Auto-tuning of Numerical Programs by Block Multi-color Ordering Code Generation and Job-Level Parallel Execution

Tatsuya Abe[1] and Mitsuhisa Sato[2,1]

[1] Advanced Institute for Computational Science, RIKEN, Hyogo, Japan
abet@riken.jp
[2] Center for Computational Sciences, University of Tsukuba, Ibaraki, Japan
msato@cs.tsukuba.ac.jp

Abstract. Multi-color ordering is a parallel ordering that allows programs to be parallelized by application to sequentially executed parts of the programs. While multi-color ordering parallelizes sequentially executed parts with data dependences and increases the number of parts executed in parallel, improved performance by multi-color ordering is sensitive to differences in the architectures and systems on which the programs are executed. This sensitivity requires us to tune the numbers of colors; i.e., modify programs for each architecture and system. In this work, we develop a code generator based on multi-color ordering and automatically tune the number of colors using a job-level parallel scripting language Xcrypt. Furthermore, we support block multi-color ordering that avoids the disadvantage of stride accesses in the original multi-color ordering, and evaluate and clarify the effectiveness of block multi-color ordering.

Keywords: Block multi-color ordering, source-to-source code generation, job-level parallel execution, scripting parallel language.

1 Introduction

Parallelization is a method applied to programs that allows us to parallelize sequentially executed parts of the program. Parallel orderings are parallelizations that extend the parts of a program executed in parallel by changing the order of computation in the program [1]. Why do the parts of a program executed in parallel increase when changing the order of computation in the program? To answer this, consider the following program as an example.

```
x(:) = 0
x(1) = 1
do i = 2, 100
   x(i) = x(i-1)
end do
```

M. Daydé, O. Marques, and K. Nakajima (Eds.): VECPAR 2012, LNCS 7851, pp. 404–419, 2013.

In the above program, `x(i)` is initially `0`, and then becomes `1` for any `i`. In this program, computation order is significant. If `x(3)` were computed in advance, `x(3)` would remain `0`, whereas `x(i)` would be `1` for all `i` except `3`. This is because `x(i)` depends on `x(i-1)`, and this data dependence obviously prevents the loop in the program from being parallelized.

In the above program, the computation order cannot be changed since the semantics of the program relies on the order. However, this is not always the case. There are some programs whose order of computation can be semantically changed. For example, recall the Gauss-Seidel method for solving systems of linear equations.

The Gauss-Seidel method is an iterative method. Roughly speaking, the point of the Gauss-Seidel method is to update values immediately, i.e., to use values at the n-th time step to calculate the n-th time step:

$$u_i^n = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}u_j^n - \sum_{j=i+1}^{n} a_{ij}u_j^{n-1})$$

where u_i^n denotes the i-th column of solution candidates at the n-th step and a_{ij} denotes the element in the i-th row and j-th column of the coefficient matrix. This calculation is different from that in the Jacobi method where values at the n-th step are not only used in the n-th step, but also in the $(n + 1)$-th step:

$$u_i^n = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}u_j^{n-1} - \sum_{j=i+1}^{n} a_{ij}u_j^{n-1}) \ .$$

While both the Gauss-Seidel and Jacobi methods are used to solve systems of linear equations, the values of u at the n-th step in these two methods are different. Nevertheless, this is not actually a problem. The aim in solving a system of linear equations is to find a solution of the system of linear equations. Both the Gauss-Seidel and Jacobi methods give us solution candidates for each system of linear equations. These solution candidates can be checked by being substituted into the system of linear equations. In this sense, differences in intermediate values along the way do not constitute a problem.

The Gauss-Seidel method contains data dependences. If we ignore these data dependences by changing the ordering, that is, we forcibly parallelize sequentially executed parts of a program, then intermediate values along the way would be different from those in the original program. Therefore, we must choose a legal ordering that leads to legal solutions.

In this paper, we propose automatic parallelization by using a parallel ordering called *multi-color ordering*. Furthermore, our method supports *block multi-color ordering* that avoids the disadvantage of the original multi-color ordering, namely, stride accesses, which result in degraded performance. Our method relies on code generation using block multi-color ordering, and job-level parallel execution. The proposed method provides programmers with an environment in which to modify and tune programs by block multi-color ordering.

It should be noted that multi-color ordering changes the algorithm, i.e., the semantics of the program. This distinguishes the work in this paper from studies on compilers.

Compilers do not change the semantics of programs although they may raise numerical errors by exchanging the order of operations in programs as a result of optimization[1]. This work is beyond the scope of compilers, and targets what compilers should not do. It is up to the programmers to apply the proposed method to their programs, and in this respect, great care should be taken.

