Towards Code Selection by Auto-tuning with ppOpen-AT

An Example of Computation Kernels from FDM

Takahiro Katagiri
Supercomputing Research Division
Information Technology Center,
The University of Tokyo / JST CREST
2-11-16 Yayoi, Bunkyo-ku,
Tokyo 113-8658, JAPAN
+81-3-5841-2743
katagiri@cc.u-tokyo.ac.jp

Masaharu Matsumoto
Supercomputing Research Division
Information Technology Center,
The University of Tokyo / JST CREST
2-11-16 Yayoi, Bunkyo-ku,
Tokyo 113-8658, JAPAN
+81-3-5841-2743
matsumoto@cc.u-tokyo.ac.jp

Satoshi Ohshima
Supercomputing Research Division
Information Technology Center,
The University of Tokyo / JST CREST
2-11-16 Yayoi, Bunkyo-ku,
Tokyo 113-8658, JAPAN
+81-3-5841-2743
ohshima@cc.u-tokyo.ac.jp

ABSTRACT

In this paper, we propose an effective kernel implementation for an application of the finite difference method (FDM) by merging computations of central-difference and explicit time expansion schemes without IF statements inside the loops. The effectiveness of the implementation depends on the CPU architecture and execution situation, such as the problem size and the number of MPI processes and OpenMP threads. We adopt auto-tuning (AT) technology to select the best implementation. The AT function for the selection, referred to as "code selection", is implemented in an AT language, namely, ppOpen-AT. The results of experiments conducted using current advanced CPUs (Xeon Phi, Ivy Bridge, and FX10) indicated that crucial speedups of conventional AT are achieved by code selection. In particular, the heaviest kernels achieved speedups of 4.21x (Xeon Phi), 2.52x (Ivy Bridge), and 2.03x (FX10).

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Software libraries; D.3.4 [**Processors**]: Code generation; G.1.8 [**Partial Differential Equations**]: Finite difference methods

General Terms

Design, Performance, Standardization

Keywords

Auto-tuning, ppOpen-AT, Code Selection, Hybrid MPI/OpenMP, Many-core Architectures

1. INTRODUCTION

Recent CPU architectures are too complex for computational scientists to perform code optimization. Auto-tuning (AT) is a promising technology for addressing performance issues in various computer environments without code modification. Historically, AT has been adapted for processes in numerical libraries, because they can be well defined in terms of their functions, such as Basic Linear Algebra Subprograms (BLAS). Hence, many numerical libraries with AT have been proposed [1]–[8]. However, their target processes remain limited, because they can only be applied to dense linear algebra, signal processing, computation kernels of sparse matrix operations, and several numerical parameters in algorithms.

Currently, multi-core CPUs are witnessing widespread use. The number of parallel operations, such as executions with processes and threads, is now of the order of several hundreds for physical cores. Hyper-threading (HT) technology enables us to nearly double the number of parallel operations that the cores can handle. Nevertheless, it is becoming increasingly difficult to use parallelism, e.g., how to optimize combinations of hybrid MPI/OpenMP execution.

1.1 A Motivating Example

Fig. 1 shows the result of a hybrid MPI/OpenMP execution with a simulation program based on a finite difference method (FDM). The execution is performed using 8 Xeon Phi nodes with HT; hence, we can achieve parallelism of up to 240 threads per node. In Fig. 1, "P8T240" denotes 8 MPI processes and 240 thread executions per MPI process. The following points can be inferred from Fig. 1:

1. Significant variation in hybrid MPI/OpenMP execution performance according to the combinations.
2. Effectiveness of the code variants and possibility of variation in the best execution.

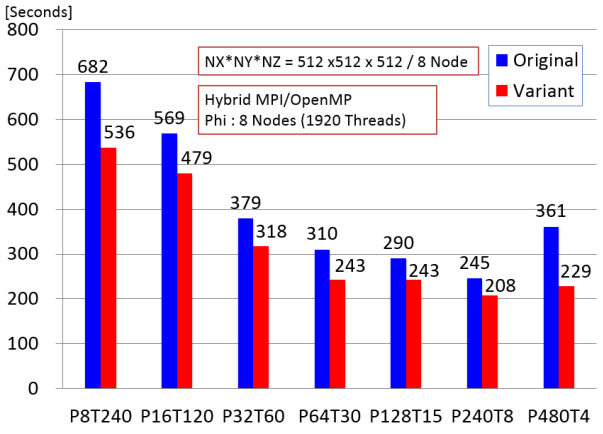


Figure 1 Execution time of hybrid MPI/OpenMP in Xeon Phi (up to 240 threads per node).

With regard to the first point, combinations of MPI/OpenMP execution affect the whole performance. In this case, the gain reaches 2.78x, which is the case for P8T240 (682 s) and P240T8 (245 s) for the original code. This means that the selection of hybrid MPI/OpenMP is a crucial factor for current many-core CPUs. The code in Fig. 1 is not optimized; hence, we need to tune the original code with several variants in each hybrid execution of MPI/OpenMP. This means that AT is desirable in each execution of hybrid MPI/OpenMP to reduce the production cost of high-performance software.

With regard to the second point, performance is improved by using code variants, which are completely different implementations as compared to the original code. In this case, the gain reaches 1.57x (P480T4). Moreover, the best execution of hybrid MPI/OpenMP may be varied by tuning the performance with the code variants. Thus, the merits of using AT cannot be ignored.

We can summarize the above discussion by asserting that (1) automatic tuning of code modification and (2) selection of code variants (hereafter, simply "code selection") are important technological factors affecting the development of high-performance and low-cost numerical software.

1.2 Related Work

Historically, code selection is not a new technology from the viewpoint of compiler optimization. For example, code selection by object code has been proposed in [9]. Selection of compiler options, which can be regarded as code selection based on the optimization ability of the compiler itself, has been proposed in [10][11]. However, mathematical or algorithmic selection of codes from different implementations is beyond the scope of compiler optimization.

Several AT frameworks that utilize runtime information have been proposed [12]–[14]. Although they can adapt code selection, they do not focus on code selection. The typical targets of such software include tunable parameters, especially system parameters, e.g., page sizes of virtual memory.

Another AT approach is to use critical run-time information, such as loop length, to perform loop transformation, e.g., loop split and loop fusion. Most AT techniques have been proposed under this category [15]–[23]. However, such frameworks do not focus on code selection, except for ABCLibScript [17], which has been investigated previously by one of the authors. In particular, code selection with variants based on different algorithms has not been considered. On the other hand, Brewer proposed a framework of high-level optimization [24] that can perform code selection from among different variants based on different algorithms. However, loop transformations and simple adaptations using AT languages, such as directive-based specifications, are beyond the scope of his study.

In this paper, we focus on an AT framework for adapting (1) code selection via a dedicated AT language and (2) loop transformation as a function of AT to ease the achievement of the targets.

1.3 Features of the AT Facility

The AT facility used in our study provides the following features.

- **Totally static code generation**

All conventional AT frameworks use a dynamic code generator for AT candidates. For example, ATLAS [2] generates code candidates in the first phase. Then, in the second phase, the code is compiled and evaluated. After evaluation of code quality, if it is not satisfactory, code generation is performed and the framework returns to the first phase. This is a typical process of dynamic code generation for AT. We refer to this AT framework as *dynamic code generation AT* (*DCG-AT*).

