Methods of Parallel Experimental Design of Online Automatic Tuning and their Application to Parallel Sparse Matrix Data Structure

Reiji Suda[1]

the University of Tokyo / JST, CREST

Abstract. Automatic tuning is one of key technologies in high performance computing, where parallel processing is essential. In this paper, we propose some methods of parallel experimental design for online automatic tuning of parallel programs. In parallel processing, two kinds of tuning should be investigated. One is *local tuning*, which optimizes local tuning parameters on each processor, and the other is *global tuning*, that affects executions of all processors. This paper deals with local tuning, and proposes three methods. The first method we consider is SEO (Serial Experiments Once), where each processor once measures the performance of all the candidates. The second method is PEO (Parallel Experiments Once), where the candidates are evaluated in a distributed manner over the parallel processors. Each processor observes the performance of a subset of the candidates that are assigned to it. Then the performance results are collected, and the candidate of the best performance is chosen. The third method is MPEO (Modified PEO), where each processor can replace the candidate chosen by PEO if it performs badly. Pros and cons of those methods are discussed, and results of application of those methods to an online optimization of parallel sparse matrix data structure are shown.

1 Introduction

Automatic tuning or autotuning is now regarded as one of key technologies in the future of high performance computing. There are several reasons. First, heterogeneous processing architecture with high performance accelerators, such as Cell, ClearSpeed and GPU, is becoming the mainstream of high performance computing. Architectural divergence requires software to adapt itself to various hardware platforms. Second, the high performance is sought by means of higher parallelism, especially a hierarchy of parallelisms: SIMD parallelism, multiple instructions, multiple threads, multiple cores, multiple nodes, and multiple clusters. It requires a hierarchy of parallelization of algorithms, and the optimal mapping of the algorithmic parallelisms to the hardware parallelisms is a non-trivial problem. Third, the progress of the high performance computers encourages the trials of solving more complex problems. Programs are becoming more and more complex, and thus hard to tune. Automatic tuning is expected to play an essential role to solve or at least mitigate those problems.

Research directions of automatic tuning can be problem specific as ATLAS[1] and FFTW[2], or general purpose as ABCLibScript[3], AutoPilot[4], and compiler technologies[5–7]. Both directions are necessary for progress of automatic tuning technology. This paper discusses an approach to a **general purpose technique of automatic tuning for parallel processing**, accompanied with an example of application to a specific problem, that is, data structure of parallel sparse matrix operations.

In parallel processing, two kinds of automatic tuning should be investigated. One is *tuning of local parameters* or *local tuning*, which optimizes tuning parameters whose influence is local to each processor, and the other is *tuning of global parameters* or *global tuning*, that affects executions of all the processors. Both of those two kinds of tuning are necessary in order to attain high performance in parallel processing. In this paper, we discuss **local tuning**.

We propose methods of experimental design for online automatic tuning of local parameters in parallel programs. Experimental design is an important topic of statistics, where the plan of experiments is optimized so to collect information efficiently. Our assumption here is that a finite number of alternative implementations, called *candidates*, are predefined, and the problem is to choose one of the candidates for each execution so that the total execution time is minimal. We consider **online automatic tuning**, where the experiments are done in the practical executions, unlike offline automatic tuning, where the trial executions to collect performance information are separated from the practical executions. In online automatic tuning, the costs of the experiments for performance evaluation are included in the total execution costs, so it is crucial to reduce the experimental costs and to collect performance information efficiently more than in offline automatic tuning.

The author has considered possibilities of several kinds of automatic tuning methods of parallel programs[8]. In this paper we propose three algorithmic implementations of (part of) those ideas, and present their applications.

The first method is **SEO (Serial Experiments Once)**, where each processor once measures the performance of all the candidates. Each processor chooses the best candidate for it. The second method is **PEO (Parallel Experiments Once)**, where the candidates are evaluated in a distributed manner over the parallel processors. Each processor observes the performance of a subset of the candidates that are assigned to it. Then the performance results are collected, and the candidate of the best performance is chosen, common for all the processors. If the true best choice differs on different processors, then PEO chooses a suboptimal candidate. The third method is **MPEO (Modified PEO)**, which is a modification of PEO. In MPEO each processor can replace the candidate chosen by PEO if it performs badly, or more specifically, worse than the candidate that performs best among those executed on that processor. Pros and cons of those methods are discussed, and results of application of those methods to an online optimization of parallel sparse matrix data structure are shown.

The rest of this paper is organized as follows. In the next section, we review basic concepts. In section 3, the algorithm we propose in this paper is described.

Section 4 will discuss the results of an application of our method to the problem of choosing data structure of sparse matrices. Section 5 is a summary of this paper, where future works are also discussed.