We formalize the parallelization of block multi-color ordering, thus allowing it to be implemented by computers. We would also like to handle a large number of programs instantiated with parameters simultaneously. Using existing tools like Bash for this seems to be adequate. However, this is not the case. Given that anyone can execute a program in a parallel or distributed computational environment, higher portability is required for tools on computers. In reality, parallel and distributed computational environments often have batch job systems to manage and control the execution of jobs, and keep the execution of one job separate from that of another. These batch job systems usually require text files containing *job scripts*. In other words, when executing a large number of programs simultaneously in a parallel or distributed computational environment, we need to create a large number of job scripts. It is tedious to do this using existing tools like Bash.

In this work we propose using a scripting language domain specific to job-level parallel execution, namely Xcrypt [2]. Xcrypt absorbs differences between computational environments and is suitable for parameter sweeps since it was originally designed for executing and controlling such programs. Since we have developed a code generator for block multi-color ordering, i.e., applying block multi-color ordering to programs is semi-automatic, we can easily automate the execution of programs with block multi-color ordering in parallel and distributed computational environments using Xcrypt. In this paper we introduce the method by means of an example.

Outline. In Sect. 2 we introduce multi-color ordering in detail. In Sect. 3 we explain block multi-color ordering, which overcomes the disadvantage of multi-color ordering as described in Sect. 2. In Sect. 4 we give an overview of how to automatically tune programs to which block multi-color ordering has been applied. In Sect. 5 we explain the code generator based on block multi-color ordering. In Sect. 6 we introduce the job-level parallel scripting language Xcrypt, which is used to execute block multi-color ordered programs automatically, while in Sect. 7 we evaluate our method through experimental results. In Sect. 8 we conclude this work with reference to future work.

2 Multi-color Ordering

Multi-color ordering is a parallel ordering that parallelizes programs with data dependences by exchanging the order of computation of columns of matrices [1]. For the purpose of understanding parallelization, consider the differences in the settings of three-dimensional real space $\mathbb{R}^3$ for the following program:

[1] For example, the Intel Composer XE compiler has a compile option `-par-report` to show reports on parallelization, while version 12.1 has the option `-guide` to provide detailed hints. However, the compiler does not give any hints that change the semantics of the program.

```
integer x_num, y_num, z_num
real(8), allocatable :: u(:,:,:), f(:,:,:), dgn(:,:,:)
real(8), allocatable :: axp(:,:,:), axm(:,:,:)
real(8), allocatable :: ayp(:,:,:), aym(:,:,:)
real(8), allocatable :: azp(:,:,:), azm(:,:,:)
integer i, j, k
...
do k = 2, z_num+1
   do j = 2, y_num+1
      do i = 2, x_num+1
         u(i, j, k) = (f(i, j, k)-&
              axp(i, j, k)*u(i+1, j, k)-&
              axm(i, j, k)*u(i-1, j, k)-&
              ayp(i, j, k)*u(i, j+1, k)-&
              aym(i, j, k)*u(i, j-1, k)-&
              azp(i, j, k)*u(i, j, k+1)-&
              azm(i, j, k)*u(i, j, k-1)&
              )/dgn(i, j, k)
      end do
   end do
end do
```

Fig. 1. Sample code in Fortran 90

$$
\begin{aligned}
u^n(i,j,k) = (&f(i,j,k) \\
&- a_{x+}(i,j,k)u^{n-1}(i+1,j,k) - a_{x-}(i,j,k)u^n(i-1,j,k) \\
&- a_{y+}(i,j,k)u^{n-1}(i,j+1,k) - a_{y-}(i,j,k)u^n(i,j-1,k) \\
&- a_{z+}(i,j,k)u^{n-1}(i,j,k+1) - a_{z-}(i,j,k)u^n(i,j,k-1) \\
&) \,/\, dgn(i,j,k)
\end{aligned}
$$

where $dgn(i,j,k)$, $f(i,j,k)$, $a_{x+}(i,j,k)$, $a_{x-}(i,j,k)$, $a_{y+}(i,j,k)$, $a_{y-}(i,j,k)$, $a_{z+}(i,j,k)$, and $a_{z-}(i,j,k)$ are various functions from the three-dimensional space to the set of real numbers $\mathbb{R}$. $u^n(i,j,k)$ denotes the value of (i,j,k) at the n-th time step. The equation shows that $u^{n-1}(i+1,j,k)$, $u^{n-1}(i,j+1,k)$, $u^{n-1}(i,j,k+1)$, $u^n(i-1,j,k)$, $u^n(i,j-1,k)$, and $u^n(i,j,k-1)$ are used to obtain $u^n(i,j,k)$. This is done by iterating loops controlled by the third (z-axis), second (y-axis), and first (x-axis) arguments in order, that is, the equation is coded as in Fig. 1 where `f`, `axp`, `axm`, `ayp`, `aym`, `azp`, `azm`, and `dgn` denote f, a_{x+}, a_{x-}, a_{y+}, a_{y-}, a_{z+}, a_{z-}, and dgn, respectively. In the program, no loop can be parallelized, i.e., iterations in loops cannot be executed in parallel since each pair of iterations in the loops has data dependences.

