposing a mesh into multiple sub-domains. This type of domain decomposition reduces the communication cost in matrix–vector product operations, which is typically found in parallel iterative solver and explicit code. Instead, DDM is an independent numerical scheme originated from the mathematical field of partial differential equation. Sometimes, it is also called as Shur complement method or iterative sub-structuring method in the field of iterative solver.

DDM is a kind of hybrid method between iterative solver and direct one. The global view of DDM is iterative, and pre-conditioner may be required in the similar way as the case of iterative solver. Coarse grid solver is typically employed as pre-conditioner. In this case, the coarse grid is derived from the graph structure of domain decomposition itself, and it can be said to be a kind of two grid version in multi-grid solver. To solve the coarse problem and the subdomain local problems, direct solver may also be utilized.

4.4 Pre- and Post-Processing: Where and How?

When a supercomputer is employed to solve a huge-scale problem, its input and output data, namely, the mesh and the result data of such a huge-scale analysis can also be very huge. Then, the pre- and post-processing becomes a critical issue.

Somehow, you need to perform these tedious tasks, but where? Perhaps, on the supercomputer? No way! ... Sorry, it is not a joke. There are two, more specific questions about the issues to handle such a huge-scale analysis. The first one is, where and how is the mesh generated? And the other one is, can we get back such huge result data from the supercomputer onto our desktop workstation, through campus LAN and the Internet?

For the first question, the mesh generator has to be ported onto the supercomputer side. That means, it also has to be fully parallelized. Unfortunately, the development of highly parallel automatic mesh generator is still the on-going research topic. To overcome the issue, instead, mesh refiner may be utilized [6]. This tool simply refines the given initial mesh into half. Speaking more exactly, in case of 3-D solid/volume element, each element is divided into eight elements, by adding a mid-node on each edge. One step refinement increases the number of elements 8 times. Uniform mesh refinement itself can be easily parallelized. Because the refiner can be invoked on the supercomputer, it is sufficient to create an initial coarse mesh on your desktop PC and send it to the supercomputer through slow network.

For the second question about the visualization of the result data, however, things are more complicated. One obvious answer is, generate images and movie data on the supercomputer side [7]. Not only this post-processor simply runs on the supercomputer, it should also be parallelized. Assuming the analysis domain is already decomposed into subdomains, it is possible for each subdomain to generate its own result image independently, and to gather these images and compose a single image, by image convolution techniques. These processes can be parallelized. This software rendering process can be contained as a part of the main analysis process. After a job is completed, you can obtain not only the result data, but also their images and movies. While waiting for the download of the huge result data, you can quickly browse these images and movies.

Well, you may think that this solution is insufficient. We agree to you. Interactivity is missing. To do so, however, there are some challenges. First, how to bring back such a huge data set? Some kinds of data compression techniques are needed. In case of structural analysis, it might be sufficient to extract only the surface portion from the volume data. Part-by-part visualization may be used also. Then, the next issue is, how to handle such a huge data set on a single desktop workstation? Another PC cluster, dedicated for visualization purpose, may be needed. The third question is about the complexity itself of the huge result data. For example, the geometry data of the model also tends to be highly complicated. Just rotating and zooming does not work. Fly-through or walk-through visualization, typically found in the virtual reality field, may be helpful.

References

1. Dongarra, J.J., et al.: Numerical Linear Algebra for High-Performance Computers. SIAM, Philadelphia (1998)
2. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia (2003)
3. Smith, B., et al.: Domain Decomposition: Parallel Multilevel Methods for Elliptical Partial Differential Equations. Cambridge University Press, Cambridge (2004)
4. Bhardwaj, M., et al.: Salinas: A scalable software for high-performance structural and solid mechanics simulations. In: Proceedings of SC02 (2002)
5. Ogino, M., Shioya, R., Kawai, H., Yoshimura, S.: Seismic response analysis of full scale nuclear vessel model with ADVENTURE system on the earth simulator. J. Earth Simul. **2**, 41–54 (2005)
6. Murotani, K., Sugimoto, S., Kawai, H., Yoshimura, S.: Hierarchical domain decomposition with parallel mesh refinement for billions-of-DOF-scale finite element analyses. Int. J. Comput. Methods (2013). doi:10.1142/S0219876213500618
7. Kawai, H., Ogino, M., Shioya, R., Yoshimura, S.: Vectorization of polygon rendering for off-line visualization of a large scale structural analysis with ADVENTURE system on the earth simulator. J. Earth Simul. **9**, 51–63 (2008)