2 Automatic Tuning

2.1 Basic Terms

First, we introduce basic terms that we use in the following discussions.

Assume that the target software has a *tunable parameter*, and by choosing an appropriate value for that tunable parameter, the software can adapt to the environments. In an abstraction it is enough to consider a single tunable parameter, since multiple tunable parameters can be treated as a single vector of tunable parameters. We assume the set of values that can be assigned to the tunable parameter is finite, and call the assignable parameter the *candidates*.

Automatic tuning is a kind of optimization, where the objective function must be defined. The objective function can be time, precision, energy dissipation, monetary costs etc., but in this paper we assume the execution time as the objective function. Abstract automatic tuning is a problem to choose a value from the set of the candidates, with which value assigned to the tunable parameter, the objective function (execution time) is minimal.

In automatic tuning, the optimization is done by experiments, that is, by evaluating the performance of the software with a candidate value assigned to the tunable parameter. It is known that the optimization can be more efficient if the approximate performance model known to the programmer is utilized. Actually most of the autotuning research works utilize some form of performance model to make the optimization efficient. But in this paper we assume no a priori knowledge, and the optimization is conducted solely by experiments, as in some cases where the developer has not enough a priori knowledge to predict the performance as a function of the tunable parameter. Also we assume no perturbations in the performance measurements. We[9, 10] have proposed a Bayesian method to treat those factors — a priori knowledge on the cost functions and perturbations in the measurements. It is in our future works to combine Bayesian method with the following methods in order to treat a priori knowledge and measurement perturbations.

Tuning can be offline and online. In offline automatic tuning, the performance measurements and optimizations are done beforehand, separately from the practical executions. In practical executions, no trials, no measurements, and no optimizations are done. Online automatic tuning is a completely opposite idea. There will be no trials, no measurements, and no optimizations are done except in the practical executions. In this paper, we assume online tuning.

2.2 Methods of Automatic Tuning of Parallel Programs

As is mentioned above, we deal with **tuning of local parameters** of parallel programs in this paper. In general, the optimum values of local tunable parameter may be different on different processors. As we assume that no a priori

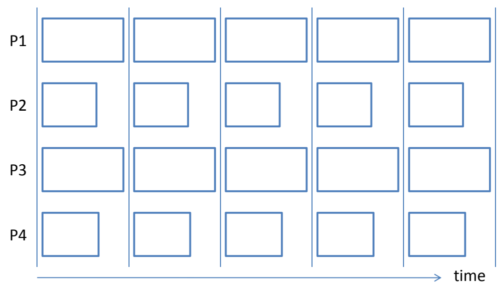

Fig. 1. Iterative computations in parallel processing — the rectangles represent computations, and the vertical bars represent synchronous communications.

knowledge is available, the only way to find the true optimum choice is to try all the candidates on each processor (which leads the idea of SEO). However, in many cases, we can expect that the performance of a candidate on a processor is correlated with its performance on another processor in a certain degree (which implies the idea of PEO).

To mold the ideas into concrete algorithms, we need assumptions how the processors interact each other. In this paper, we assume a simple model of data parallelism, as shown in Figure 1. The following model is appropriate, for example, with Krylov iterative linear solvers. The whole computation consists of a number of identical iterations. An iteration consists of a computation part and a communication part, where computations are independent and communication is synchronous. The computations of different processors may be different, but we assume some similarity. We assume some differences and correlations of the performance results of the candidates on the processors. In this paper, we consider automatic tuning of the computation part. Thus our aim is to minimize the execution time of the computation part on each processor.

3 Proposed Methods

In this section, we describe our methods of online automatic tuning of parallel programs. Let P be the number of processors, M be the number of the candidates, and K be the number of iterations. Let T_{ip} be the execution time of the computation part with the ith candidate on the pth processor.

3.1 SEO (Serial Experiments Once)

SEO evaluates all the candidates on each processor, and chooses the best performed one on each processor, independently. Figure 2 shows a pseudo-code of SEO in a SPMD (Single Program Multiple Data) model.

SEO evaluates each candidate once on each processor. Once is enough because we assume no perturbations of measured performance. After M iterations, each

```
(on the $p$th processor)
for $k = 1$ to $K$ do
    if ($k \leq M$) then
        Assign the $k$th candidate to the tunable parameter
    else
        if ($k = M + 1$) then
            $m_p = \mathrm{argmin}_m \{T_{mp}\}$
        endif
        Assign $m_p$ to the tunable parameter
    endif
    Do the computation
    if ($k \leq M$)
        $T_{kp}$ = the execution time of the computation
    endif
    Do the communication and synchronization
endfor
```

Fig. 2. Pseudo-code of SEO

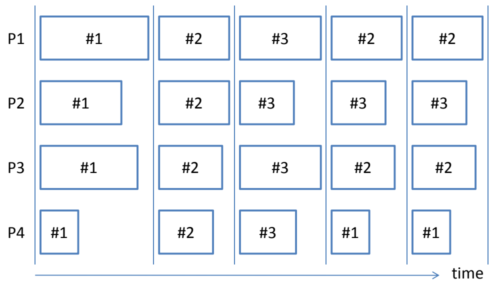

Fig. 3. SEO — all candidates are tried, and the choice is done independently on each processor.

processor finds the best value for the tunable parameter for it. The best value on the pth processor is m_p in Figure 2, and it is used in the later iterations.