Multi-color ordering forces such loops to be parallelized, i.e., some iterations in the loops are executed in parallel. It appears that parallelization changes the semantics of

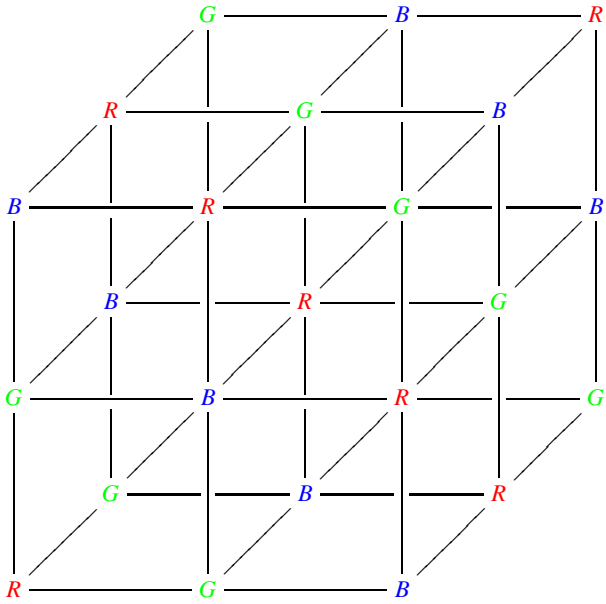


Fig. 2. Multi-color ordering of lattice points

the program. This is true. However, multi-color ordering does not force all iterations in the loops to be executed in parallel. Adjacent pairs on the three-dimensional lattice are not executed in parallel. An example with the number of colors set to 3, is illustrated in Fig. 2 where the lattice points executed in parallel are colored in the same color for simplicity. Here, *R*, *G*, and *B* are colored in red, green, and blue, respectively.

The program is one for stencil computation. Any lattice point immediately affects adjacent lattice points. Adjacent pairs should not be considered for parallel execution. Multi-color ordering executes computations at distinct lattice points. Actually, the semantics of a program to which multi-color ordering has been applied is different from that of the original program. Therefore, we cannot apply multi-color ordering to all programs and should take care in applying this ordering to programs.

Iterative methods for solving systems of linear equations are typical examples of programs to which we can apply multi-color ordering. These methods involve a sequence of initial values that is iteratively updated to a new sequence, and finally becomes a solution. Since a sequence is often proved to be a legal solution by substituting the sequence in the linear equations and analyzing the residual errors, intermediate values on the way do not matter. A change in ordering affects only the convergence rate. Therefore, we can apply multi-color ordering to such iterative methods.

We conclude this section by showing the effect of applying multi-color ordering to a program. In general, there are certain sufficient conditions that prevent adjacent lattice points from being colored in the same color. To enable us to consider these

sufficient conditions in this section, we consider an equation in one-dimensional space. The following program:

```
do i = 2, x_num+1
   u(i) = (f(i)-axp(i)*u(i+1)-axm(i)*u(i-1))/dgn(i)
end do
```

is translated to

```
mulco_color_num = 3
do mulco_color = 0, mulco_color_num-1
   do i = 2+mulco_color, x_num+1, mulco_color_num
      u(i) = (f(i)-axp(i)*u(i+1)-axm(i)*u(i-1))/dgn(i)
   end do
end do
```

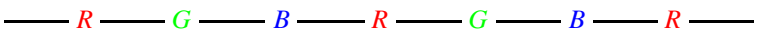
where `mulco_color_num` denotes the number of colors. Here, it is assumed to be 3. The `mulco_color` is a loop variable over $0, \ldots,$ `mulco_color_num`. When `mulco_color` is 0, computations at the red lattice points are executed in the loop. Similarly, when it is 1, computations at the green points are executed, and when `mulco_color` is 2, computations at the blue points are executed. In other words, multi-color ordering is a parallelization that executes independent computations (on non-adjacent lattice points) in parallel.

3 Block Multi-color Ordering

Multi-color ordering parallelizes sequentially executed parts with data dependences and increases the number of parts executed in parallel. However, the naïve implementation of multi-color ordering described in the previous section has the disadvantage of stride accesses, that is, jumps in accessing elements in arrays, thereby decreasing the performance of programs.

To avoid this disadvantage, block multi-color ordering, which is called *block red-black ordering* when two colors are used, has been proposed [3]. A deterioration in performance is caused by stride accesses to elements in arrays. Block multi-color ordering prevents iterations in loops from becoming too fine-grained and keeps loop iterations coarse-grained. Whereas multi-color ordering colors lattice points, block multi-color ordering colors sets of lattice points called blocks. While computations in blocks are executed sequentially, blocks are processed in parallel. When the block size is 1, block multi-color ordering is simply multi-color ordering. Conversely, the larger the grain size, the more sequential a program with block multi-color ordering becomes.