The framework used by ppOpen-AT [25]-[27] is different from DCG-AT; it is based on the FIBER framework [28]. In FIBER, an auto-tuner simultaneously generates all AT candidate codes before the release time, and all of them include the original code. We refer to this AT framework as *static code generation AT* (*SCG-AT*). One of the drawbacks of SCG-AT is the increased code size due to the inclusion of the original code; however, we verified that such an increase is not significant in the case of a real simulation code [25]-[27]. Hence, SCG-AT is feasible for real applications, because the targets of AT are limited in actual use by specifying the developer's knowledge. Moreover, SCG-AT can adapt very easily to supercomputer systems from the viewpoint of batch job systems and it does not require a software stack [27], e.g., script languages.

- **Easy-to-adapt code selection**

ppOpen-AT supplies a function for code selection via a directive. Loop transformations can also be specified via a directive. Code selection and loop transformation can be adapted simultaneously for arbitrary parts of programs. This is the main advantage of ppOpen-AT.

1.4 Originality

This paper makes the following contributions:

1. It proposes an effective implementation of FDM by merging computations of central-difference and explicit time expansion schemes without IF statements within loops. This facilitates high performance by compiler optimization, especially for current multi-core and many-core CPUs.
2. It provides an AT function for code selection to utilize the code discussed above. Adaptation of loop transformations with respect to code selection is another original contribution of this paper.

The remainder of this paper is organized as follows. Section 2 describes our proposed implementation and AT framework. Section 3 discusses the target application and illustrates the adaptation of ppOpen-AT. Section 4 presents our performance evaluations. Finally, Section 5 summarizes our findings and concludes the paper with a brief discussion on the scope for future work.

2. ppOpen-AT

2.1 Overview

ppOpen-AT is designed to apply AT functions to numerical libraries with a minimal software stack. The software framework of ppOpen-AT is based on ABCLibScript [17] and the FIBER framework [28].

ppOpen-AT was originally designed to utilize basic functions from ABCLibScript [17]. However, a number of enhancements were made for loop transformations [25]-[29]: (1) loop split with data dependencies in statements in loops, (2) loop collapse, (3) re-ordering of statements in loops, and (4) combinations of loop transformations in (1)–(3).

2.2 Before Execute-time AT and Characteristics of Target Application

In the FIBER framework, three types of AT time can be specified: (1) install-time, (2) before execute-time, and (3) run-time. In this paper, we focus on before execute-time AT (BET-AT) [28]. In FDM, the user specifies a dedicated problem size to perform large-scale computation. Then, software based on the size of the problem is run thousands of times. In this situation, BET-AT is the best approach for using run-time information, such as loop length.

Given the BET-AT time, the auto-tuner is called after the users of the library have set the problem size and system parameters, such as number of OpenMP threads and MPI processes. Hence, the performance parameter search space can be reduced on the basis of the BET-AT time.

3. ADOPTION OF PPOPEN-AT

3.1 Seism3D

3.1.1 Overview

Our target application is Seism3D, an earthquake wave analysis program [30]. The program is constructed and released as ppOpen-APPL/FDM, one of the software packages in the ppOpen-HPC project. Because the code of ppOpen-APPL/FDM is based on a 3D simulation method, 3D arrays are allocated. The data type used is single precision.

The governing equations of Seism3D are equations of motion:

$$\rho \ddot{u}_p = \frac{\partial \sigma_{xp}}{\partial x} + \frac{\partial \sigma_{yp}}{\partial y} + \frac{\partial \sigma_{zp}}{\partial z} + f_p \quad (p = x, y, z) \tag{1}$$

where σ_{pq} is the stress tensor, ρ is the density, f_p is the body force, and $\ddot{u}_p$ is the particle acceleration. Here, σ_{pq} represents the stresses in an isotropic elastic medium and is defined as

$$\sigma_{pq} = \lambda \left(\frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z} \right) \delta_{pq} + \mu \left(\frac{\partial u_p}{\partial q} + \frac{\partial u_q}{\partial p} \right) \quad (p, q = x, y, z) \tag{2}$$

where λ and μ are Lamé constants and δ is the Kronecker delta [30].

In Seism3D, the velocities $\dot{u}_p$ at the next time step are updated by the following equation, obtained by time integration with respect to Eq. (1):

$$\frac{\partial \dot{u}_p}{\partial t} = \frac{1}{\rho} \left(\frac{\partial \sigma_{xp}}{\partial x} + \frac{\partial \sigma_{yp}}{\partial y} + \frac{\partial \sigma_{zp}}{\partial z} \right) + \frac{f_p}{\rho} \tag{3}$$

On the other hand, time differentiation of Eq. (2) yields the following equation:

$$\frac{\partial \sigma_{pq}}{\partial t} = \lambda \left(\frac{\partial \dot{u}_x}{\partial x} + \frac{\partial \dot{u}_y}{\partial y} + \frac{\partial \dot{u}_z}{\partial z} \right) \delta_{pq} + \mu \left(\frac{\partial \dot{u}_p}{\partial q} + \frac{\partial \dot{u}_q}{\partial p} \right) \quad (p, q = x, y, z) \tag{4}$$

To solve Eq. (1), the velocities in Eq. (3) that correspond to the kernel **update_vel** and the stresses in Eq. (4) that correspond to the kernel **update_stress** are alternately calculated using the leap-frog scheme in Seism3D. These physical quantities are defined in a 3D computational domain with a staggered grid and are discretized by a fourth-order accurate central-difference scheme.

3.1.2 Process Flow

Fig. 3 shows the process flow for solving Eq. (1) via FDM.

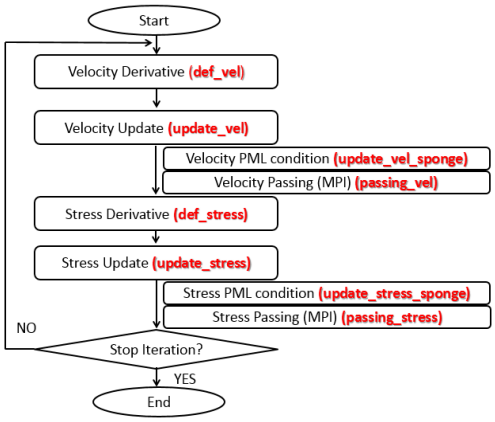


Figure 2 Processes and kernels in Seism3D (ppOpen-APPL/FDM).

Owing to 3D FDM, all processes form three nested loops in Fig. 2. MPI is used to send boundary data (Halo) to neighbor processes. Seism3D uses a 3D staggered grid; hence, the neighbor processes take a maximum of six directions. Hence, a maximum of six MPI_Isend messages are issued. In the next phase, four MPI_Irecv messages are issued. In the final phase, MPI_Waitall is issued to confirm the sending of the messages. Because there is no global explicit synchronization in the loop in Fig. 2, the delay of each kernel is propagated to the next iteration.

3.2 Target Kernels

The heavy kernels are **def_stress** and **update_stress**. They account for around 40% or more of the total execution time [31]. In this section, we treat these two kernels. Fig. 3 shows the kernel **update_stress**.

```
!$omp parallel do private (k, j, i, RL1, RM1, RM2, RLRM2, DXVX1, DYVY1, …)
<1>do k = NZ00, NZ01
<2>  do j = NY00, NY01
<3>    do i = NX00, NX01
<4>      RM1 = RIG (I,J,K)
<5>      RM2 = RM1 + RM1; RL1 = LAM (I,J,K)  RLRM2 = RL1+RM2;
<6>      DXVX1 = DXVX(I,J,K); DYVY1 = DYVY(I,J,K); DZVZ1 = DZVZ(I,J,K)
<7>      D3V3  = DXVX1 + DYVY1 + DZVZ1
<8>      SXX (I,J,K) = SXX (I,J,K) + (RLRM2*(D3V3)-RM2*(DZVZ1+DYVY1) ) * DT
<9>      SYY (I,J,K) = SYY (I,J,K) + (RLRM2*(D3V3)-RM2*(DXVX1+DZVZ1) ) * DT
<10>     SZZ (I,J,K) = SZZ (I,J,K) + (RLRM2*(D3V3)-RM2*(DXVX1+DYVY1) ) * DT
<11>     DXVYDYVX1 = DXVY(I,J,K)+DYVX(I,J,K)
<12>     DXVZDZVX1 = DXVZ(I,J,K)+DZVX(I,J,K)
<13>     DYVZDZVY1 = DYVZ(I,J,K)+DZVY(I,J,K)
<14>     SXY (I,J,K) = SXY (I,J,K) + RM1 * DXVYDYVX1 * DT
<15>     SXZ (I,J,K) = SXZ (I,J,K) + RM1 * DXVZDZVX1 * DT
<16>     SYZ (I,J,K) = SYZ (I,J,K) + RM1 * DYVZDZVY1 * DT
<17>   end do
<18> end do
<19>end do
!$omp end parallel do
```

Figure 3 Form of kernel *update_stress*.

3.3 Auto-tuning Description

3.3.1 Description of Loop Split and Loop Collapse

To evaluate the loop split and the loop collapse functions, we specify the ppOpen-AT directives [27].

First, we need to declare the AT region. The AT region is specified using "**!OAT$ static LoopFusionSplit region start**" and "**!OAT$ static LoopFusionSplit region end**" before the start line for the target kernels. To perform loop split, we specify "**!OAT$ SplitPoint (k, j, i)**" after the line of the target position. To establish this split, we need a copy of the statements in some cases. We can also specify this copy with "**!OAT$ SplitPointCopyDef sub region start**" and "**!OAT$ SplitPointCopyDef sub region end**" before the target statement. If we need to copy statements to the split loop, we can specify this with "**!OAT$ SplitPointCopyInsert**" after the line "**!OAT$ SplitPoint (k, j, i).**"

3.3.2 Description of Code Selection

In ppOpen-AT, the code selection function is supplied. The AT region is specified using "**!OAT$ static select region start**" and "**!OAT$ static select region end**" before the start line for the target kernels. The target codes need to be specified with "**!OAT$ select sub region start**" and "**!OAT$ select sub region end**" before the target statement. The hierarchical AT (e.g., the target code calls a procedure named *foo()* and the AT region exists inside *foo()*) can be specified.

3.4 Code Variants

3.4.1 Conventional Implementation

The target architecture of Seism3D is a vector machine, such as the NEC SX series (the earth simulator) in Japan. Owing to the nature of vector machines, the code is written such that it can accept a high byte per FLOP (B/F) ratio to achieve high performance. Fig. 4 shows an overview of the original implementation for the kernels **def_stress** and **update_stress**.

```
! Fourth-order accurate central-difference scheme for
velocity. (def_stress)
<1> call ppohFDM_pdiffx3_m4( VX,DXVX, NXP,NYP,NZP,NXP0,NXP1,NYP0,…)
<2> call ppohFDM_pdiffy3_p4( VX,DYVX, NXP,NYP,NZP,NXP0,NXP1,NYP0,…)
<3> call ppohFDM_pdiffz3_p4( VX,DZVX, NXP,NYP,NZP,NXP0,NXP1,NYP0,…)
<4>…
! Process of model boundary.
<5> if ( is_fs .or. is_nearfs ) then
<6>  call ppohFDM_bc_vel_deriv( KFSZ,NIFS,NJFS,IFSX,IFSY,IFSZ,JFSX,JFSY,JFSZ )
<7> end if
! Explicit time expansion by leap-frog scheme.
(update_stress, see Fig. 3.)
<8>  call ppohFDM_update_stress(1, NXP, 1, NYP, 1, NZP)
```

Figure 4 Original implementation of kernels *def_stress* and *update_stress*.

According to Fig. 4, many subroutines are called, especially the kernel **def_stress**. In addition, the calculated velocity in **def_stress** cannot be reused in **update_stress**. This includes the B/F ratio for **update_stress**. The estimated B/F ratio for **update_stress** is 1.7 [31], which is not suitable for current CPUs; the tendency of B/F ratios by hardware is around 0.5 in current CPUs.

3.4.2 Code Variant for Scalar Machines

The B/F ratio of **update_stress** can be reduced by merging the processes in **def_stress** (<1>-<4> in Fig. 4) with **update_stress** (<8> in Fig. 4). This implementation has been proposed in [31].

There are several optimization techniques for implementing the merged code. For example, loop collapse of the outer three loops (see lines <1>-<3> in Fig. 3) is one of the candidates for maximizing loop length to increase parallelism. This involves thread parallelism by OpenMP [25]. Another approach is to split loops to maximize register allocations by the compiler [27], e.g., splitting of loops in line <11> in Fig. 3. In this paper, we focus on loop collapse of the outer two loops and loop split in line <11> in Fig. 3 for the k, j, and i loops of the code variants. This is because we could maximize the performance of Xeon Phi in a previous study [31][1].

Based on the above discussions, an overview of the code is shown in Fig. 5. In this figure, the following processes are included within the loop in **update_stress**: (1) fourth-order accurate central-difference scheme for velocity, (2) process of model boundary, and (3) explicit time expansion by leap-frog scheme. The B/F ratio is reduced to 0.4 in Fig. 5 [31]. Hence, the code in Fig. 5 tends to fit current scalar CPUs, including many-core CPUs.

One of the drawbacks of the code shown in Fig. 5 lies in the IF statements, such as line <9>. The IF statements prevent compiler optimization, e.g., prefetching of data. Hence, performance may be reduced to that of the original code depending on the CPU.

3.4.1 Code Variant for Scalar Machine without IF Statements (Proposed Implementation)

Following from the discussion in Section 3.4.2, we need to discard the IF statements to achieve higher performance of the code shown in Fig. 5. By reordering the processes in Fig. 5, we can obtain a code without IF statements for the main computations. Fig. 6 shows the code. **This implementation has an originality in terms of implementation of Seism3D.**

In Fig. 6, we introduce two types of loops: (1) loops for main computations without IF statements (lines <1>-<12> in Fig. 6), and (2) loop for process of model boundary with IF statements (lines <13>-<37> in Fig. 6). By using the first type, we can achieve speedup of the code shown in Fig. 5.