Block sizes for block multi-color ordering are customizable. The following are examples of blocks with sizes 1 and 2:



where lattice points R, G, and B are colored in red, green, and blue, respectively. Thus, block multi-color ordering decreases the number of stride accesses in programs and enables us to benefit from parallel computational environments. Extensions of the applicable scope of block multi-color ordering are therefore studied [4].

4 Toward Auto-tuning

Block multi-color ordering is one of the parallelizations that can be applied to programs. As described in Sect. 3, we can customize block sizes and the number of colors. However, block multi-color ordering requires us to modify the program source code according to the block size and number of colors. Moreover, the performance of programs to which multi-color ordering has been applied is known to be sensitive to the number of colors [5].

In this work we relieve programmers of modifying their source code in order to apply block multi-color ordering to their programs. Concretely, we develop a source code generator for programs written in Fortran 90. The code generator analyzes the Fortran 90 source code both lexically and syntactically, and returns a program with block multi-color ordering.

Furthermore, we provide programmers with an environment for automatically tuning their programs with block multi-color ordering. Since the code generator enables us to apply block multi-color ordering to programs automatically, it is sufficient to set the block size and number of colors for automatic tuning of our programs. Then, we use *job-level parallel execution*. Job-level parallel execution is the coarsest form of parallel execution, and does not require any modifications to the programs for execution in parallel. Job-level parallel execution is suitable for tuning programs with parameter sweeps. We make it possible to apply block multi-color ordering to programs automatically by using techniques for code generation and job-level parallel execution.

5 A Code Generator for Block Multi-Color Ordering

We have developed a code generator based on block MULti Color Ordering (Mulco). Mulco takes Fortran 90 source code and returns source code that includes block multi-color ordering. Mulco supports programs with computations in n-dimensional space ($n = 1, 2, 3$). Mulco also takes as parameters, the number of colors and the x-axis block size. However, this version of Mulco limits the y- and z-axes block sizes to 1. This is because it is sufficient to allow only the x-axis block size to be customizable for the purpose of removing stride accesses, which is the main disadvantage of multi-color ordering.

We reuse the sample code for the three-dimensional space given in Section 2 with the only difference being that the outermost loop has an annotation `!$bmulco(3)` as shown in Fig. 3. Mulco applies block multi-color ordering to a program when it finds `!$bmulco(`n`)` in the source code of the program, where n denotes the number of dimensions. Since n is 3 in the sample code, Mulco applies block multi-color ordering in a three-dimensional space to the program. Mulco translates the sample code into that given in Fig. 4, in which we have manually added extra carriage returns owing to

```
!$bmulco(3)
do k = 2, z_num+1
   do j = 2, y_num+1
      do i = 2, x_num+1
         u(i, j, k) = (f(i, j, k)-&
              axp(i, j, k)*u(i+1, j, k)-&
              axm(i, j, k)*u(i-1, j, k)-&
              ayp(i, j, k)*u(i, j+1, k)-&
              aym(i, j, k)*u(i, j-1, k)-&
              azp(i, j, k)*u(i, j, k+1)-&
              azm(i, j, k)*u(i, j, k-1)&
              )/dgn(i, j, k)
      end do
   end do
end do
```

Fig. 3. Sample Fortran 90 code with an annotation

the limitations on page size for the paper. Actually, Mulco adds the minimal carriage returns compatible with the Fortran 90 grammar.

Variables `mulco_color_num` and `mulco_block_size_`n ($n = 1, 2, 3$) denote the number of colors and the sizes of blocks (1: x-axis, 2: y-axis, and 3: z-axis), respectively. Although `mulco_block_size_2` and `mulco_block_size_3` give the sizes of the y- and z-axes blocks, the current version of Mulco does not support them, i.e., they are fixed at 1. The number of colors and the size of the x-axis blocks are given to Mulco as arguments. Variables `mulco_block_num_2` and`mulco_block_num_3` denote the number of y- and z-axes blocks, respectively. Since we fix the sizes of the y- and z-axes blocks to 1, the numbers of y- and z-axes blocks correspond with the numbers of y- and z-axes lattice points, respectively. Variable `mulco_block_num_1` denotes the minimum number of x-axis blocks containing all colors, i.e., the quotient of the number of x-axis blocks and the number of colors.

Variable `mulco_color` is the loop variable that represents the color using values 1 or 2, since `mulco_color_num` is 2. As mentioned in Sect. 2, when the number of colors is 2, the ordering is called a block red-black ordering. In the language of block red-black ordering, all blocks in red (or black) are processed before any blocks in black (or red, resp.).