[1] There is scope to optimize the loops with another loop transformation by AT.

```
!$omp parallel do private (i,j,k,RL1,RM1,RM2,RLRM2,DXVX, ···)
<1> do k_j=1, (NZ01-NZ00+1)*(NY01-NY00+1)
<2>  k=(k_j-1)/(NY01-NY00+1)+NZ00;  j=mod((k_j-1),(NY01-NY00+1))+NY00
<3>    do i = NX00, NX01
<4>     RL1   = LAM (I,J,K);  RM1   = RIG (I,J,K)
<5>     RM2   = RM1 + RM1; RLRM2 = RL1+RM2
! Fourth-order accurate central-difference scheme for velocity.
<6>   DXVX0 = (VX(I,J,K)  -VX(I-1,J,K))*C40/dx - (VX(I+1,J,K)-VX(I-2,J,K))*C41/dx
<7>   DYVY0 = (VY(I,J,K)  -VY(I,J-1,K))*C40/dy  - (VY(I,J+1,K)-VY(I,J-2,K))*C41/dy
<8>   DZVZ0 = (VZ(I,J,K)  -VZ(I,J,K-1))*C40/dz - (VZ(I,J,K+1)-VZ(I,J,K-2))*C41/dz
! Process of model boundary.
! X direction
<9>   if (idx==0) then
<10>    if (i==1)then
<11>      DXVX0 = ( VX(1,J,K) - 0.0_PN    )/ DX
<12>    end if
<13>    if (i==2) then
<14>      DXVX0 = ( VX(2,J,K) - VX(1,J,K) )/ DX
<15>  end if;  end if
<16>  if ( idx == IP-1 ) then
<17>    if (i==NXP)then
<18>      DXVX0 = ( VX(NXP,J,K) - VX(NXP-1,J,K) ) /  DX
<19>   end if; end if
! Y direction
<20>  … <As same as processes in X direction>
 ! Z direction
<21>  …  <As same as processes in X direction>
! Explicit time expansion by leap-frog scheme.
<22>    DXVX1 = DXVX0;  DYVY1 = DYVY0
<23>    DZVZ1 = DZVZ0;  D3V3  = DXVX1 + DYVY1 + DZVZ1
<24>  SXX (I,J,K) = SXX (I,J,K) + (RLRM2*(D3V3)-RM2*(DZVZ1+DYVY1) ) * DT
<25>  SYY (I,J,K) = SYY (I,J,K) + (RLRM2*(D3V3)-RM2*(DXVX1+DZVZ1) ) * DT
<26>  SZZ (I,J,K) = SZZ (I,J,K)  + (RLRM2*(D3V3)-RM2*(DXVX1+DYVY1) ) * DT
<27>  end do
<28> end do
!$omp end parallel do
```

Figure 5 Merged implementation of kernels *def_stress* and *update_stress*. Calculation of the elements $\sigma_{xx}, \sigma_{yy}, \sigma_{zz}$ for the stress tensor is shown.

```
! < Loops for main computations >
!$omp parallel do private(i,j,k,RL1,RM1,···)
<1> do k_j=1, (NZ01-NZ00+1)*(NY01-NY00+1)
<2>  k=(k_j-1)/(NY01-NY00+1)+NZ00;  j=mod((k_j-1),(NY01-NY00+1))+NY00
<3>  do i = NX00, NX01
<4>    RL1=LAM (I,J,K); RM1=RIG (I,J,K); RM2=RM1 + RM1; RLRM2=RL1+RM2
! Fourth-order accurate central-difference scheme for velocity.
<5>   DXVX0 = (VX(I,J,K)  -VX(I-1,J,K))*C40/dx - (VX(I+1,J,K)-VX(I-2,J,K))*C41/dx
<6>   …
! Explicit time expansion by leap-frog scheme.
<7>   DXVX1=DXVX0; DYVY1=DYVY0;
<8>   DZVZ1=DZVZ0; D3V3=DXVX1+DYVY1+DZVZ1;
<9>   SXX (I,J,K) = SXX (I,J,K) + (RLRM2*(D3V3)-RM2* (DZVZ1+DYVY1) ) * DT
<10>  …
<11> end do
<12>end do
!$omp end parallel do
! < Loop for process of model boundary >
<13> if( is_fs .or. is_nearfs ) then
!$omp parallel do private(i,j,k,RL1,RM1,・・・)
<14> do i=NX00,NX01
<15>   do j=NY00,NY01
<16>     do k = KFSZ(i,j)-1, KFSZ(i,j)+1, 2
<17>       RL1=LAM (I,J,K); RM1=RIG (I,J,K); RM2=RM1+RM1; RLRM2=RL1+RM2
! Fourth-order accurate central-difference scheme for velocity.
<18>       DXVX0=(VX(I,J,K)-VX(I-1,J,K))*C40/dx-(VX(I+1,J,K)-VX(I-2,J,K))*C41/dx
<19>       …
! Process of model boundary.
<20>       if (K==KFSZ(I,J)+1) then
<21>         DZVX0 = ( VX(I,J,KFSZ(I,J)+2)-VX(I,J,KFSZ(I,J)+1) )/ DZ
<22>         DZVY0 = ( VY(I,J,KFSZ(I,J)+2)-VY(I,J,KFSZ(I,J)+1) )/ DZ
<23>       else if (K==KFSZ(I,J)-1) then
<24>         DZVX0 = ( VX(I,J,KFSZ(I,J)  )-VX(I,J,KFSZ(I,J)-1) )/ DZ
<25>         DZVY0 = ( VY(I,J,KFSZ(I,J)  )-VY(I,J,KFSZ(I,J)-1) )/ DZ
<26>       end if
<27>         DXVX1 = DXVX0; DYVY1 = DYVY0; DZVZ1 = DZVZ0
<28>         D3V3=DXVX1+DYVY1+DZVZ1; DXVYDYVX1=DXVY0+DYVX0
<29>         DXVZDZVX1 = DXVZ0+DZVX0; DYVZDZVY1 = DYVZ0+DZVY0
<30>         if (K==KFSZ(I,J)+1) then
<31>           KK=2
<32>         else
<33>           KK=1
<34>         end if
! Explicit time expansion by leap-frog scheme.
<35>       SXX (I,J,K)=SSXX (I,J,KK)+(RLRM2*(D3V3)-RM2*(DZVZ1+DYVY1))*DT
<36>       …
<37> end do;  end do;  end do
!$omp end parallel do
```

Figure 6 IF-free implementation of kernels *def_stress* and *update_stress*. Calculation of the elements $\sigma_{xx}, \sigma_{yy}, \sigma_{zz}$ for the stress tensor is shown.

3.5 Code Selection with ppOpen-AT

To implement the code variants shown in Figs. 5 and 6 for the target program, we use the *select* directive of ppOpen-AT. Fig. 7 shows the implementation of the *select* directive.

```
<1>!OAT$ static select region start
<2>!OAT$ name ppohFDMupdate_stress_select
<3>   !OAT$ select sub region start
<4>call ppohFDM_pdiffx3_m4( VX,DXVX,NXP,NYP,NZP,…)
<5>call ppohFDM_pdiffy3_p4( VX,DYVX, NXP,NYP,NZP,…)
<6>call ppohFDM_pdiffz3_p4( VX,DZVX, NXP,NYP,NZP,…)
<7>   …
<8> if ( is_fs .or. is_nearfs ) then
<9>    call ppohFDM_bc_vel_deriv ( KFSZ,NIFS,NJFS,IFSX,… )
<10>end if
<11>call ppohFDM_update_stress (1, NXP, 1, NYP, 1, NZP)
<12>  !OAT$ select sub region end
<13>  !OAT$ select sub region start
<14> call ppohFDM_update_stress_Scalar
          ( 1, NXP, 1, NYP, 1, NZP )
<15>  !OAT$ select sub region end
<16>  !OAT$ select sub region start
<17> call ppohFDM_update_stress_IF_free
        ( 1, NXP, 1, NYP, 1, NZP)
<18>  !OAT$ select sub region end
<19>!OAT$ static select region end
```

Figure 7 Code selection with *select* directive of ppOpen-AT. The red frames are the targets of code selection.

In Fig. 7, the code variants are implemented with subroutine calls. The subroutine **ppohFDM_update_stress_Scalar** in line <14> calls the code in Fig. 5, while the subroutine **ppohFDM_update_stress_IF_free** in line <17> calls the code in Fig. 6. As can be seen, it is very easy to implement code selection by using the *select* directive, because it involves only specifications with the directives "**!OAT$ select sub region start**" ~ "**!OAT$ select sub region end**" for the target codes. The best sub region is selected based on execution time for default setting of the *select* directive in timing of BET-AT.

3.6 AT Time

For the AT time in this program, we propose the use of BET-AT in the FIBER framework [28]. The BET-AT time is user-specified. Hence, in this time, the problem size as well as the number of MPI processes and OpenMP threads are determined and utilized for AT. In ppOpenAT, we can specify the BET-AT by the *static* directive (see line <1> in Fig. 7.)

Fig. 8 shows a flowchart of utilization for automatically generated code by ppOpen-AT in BET-AT. After the user fixes the size of the problem as well as the number of MPI processes and OpenMP threads, AT is executed with "OAT_EXEC=1" for the environmental variable. After AT, we do not need to specify AT anymore, unless the problem size and the number of MPI processes and OpenMP threads need to be varied. In such cases, we need to set the environmental value with "OAT_EXEC=0." Hence, the AT time can be ignored if we run the program many times. Several numerical programs follow the scheme shown in Fig. 8, including Seism3D.

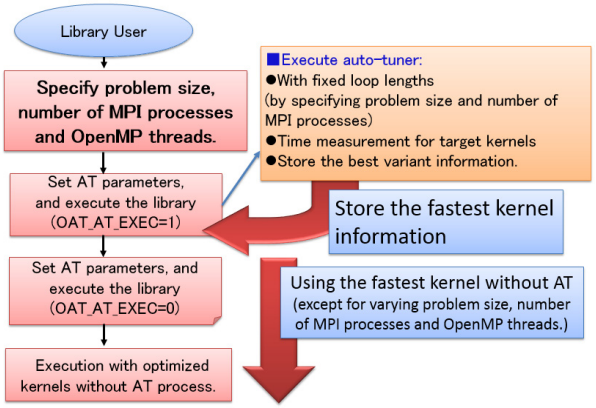


Figure 8 Flow chart of utilization for automatic generated code by ppOpen-AT in the BET-AT.

4. Performance Evaluation

4.1 Experimental Environment

4.1.1 Intel Xeon Phi

We used an Intel Xeon Phi co-processor system. The CPU is the Xeon Phi 5110P (60 cores, 1.053 GHz, 8 GB memory, 1 TFLOPS theoretical peak performance) connected to one board on each node of the cluster. We used InfiniBand FDR × 2 ports with Mellanox Connect-IB as the network adapter of each board. Further, we used PCI-E Gen3 × 16 and 56 Gbps × 2 to obtain a theoretical peak bandwidth of 13.6 GB/s. Full-bisection connection was used. We also used Intel MPI version 5.0 Update 3 (Build 20150128), with native mode execution only. For the compiler, we used Intel Fortran version 15.0.3 (20150407), with compiler options "-ipo20 -O3 -warn all -openmp -mcmodel=medium -shared-intel -mmiccl -align array64byte." NUMA affinity was set to KMP_AFFINITY = granularity = fine, balanced. The number of hyper-threads was 4.

4.1.2 Intel Xeon (Ivy Bridge)

The CPU of Intel Ivy Bridge is the Intel Xeon E5-2670 V2 (2.50 GHz) with 10 cores per socket, with two sockets implemented in the same package. Hence, 20 threads per node is the maximum number of threads for Ivy Bridge. The number of hyper-threads in this experiment was 1. For the compiler, we used Intel Fortran version 15.0.3 (20150407), with compiler options "-ipo20 -O3 -warn all -openmp -mcmodel=medium -shared-intel." NUMA affinity was set to KMP_AFFINITY = granularity = fine, compact. The number of hyper-threads was 1.

4.1.3 The FX10

The CPU of the Fujitsu PRIMEHPC FX10 (FX10), which is installed at the Information Technology Center, The University of Tokyo, is the SPARC64 IX-fx (1.848 GHz) with 16 cores. Hence, 16 threads per node is the maximum number of threads for FX10. For the compiler, we used Fujitsu Fortran Version 1.2.1, with compiler options "-Kfast,openmp."

4.2 Benchmark Condition

4.2.1 Problem Sizes

We used ppOpen-APPL/FDM ver. 0.2.0 in this experiment. The target region was specified with NX × NY × NZ = 512 × 512 × 512/8 Nodes. We set 2000 iterations as the benchmark to

measure the main loop. The measured time of the kernels is the mean time of all the MPI processes.

4.2.2 Hybrid MPI/OpenMP Execution

To evaluate hybrid MPI/OpenMP execution, we investigated the performance for several types of executions. ppOpen-APPL/FDM has a lower limit for hybrid MPI/OpenMP execution: at least 8 MPI processes have to be utilized.

We denote the hybrid MPI/OpenMP execution as P***X***T***Y***, where ***X*** represents the number of MPI processes and ***Y*** represents the number of threads per MPI process.

The possible combinations with 8 nodes and 240 threads per node in the Xeon Phi are {P8T240 | P16T120 | P32T60 | P64T30 | P128T15 | P240T8 | P480T4}. Theoretically, it is possible to execute P960T2 and P1920T1 (pure MPI). However, an MPI error arose in our environment. Consequently, we removed those combinations.

In Ivy Bridge and FX10, the possible combinations are {P8T20 | P16T10 | P32T5 | P40T4 | P80T2 | P160T1 (pure MPI)} and {P8T16 | P16T8 | P32T4 | P64T2 | P128T1 (pure MPI)}, respectively.

4.2.3 AT Condition

As explained earlier, we used BET-AT in FIBER [28]. Before the main iteration, all AT candidates were executed and selected with the best implementation. This means that we used a brute-force search for the AT candidates. Our experiments involved 49 kernels for the conventional implementation (Fig. 4) and 2 for the new variants (Figs. 5 and 6.) For the conventional implementation, we adapted several loop transformations, such as loop collapse and loop split for AT. The details are published in [27]. In summary, loop transformation for the original code and code selection for the new code variants were adapted in our AT.

The iteration count for measuring the kernels was set to 100. In the following experiment, the AT execution time was not included, but the time for obtaining the best parameter via the tuning database was included. This means that the I/O time for the database was included in the execution time.