Directives `!$omp parallel` and `!$omp do` are OpenMP directives [6]. The directive `!$omp do` dictates that iterations in the loop should be executed in parallel. Mulco generates source code in which the outermost loop (z-axis) in a space is parallelized. It should be possible to parallelize a loop of colors when the number of colors is greater than the number of threads the computer can handle. However, we have not achieved anything significant in our experimental setting discussed in Sect. 7. Therefore, we refrain from referring to this in the paper.

```
!$bmulcoed(3)
mulco_color_num = 2
mulco_block_size_3 = 1
mulco_block_size_2 = 1
mulco_block_size_1 = 16
mulco_block_num_3 = ((z_num+1-2)+1)/mulco_block_size_3
mulco_block_num_2 = ((y_num+1-2)+1)/mulco_block_size_2
mulco_block_num_1 = &
((x_num+1-2)+1)/(mulco_color_num*mulco_block_size_1)
do mulco_color = 0, mulco_color_num-1
!$omp parallel
!$omp do
do mulco_block_3 = 0, mulco_block_num_3-1
do k = 2+mulco_block_size_3*mulco_block_3, &
       2+mulco_block_size_3*(mulco_block_3+1)-1
   do mulco_block_2 = 0, mulco_block_num_2-1
      mulco_color_remainder = &
      mod((mulco_color+mulco_block_3+mulco_block_2),&
          mulco_color_num)
   do j = 2+mulco_block_size_2*mulco_block_2, &
          2+mulco_block_size_2*(mulco_block_2+1)-1
      do mulco_block_1 = 0, mulco_block_num_1-1
      do i = 2+mulco_block_size_1*(mulco_color_remainder+&
             mulco_block_1*mulco_color_num), &
             2+mulco_block_size_1*(mulco_color_remainder+&
             mulco_block_1*mulco_color_num+1)-1
         u(i, j, k) = ( f(i, j, k)-&
              axp(i, j, k) * u(i+1, j, k)-&
              axm(i, j, k) * u(i-1, j, k)-&
              ayp(i, j, k) * u(i, j+1, k)-&
              aym(i, j, k) * u(i, j-1, k)-&
              azp(i, j, k) * u(i, j, k+1)-&
              azm(i, j, k) * u(i, j, k-1) &
              )/dgn(i, j, k)
      end do
   end do
   end do
   end do
end do
end do
!$omp end do
!$omp end parallel
end do
```

Fig. 4. Translation of the Fortran 90 code based on block multi-color ordering

Variables `i`, `j`, and `k` are loop variables ranging from the initial to terminal points of the blocks. Note that the initial point of *i* is offset by `mulco_color_remainder`, the sum of `mulco_color`, `mulco_block_3`, and `mulco_block_2` modulo `mulco_color_num`. This is the point of block multi-color ordering that prevents any adjacent pair of lattice points from being colored in the same color.

As an aside, Mulco can also generate code with a multi-color ordering as given in Sect. 2 when finding the annotation `!$mulco(1)`.

6 A Scripting Language for Job-Level Parallel Execution

In this section we introduce the scripting language for job-level parallel execution, Xcrypt, developed by Hiraishi et al. [2]. In high performance computing, a program is usually executed as a job through a batch job system to prevent other jobs from interfering with that job. It is necessary to create a text file called a job script in order to submit a job to a batch job system. In addition, the format of the job script depends on the particular batch job system, making it difficult or tedious to submit, and moreover, to control a large number of jobs.

Xcrypt is a domain specific language for controlling jobs. In Xcrypt there are different layers for system administrators, module developers, and end users allowing administrators to configure Xcrypt systems, developers to provide useful modules for Xcrypt users, and end users to use Xcrypt. With this mechanism, end users can control their jobs without considering differences in systems.

Xcrypt is implemented almost as a superset[2] of the Perl programming language, with most of the new functionality implemented as functions or modules in Perl. This decreases the cost of learning a new scripting language that differs from existing scripting languages.

The example Xcrypt script shown in Fig. 5 is used to highlight the syntax of Xcrypt.

The statement `use base qw (limit core)` is a Perl statement that declares super classes similar to the notion in object oriented programming languages. Module `core` is a required module in Xcrypt, while module `limit` limits the number of jobs submitted at any one time. Xcrypt has special methods `before` and `after` that are executed before and after executing a job, respectively. Module `limit` increments and decrements a semaphore in the method `before`. These methods are used as hooks for a job, allowing the number of submitted jobs to be controlled.

Module `data_extractor` provides a way to extract data from text files. Various methods of `data_extractor` are used to obtain the elapsed time of execution later in this script.

Function `limit::initialize(1)` sets the value of the semaphore to control the number of submitted jobs.