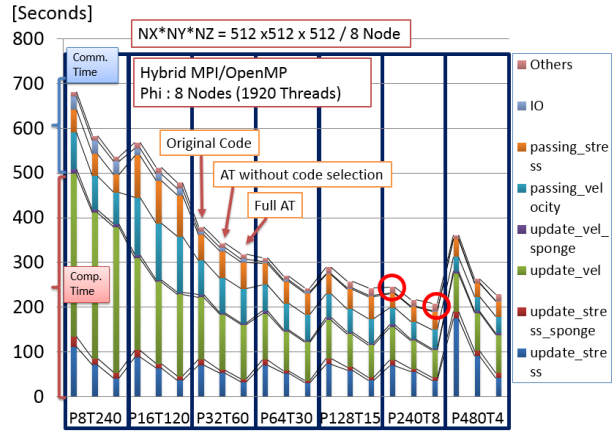
Our experimental results showed that the AT execution time was less than 20% of the total execution time for 2000 iterations. As mentioned earlier, the AT time can be ignored if we run the AT-tuned program many times.

4.3 Effect of AT on Selection of Computation Kernels

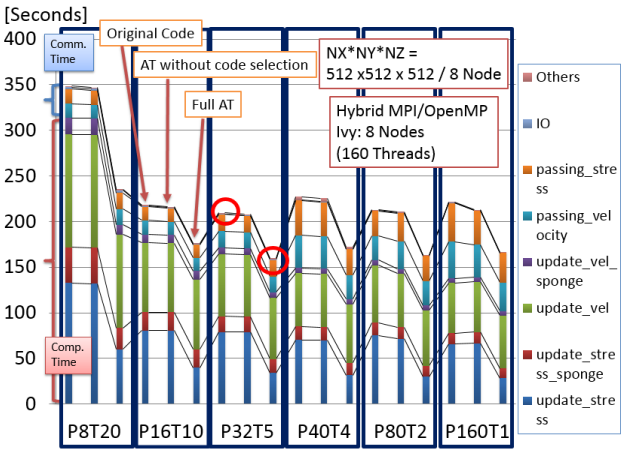
4.3.1 Entire Execution Time

Fig. 9 shows the effect of AT on the entire execution time.

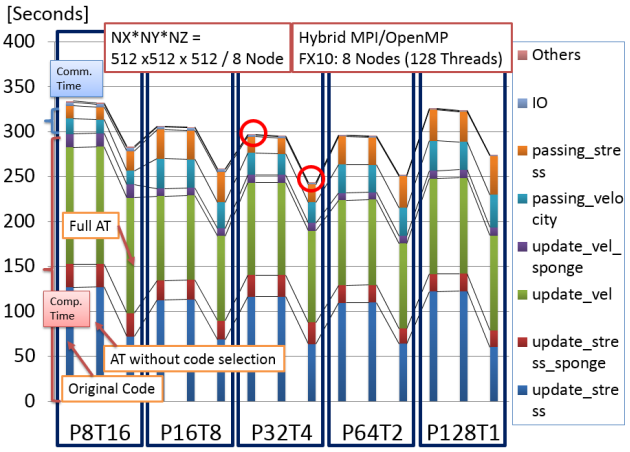
In this figure, the leftmost bars in each execution of hybrid MPI/OpenMP show the original code execution without AT. The middle bars show the conventional AT result with respect to the original code by adapting loop collapses, loop splits, and reordering of statements [27]. The rightmost bars show the proposed AT, referred to as “Full AT” in the figure, which is adapted for code selection. The results in Fig. 9 indicate that it is crucial to adapt code selection for all CPUs. In other aspects, we do not find any change in the best execution for hybrid MPI/OpenMP in this experiment.



(a) Xeon Phi



(b) Ivy Bridge

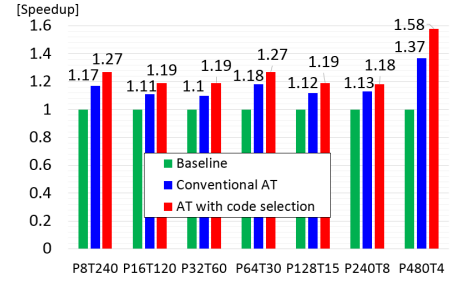


(c) FX10

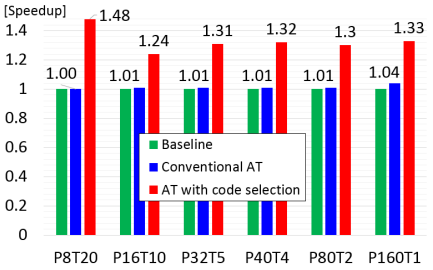
Figure 9 Effect of AT on entire execution time (2000 time steps). The red circles denote the fastest execution of the original code and AT with code selection.

On the other hand, Xeon Phi is considerably affected by changing the execution of hybrid MPI/OpenMP, while the effect on the other CPUs is not significant. One of the reasons for this is the poor performance of OpenMP with a large number of threads (see execution with 240 threads per node in Xeon Phi).

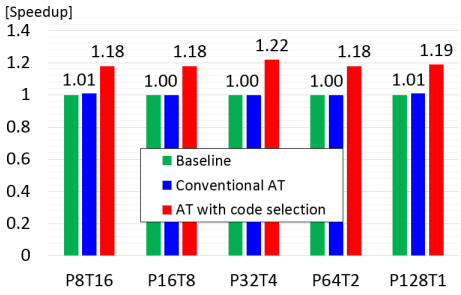
Fig. 10 shows the speedup factors of the entire execution time.



(a) Xeon Phi



(b) Ivy Bridge



(c) FX10

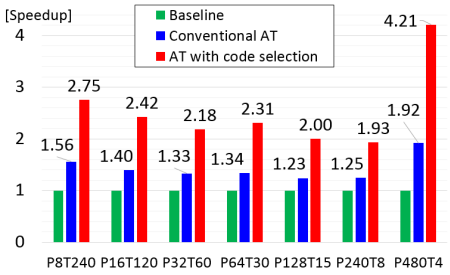
Figure 10 Speedup factors of the entire execution time. The factors are normalized by the execution time of the original code (referred to as "Baseline") with 2000 time steps.

According to Fig. 10, crucial speedups are achieved by code selection. The maximum factors are 1.58x (Xeon Phi), 1.48x (Ivy Bridge), and 1.22x (FX10). Except for Xeon Phi, the speedup factors are almost constant. The speedup factors increase with the number of MPI processes in Xeon Phi because the computational efficiency of **update_stress** is somehow poor in the baseline case (see P480T4 in Fig. 9(a)). One of reasons is that the length of the outer loop in each MPI process decreases; hence, it is difficult to adapt compiler optimization, such as prefetching, in the case of the original code. This is because the B/F ratio of the original code is high, e.g., B/F = 1.7. With code selection, we can adopt different implementations with low B/F ratios to achieve optimization using computers.

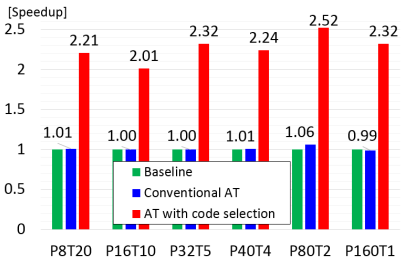
4.3.2 def_stress and update_stress

Fig. 11 shows the aggregated execution time of **def_stress** and **update_stress** with 2000 time steps.