[2] Xcrypt has certain additional reserved keywords apart from those in Perl and includes name spaces. However, Xcrypt supports the complete syntax of Perl, i.e., an interpreter for Xcrypt can execute any script written in Perl.

```
use base qw (limit core);
use data_extractor;

limit::initialize(1);

foreach $j (1..9) {
    $c = 2**$j;
    foreach $i (0..(9-$j)) {
        $x = 2**$i;
        spawn {
            system("./mulco bmulco3.f90 $c $x;" .
                   "ifort -openmp bmulco3_mulco.f90;" .
                   "time ./a.out");
        }_after_in_job_{
            my $self = shift;
            $fh0 = data_extractor->new($self->JS_stderr);
            $fh0->extract_line_rn('real');
            $fh0->extract_column_nn('end');
            my @output = $fh0->execute();
            open ($fh1, '>>', 'result.dat') or die $!;
            print $fh1 "$c $x 1 1 $output[0]\n";
        };
    }
}
sync;
```

Fig. 5. Example script in Xcrypt

Variables `c` and `x` range over the colors and the x-axis block sizes, respectively. Anything executed on computation nodes is written in a block statement `spawn`[3]. Since the statement `system` executes the string given as an argument as a command, the command is executed not at hand but on a computation node. Although a job script is required for communicating with a batch job system as previously mentioned, Xcrypt automatically creates an appropriate job script from the descriptions in the `spawn` block. Program `mulco` refers to Mulco as explained in the previous section. Mulco takes the source code written in Fortran 90, the number of colors, and the x-, y-, and z-axes block sizes as arguments. We use the Intel Composer XE compiler version 12.1 with option `-openmp` to use OpenMP. Mulco generates a Fortran 90 file with the name (*the file name of the source code*)`_mulco.f90`. Xcrypt also includes a block called _after_in_job_. Everything about extracting data in this block is asynchronously done after the main processing of the job. It is implemented by Xcrypt's special method `after` as described before. It is useful to describe this in the terminology of object oriented languages.

[3] Strictly speaking, `spawn` and the following `_after_in_job_` are not block statements, but functions in Perl. Perl and various other modern scripting languages have such mechanisms as syntax extensions.

Functions `new`, `extract_line_rn`, and `extract_column_nn` are methods in the module `data_extractor` as described. Function `sync` waits for all threads generated by `spawn` to complete.

7 Experimental Results

In this section we present experimental results for the elapsed time in executing numerical programs on one computation node[4]. The computer used in the experiments has the following specifications:

CPU:	Intel Xeon X5650 2.67 GHz
Cores:	12 (6 cores × 2)
Memory:	12 GB
OS:	CentOS 6.2
Compiler:	Intel Composer XE 12.1.2

In addition, we used `-openmp` as a compiler option as described in the previous section, since the programs are parallelized using OpenMP. The program given in Fig. 3 was used in these experiments. It makes use of nine arrays of `real(8)`. Since the memory size is 12 GB, the number of elements in a single array cannot be greater than $(12 \cdot 10^9)/(9 \cdot 2^3)$. Thus, we fix the size of a space at $512 * 512 * 512$, since $(12 \cdot 10^9)/(9 \cdot 2^3 \cdot (2^9)^3) \approx 1.24$ holds. Through the experiments we set out to investigate the relationship between the number of colors and the size of blocks. We set the number of colors to be greater than 1. Since the one-dimensional size of an array is 512, the maximum size of a block side is 256.

First, we give the results for varying numbers of colors and block sizes in Table 1. Some values in Table 1 are given as n/a owing to the limitation on the number of colors and block sizes as explained above.

Next, we investigated the effect of varying the number of colors. Figure 6 illustrates the results with the block size fixed at 1, i.e., non-block multi color ordering. As described in Sect. 4 we can see that the performance of the program is sensitive to the number of colors in block multi-color ordering, although not to the extent of being called fragile. In general, there is an optimal threshold for the number of colors for parallel execution. If the number of colors is less than the threshold, we cannot benefit from parallel environments. If the number of colors is greater than the threshold, the overhead of parallel execution contributes the greater part of the elapsed times. However, we did not obtain any results such as these. In our experiments, two colors was almost always the best. Even with a lesser number of colors, parallel execution of the inner loop may contribute to improved performance. Sensitivity to the number of colors has been mentioned with respect to multi-color ordering. In fact, this can be seen in

[4] In parallel and distributed computational environments the outermost loop in a program should be parallelized not at thread-level, but at process-level, e.g., by using certain MPI libraries and not OpenMP. This conforms to our proposal in Sect. 6 for using Xcrypt to execute programs in parallel and distributed computational environments. We assume that this is done manually. Mulco should be developed as generating a program in which the outermost loop is parallelized not by OpenMP, but by MPI.