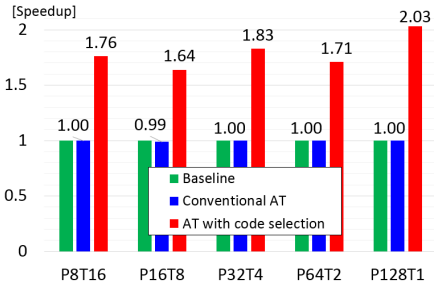
According to Fig. 11, crucial speedups are achieved by adapting code selection. The maximum speedup factors are larger than those of the entire execution, i.e., 4.21x (Xeon Phi), 2.52x (Ivy Bridge), and 2.03x (FX10).



(a) Xeon Phi



(b) Ivy Bridge



(c) FX10

Figure 11 Speedup factors of *def_stress* and *update_stress* normalized by the time of the baseline. This is based on the aggregated time with 2000 time steps for *def_stress* and *update_stress*.

4.4 Parameter Change

In next step, we investigate how the best parameter changes. Fig. 12 shows the best parameter among the three CPU architectures. In the figure, the x-axis represents MPI parallelism. For example, MPI parallelism of "1" signifies P8T240 for Xeon Phi, P8T20 for Ivy Bridge, and P8T16 for FX10; MPI parallelism of "2" signifies P16T128 for Xeon Phi; and so on.

Interestingly, in Fig. 12, the best implementation is changed according to the MPI parallelism in Xeon Phi, while the other CPUs are selected with the IF-free kernel (Fig. 6). Next, we analyze the execution speed of the kernels for the scalar machine and the IF-free case. Fig. 13 shows the execution speed of the two kernels for 100 iterations, which can be obtained in the BET-AT phase.

According to Fig. 13, the best kernel is changed in P64T30 to utilize the IF-free kernel. The maximum speedup of the kernel for the scalar machine is 1.24x (P8T240). One of the reasons for this phenomenon is the change in the loop lengths. If we use a small number of MPI processes, the loop length is greater than that in the case of a large number of MPI processes. In this situation, loop optimization can be easily achieved even if there are IF-statements inside the loop, e.g., by prefetching of data with long distance.

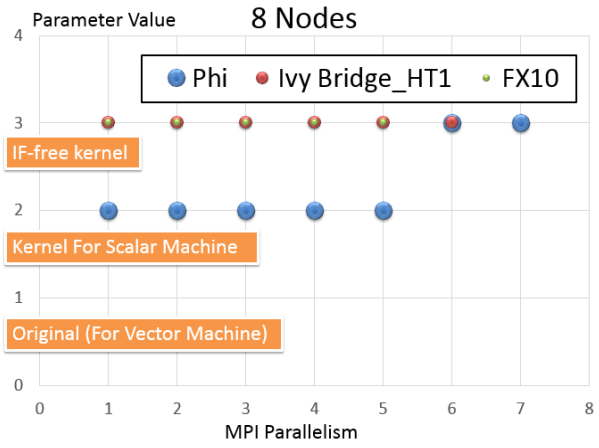


Figure 12 Parameter change of *update_stress*.

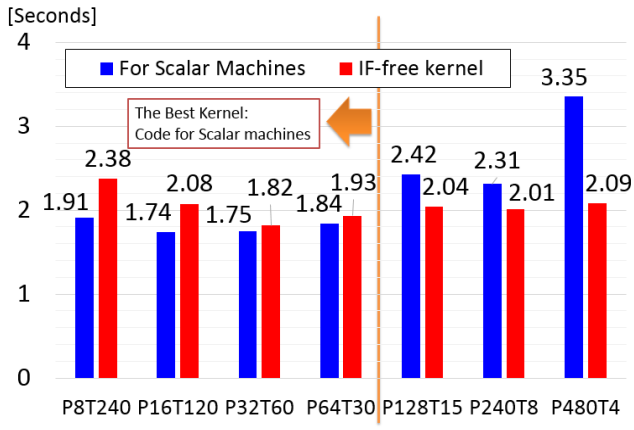


Figure 13 Change in the best code for *update_stress* for Xeon Phi. Execution of 100 iterations of the kernel is measured in the BET-AT phase.

Although the exact reason or performance model of the execution is not clear in Fig. 13, computational scientists are only interested in utilizing the faster kernel. To meet this requirement, AT can provide a solution. The results of Fig. 13 represent the case in which we shall provide the AT time of BET-AT to application users. In addition, the results serve as a highly motivating example to adapt AT to real applications.

5. CONCLUSION

In this paper, we first proposed an effective kernel implementation of an application with FDM by merging computations of central-difference and explicit time expansion schemes. The rationale was to discard IF statements, which are usually located inside loops to implement model boundary processes for FDM applications.

Second, the effectiveness of the new implementation depends on the CPU architecture and execution situation, such as problem size and the number of MPI processes and OpenMP threads. To address this issue, we adopted auto-tuning (AT) technology to select the best implementation. AT of code selection is easily implemented by using ppOpenAT.

The results of current advanced CPUs, such as Xeon Phi, Ivy Bridge, and FX10, indicated that crucial speedups can be achieved with AT by code selection. In particular, the heaviest kernels, referred to as **def_stress** and **update_stress**, were highly accelerated. The speedup factors reached 4.21x (Xeon Phi), 2.52x (Ivy Bridge), and 2.03x (FX10). Moreover, we identified the case of changing the best implementation with code selection according to the execution style of hybrid MPI/OpenMP. This is the example that motivated us to adapt AT.

In the future, we plan to apply our framework to other computer environments, such as GPU and off-loading with Xeon Phi. ppOpen-AT is freeware, the codes for which can be obtained from the ppOpen-HPC homepage [32][33].

6. ACKNOWLEDGMENTS

This study was supported by the "ppOpen-HPC: Open Source Infrastructure for Development and Execution of Large-Scale Scientific Applications on Post-Peta-Scale Supercomputers with Automatic Tuning (AT)" program of Basic Research Programs: CREST, Development of System Software Technologies for post-Peta Scale High Performance Computing, Japan Science and Technology Agency (JST), Japan. We would like to thank all members of the ppOpen-HPC project, especially Professor Kengo Nakajima from the University of Tokyo, for supporting our study. We would also like to thank Professor Takashi Furumura from the University of Tokyo and Dr. Futoshi Mori from Iwate Medical University for providing us with ppOpen-APPL/FDM. Further, we would like to thank Associate Professor Toshihiro Hanawa from the University of Tokyo for his suggestion of using a computer environment with Xeon Phi coprocessors.

7. REFERENCES

[1] M. Frigo, S.G. Johnson, "FFTW: An adaptive software architecture for the FFT," in Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 3, IEEE Press, Los Alamitos, CA, pp. 1381–1384, 1998.

[2] R.C. Whaley, A. Petitet, J.J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," Parallel Computing, Vol. 27, Issue 1–2, pp. 3–35, 2001.

[3] H.Kuroda, T. Katagiri, M. Kudoh, Y. Kanada, ILIB_GMRES: An auto-tuning parallel iterative solver for linear equations, Proceedings of SC2001, 2001.(A Poster)

[4] T. Katagiri, K. Kise, H. Honda, and T. Yuba, "Effect of auto-tuning with user's knowledge for numerical software," Proceedings of ACM Computing Frontiers 04, pp. 12–25, 2004.