Table 1. Elapsed times (colors versus block size)

Time (sec.)		Colors								
		2	4	8	16	32	64	128	256	512
Block size	2^0	3.516	4.304	6.505	6.716	6.716	6.920	6.105	6.107	7.119
	2^1	3.504	4.518	5.316	5.116	5.116	4.715	4.718	4.919	n/a
	2^2	3.503	4.118	4.517	4.517	4.504	4.133	4.117	n/a	n/a
	2^3	3.326	3.703	3.923	3.917	3.717	3.726	n/a	n/a	n/a
	2^4	3.317	3.704	3.514	3.512	3.306	n/a	n/a	n/a	n/a
	2^5	3.103	3.318	3.516	3.305	n/a	n/a	n/a	n/a	n/a
	2^6	3.117	3.115	3.304	n/a	n/a	n/a	n/a	n/a	n/a
	2^7	3.103	3.118	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	2^8	2.903	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	2^9	3.092	3.092	3.092	2.892	3.090	2.891	3.091	2.891	2.893
	2^{10}	3.091	2.891	2.889	2.893	2.890	2.892	2.891	2.891	n/a
	2^{11}	2.905	2.891	2.886	2.890	2.889	2.891	2.892	n/a	n/a
	2^{12}	2.896	2.891	2.893	2.892	2.891	2.894	n/a	n/a	n/a
	2^{13}	2.891	2.891	2.891	2.889	2.891	n/a	n/a	n/a	n/a
	2^{14}	2.892	2.892	2.890	2.891	n/a	n/a	n/a	n/a	n/a
	2^{15}	2.890	2.886	2.888	n/a	n/a	n/a	n/a	n/a	n/a
	2^{16}	2.891	2.893	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	2^{17}	2.891	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	2^{18}	2.891	2.890	2.891	2.891	2.910	2.893	3.291	3.692	4.490
	2^{19}	2.891	2.890	2.890	2.893	3.093	3.092	3.701	4.291	n/a
	2^{20}	2.890	2.892	2.894	3.090	3.093	3.691	4.292	n/a	n/a
	2^{21}	2.891	2.892	3.091	3.091	3.692	4.292	n/a	n/a	n/a
	2^{22}	2.891	2.888	3.291	3.691	4.274	n/a	n/a	n/a	n/a
	2^{23}	2.888	3.290	3.692	4.292	n/a	n/a	n/a	n/a	n/a
	2^{24}	3.093	3.489	4.293	n/a	n/a	n/a	n/a	n/a	n/a
	2^{25}	3.491	4.292	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	2^{26}	4.292	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

the histogram in Table 1 with only a single color. However, it seems to be unnecessary to adjust the number of colors when the block size is large. We have not been able to obtain any explicit results thus far.

Finally, we fixed the number of colors at 2, and obtained results for varying block sizes as shown in Fig. 7. The best elapsed time is less than those for block size 1. That is, the bar graph in Fig. 7 confirms that block multi-color ordering clearly contributes to better performance of this program than non-block multi-color ordering. The best time is found when the block size is 2^{23}. Since the number of lattice points of an object is 2^{27}, the number of iterations is $2^{27}/2^{23} = 16$. Since the number 16 is the upper bound of more than the number of computational cores 12, it can be considered to be the best to coarse-grainedness, an essence of block multi color ordering.

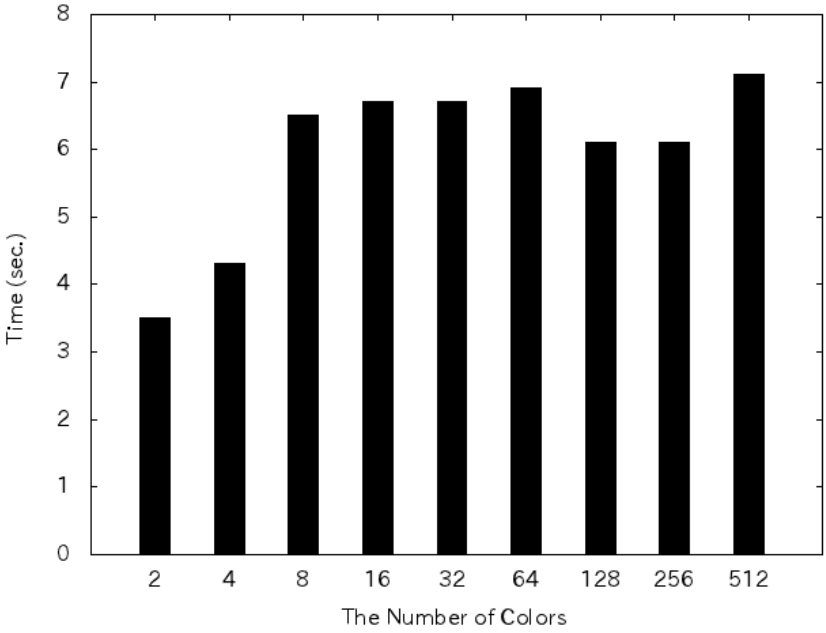


Fig. 6. Elapsed times (colors)

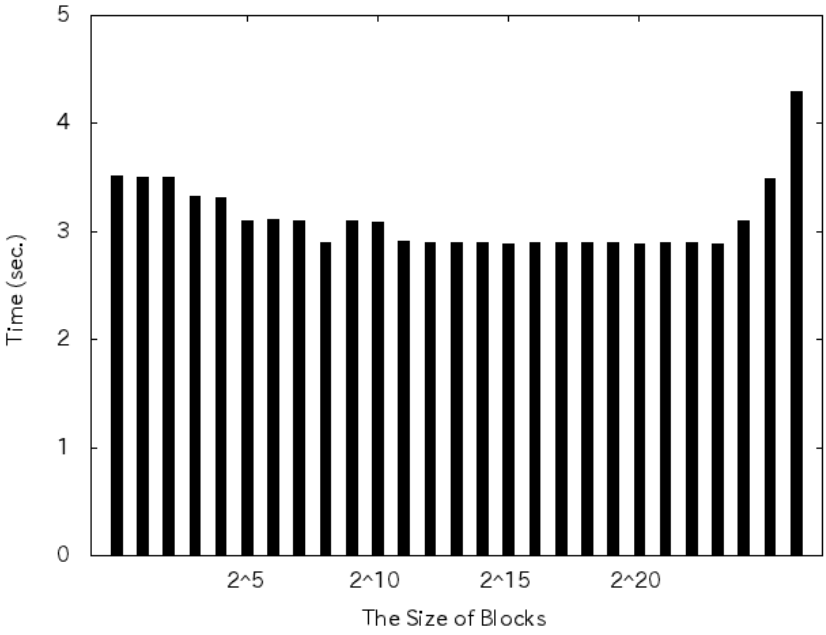


Fig. 7. Elapsed times (block sizes)

8 Conclusion, Related Work, and Future Work

In this paper we proposed an auto-tuning method incorporating code generation and job-level parallel execution. The resulting code generator is based on block multi-color ordering and we use a domain specific language, Xcrypt, for job-level parallel execution, making it easy to generate parameters and jobs from programs, and to control the jobs. Experimental results for varying numbers of colors and blocks sizes were presented.

As a means for implementing our method, we developed the code generator Mulco. Using the directive `!$bmulco(`n`)` in the source code of a Fortran program instructs Mulco to generate Fortran code. Thus, Mulco requires a small modification to the original Fortran code and exploits existing resources. This is the main difference between the proposed method and other methods, which require significant modifications to the original program, or programs to be written from scratch, when parallelizing existing programs.

ROSE is a well-known auto-tuning method that uses source-to-source program transformation [7]. ROSE provides a framework for development in which programmers can execute their methods including optimizations and parallelizations. While ROSE provides such a platform, we provide two tools that can be embedded in any program flow. Our auto-tuning method consists of a combination of source-to-source code generation and job-level parallel execution. Programmers can use our tools as components in their program flow when and wherever they choose. Hitherto we have mostly used the so-called Unix tools as typified by GNU tools. We are developing our tools with specific focus on parallel and distributed computational environments.

The current version of Mulco does not support changes in the sizes of y- and z-axis blocks, that is, Mulco in this version cannot divide a research object into blocks of the shape except cuboid[5]. A support for any y- and z-axis blocks remains a future work. From the results of the experiments in this work we found that larger block sizes are better; however, we have not yet found the maximum block size that gives optimum performance. This is also left to a future work.

Acknowledgments. The authors wish to thank Takeshi Iwashita and Masatoshi Kawai for their helpful comments throughout this work. The authors would also like to thank the anonymous reviewers for their suggestions and comments to the submitted draft.

References

1. Duff, I.S., Meurant, G.A.: The effect of ordering on preconditioned conjugate gradient. BIT Numerical Mathematics 29(4), 635–657 (1989)
2. Hiraishi, T., Abe, T., Miyake, Y., Iwashita, T., Nakashima, H.: Xcrypt: Flexible and intuitive job-parallel script language. In: The 8th Symposium on Advanced Computing Systems and Infrastructures, pp. 183–191 (2010) (in Japanese)
3. Iwashita, T., Shimasaki, M.: Block red-black ordering: A new ordering strategy for parallelization of ICCG method. International Journal of Parallel Programming 31(1), 55–75 (2003)

[5] This is a suggestion by Takeshi Iwashita at the workshop.

4. Kawai, M., Iwashita, T., Nakashima, H.: Parallel multigrid Poisson solver based on block red-black ordering. In: High Performance Computing Symposium, Information Processing Society of Japan, pp. 107–116 (2012) (in Japanese)
5. Doi, S., Washio, T.: Ordering strategies and related techniques to overcome the trade-off between parallelism and convergence in incomplete factorizations. Parallel Computing 25(13-14), 1995–2014 (1999)
6. Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Holzmann, G.J.: Parallel Programming in OpenMP. Morgan Kaufmann (2000)
7. Quinlan, D.J.: Rose: Compiler support for object-oriented frameworks. Parallel Processing Letters 10(2), 215–226 (2000)