[5] E-J. Im, K. Yelick, R. Vuduc, "SPARSITY: Optimization framework for sparse matrix kernels," International Journal of High Performance Computing Applications (IJHPCA), Vol. 18, No. 1, pp. 135–158, 2004.

[6] R. Vuduc, J.W. Demmel, K.A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," In Proceedings of SciDAC, Journal of Physics: Conference Series, Vol. 16, pp. 521–530, 2005.

[7] T. Katagiri, K. Kise, H. Honda, T. Yuba, "ABCLib_DRSSED: A parallel eigensolver with an auto-tuning facility," Parallel Computing, Vol. 32, Issue 3, pp. 231–250, 2006.

[8] T. Sakurai, T. Katagiri, H. Kuroda, K. Naono, M. Igai, S. Ohshima, "A sparse matrix library with automatic selection

of iterative solvers and preconditioners," Proceedings of iWAPT2013 (In conjunction workshop with International Conference on Computational Science, ICCS2013),Vol.18, pp. 1332–1341, 2013.

[9] J. W. Davidson, C.W. Franser, "Code selection through object code optimization," ACM Transactions on Programming Languages and Systems, Vol. 6, No. 4, pp. 505–526, 1984.

[10] M. Haneda, P.M.W.Knijnenburg, H.A.G. Wijshoff, "Generating new general compiler optimization settings," Proceedings of ACM International Conference on Supercomputing (ICS05), pp.161–168 ,2005.

[11] S.-C. Lin, C.-K. Chang, "Automatic selection of GCC optimization options using a gene weighted genetic algorithm", Proc. of IEEE Computer Systems Architecture Conference 2008, pp.1–8, 2008.

[12] R.L. Ribler, J.S. Vetter, H. Simitci, D.A. Reed, "Autopilot: Adaptive control of distributed applications," In HPDC '98: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing, Washington, DC, USA, pp. 172, 1998.

[13] C. Cascaval, E. Duesterwald, P.F. Sweeney, R.W. Wisniewski, "Multiple page size modeling and optimization," Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT 2005), pp. 339–349, 2005.

[14] A. Tiwari, J.K. Hollingsworth, "Online adaptive code generation and tuning," Parallel and Distributed Processing Symposium (IPDPS), pp. 879–892, 2011.

[15] J. Xiong, J. Johnson, R. Johnson, D. Padua, "SPL: A language and compiler for DSP algorithms," Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01), pp. 298–308, 2001.

[16] M.J. Voss, R. Eigenmann, "High-level adaptive program optimization with ADAPT," ACM SIGPLAN Notices 2001, pp. 93–102.

[17] T. Katagiri, K. Kise, H. Honda, T. Yuba, "ABCLibScript: A directive to support specification of an auto-tuning facility for numerical software," Parallel Computing, Vol. 32, Issue 1, pp. 92–112, 2006.

[18] Q. Yi, K. Seymour, H. You, R. Vuduc, D. Quinlan, "POET: Parameterized optimizations for empirical tuning," Proceedings of Parallel and Distributed Processing Symposium (IPDPS 2007), pp. 1–8, 2007.

[19] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M.J. Garzar´an, D. Padua, K. Pingali, "A language for the compact representation of multiple program versions," Proceedings of Languages and Compilers for Parallel Computers (LCPC'05), Lecture Notes in Computer Science 4339, pp. 136–151, 2007.

[20] C. Chen, J. Chame, M. Hall, J. Shin, G. Rudy, "Loop transformation recipes for code generation and auto-tuning," Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing, 2009.

[21] S. Ramalingam, M. Hall, C. Chen, "Improving high-performance sparse libraries using compiler-assisted specialization: A PETSc case study," Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), pp. 487–496, 2012.

[22] C. Schaefer, V. Pankratius, W.F. Tichy, "Atune-IL: An instrumentation language for auto-tuning parallel applications," Proceedings of the 15th International Euro-Par Conference on Parallel Processing (Euro-Par09), pp. 9–20, 2009.

[23] S. Benkner, S. Pllana, J.L. Traff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, V. Osipov, "PEPPHER: Efficient and productive usage of hybrid computing systems," IEEE Micro Vol. 31, No. 5 pp. 28–41, 2011.

[24] E. A. Brewer, "High-level optimization via automated statistical modeling," Proc. of ACM PPOPP95, pp.80-91, 1995.

[25] T. Katagiri, S. Ito, S. Ohshima, "Early experiences for adaptation of auto-tuning by ppOpen-AT to an explicit method," Special Session: Auto-Tuning for Multicore and GPU (ATMG) (In Conjunction with the IEEE MCSoC-13), Proceedings of MCSoC-13, pp. 153158, 2013.

[26] T. Katagiri, S. Ohshima, M. Matsumoto, "Auto-tuning of computation kernels from an FDM Code with ppOpen-AT," Special Session: Auto-Tuning for Multicore and GPU (ATMG) (In Conjunction with the IEEE MCSoC-14), Proceedings of MCSoC-14, pp. 92–98, 2014.

[27] T. Katagiri, S. Ohshima, M. Matsumoto, "Directive-based auto-tuning for the finite difference method on the Xeon Phi," Proceedings of IPDPSW2015, pp. 1221–1230, 2015.

[28] T. Katagiri, K. Kise, H. Honda, T. Yuba, "FIBER: A general framework for auto-tuning software," The Fifth International Symposium on High Performance Computing (ISHPC-V), Springer LNCS 2858, pp. 146–159, 2003.

[29] T. Tanaka, R. Otsuka, A. Fujii, T. Katagiri, T. Imamura, "Implementation of d-Spline-based incremental performance parameter estimation method with ppOpen-AT," Scientific Programming, IOS Press, Vol. 22, No. 4, pp. 299–307, 2014.

[30] T. Furumura, L. Chen, "Parallel simulation of strong ground motions during recent and historical damaging earthquakes in Tokyo, Japan," Parallel Computing, Vol. 31, pp. 149–165, 2005.

[31] F. Mori, M. Matsumoto, T. Furumura, "Performance optimization of the 3D FDM simulation of seismic wave propagation on the Intel Xeon Phi coprocessor using the ppOpen-APPL/FDM library," Proc. of High Performance Computing for Computational Science -- VECPAR 2014, Lecture Notes in Computer Science, Springer, Vol. 8969 of the series pp. 66–76, 2015.

[32] ppOpen-HPC: http://ppopenhpc.cc.u-tokyo.ac.jp/

[33] K. Nakajima, M. Satoh, T. Furumura, H. Okuda, T. Iwashita, H. Sakaguchi, T. Katagiri, M. Matsumoto, S. Ohshima, H. Jitsumoto, T. Arakawa, F. Mori, T. Kitayama, A. Ida and M. Y. Matsuo, "ppOpen-HPC: Open Source Infrastructure for Development and Execution of Large-Scale Scientific Applications on Post-Peta-Scale Supercomputers with Automatic Tuning (AT)," K. Fujisawa et al. (eds.), Optimization in the Real World, Mathematics for Industry 13, Springer, pp.15-35, DOI: 10.1007/978-4-431-55420-2_2, 2016.