Figure 3 depicts an example execution of SEO. There are three candidates, #1, #2, and #3. In the first three iterations, those candidates are chosen in each processor. In the fourth and the later iterations, the candidate which gave the minimum execution time is chosen on each processor.

The advantage of SEO is that it finds the optimum choice for all the processors. Thus it will be good for the number of iterations K is much larger than the number of the candidates M. However it tries all the candidates for performance evaluation, and the costs of that performance evaluation will be high if the number of the candidates M is large. So it may perform badly if the number of iterations K is relatively small.

3.2 PEO (Parallel Experiments Once)

Next we propose PEO, where parallel experiments are employed. PEO distributes the candidates into the processors. Each processor measures the performance of the candidates that are assigned to that processor. The performance data is gathered, and the candidate of the best performance is chosen on all the processors.

We assume that the computation on each processor is similar but may not be identical. Thus some processor may have more computations than the other processors. What we want to know is the difference of the performance of the candidates and not the difference of the computational loads of the processors. So it is not sensible to compare the candidates in the computing time T_{ip}.

We propose the following method. Before execution, one of the candidates is chosen. Let us call it the *reference* candidate. In the first iteration, all the processors use the reference candidate to measure the execution time T_{rp}, where r represent the reference candidate. The remaining $M - 1$ candidates are distributed among the processors, and evaluated on the assigned processors, from the second iteration. After all the candidates are evaluated, we define *relative performance* R_{ip} of the ith candidate executed on the pth processor as

$$R_{ip} = T_{ip}/T_{rp}$$

That is, the relative performance is the execution time relative to that of the reference candidate. Then, we calculate the minimum of R_{ip}, and the candidate that gives the minimum of R_{ip} is regarded as the optimum choice.

Figure 4 shows a pseudo-code of PEO. In this code, the reference candidate is assumed to be numbered as 1. First $K_0 = \lceil (M-1)/P \rceil + 1$ iterations are dedicated to experiments, and first of it is dedicated to the reference candidate. From the second iteration to the K_0th iteration, each processor evaluates one candidate per iteration. In the pseudo-code, PEO assigns the candidates cyclically to the processors. In the $K_0 + 1$st iteration, PEO determines which candidate performed best in the past executions. In the pseudo-code, MPI_Allreduce with MPI_MINLOC known in the MPI standard is used for illustration. That library routine collects the first arguments from all processors, and calculates the minimum of the first element in the collected first arguments. Then the minimum value is assigned to the first element of the second argument, and the second element accompanied with the minimum value is assigned to the second element of the second argument. The latter is m, which PEO regards the best choice. Note that this MPI_Allreduce is the only communication that PEO requires.

Figure 5 depicts an example execution of PEO. The reference candidate #1 is used in the first iteration on all the processors. In the second iteration, the other candidates #2 and #3 are evaluated in parallel. In the third and the later iterations, the candidate with the best relative performance is chosen, which is #2 in this case.

The advantage of PEO is the small number of experiments. Most of the candidates are evaluated only on one of the processors. As SEO requires M iterations for performance evaluation, PEO requires only $K_0 \approx M/P + 1$ iterations for

```
(on the $p$th processor)
$K_0 = \lceil (M-1)/P \rceil + 1$ // number of iterations for experiments
for $k = 1$ to $K$ do
    if ($k = 1$) then
        Assign 1st (reference) candidate to the tunable parameter
    else if ($k \leq K_0$) then
        $i = (((k-1)P + p) \bmod (M-1)) + 1$ // cyclic assignment of candidates
        Assign the $i$th candidate to the tunable parameter
    else
        if ($k = K_0 + 1$) then
            $m_p = \mathrm{argmin}_m \{T_{mp}/T_{rp}\}$
            $\mathrm{MPI\_Allreduce}((T_{m_p p}/T_{rp}, m_p), (R, m), \mathrm{MPI\_MINLOC})$
        endif
        Assign $m$ to the tunable parameter
    endif
    Do the computation
    if ($k \leq K_0$)
        $c =$ the chosen candidate
        $T_{cp} =$ the execution time of the computation
    endif
    Do the communication and synchronization
endfor
```

Fig. 4. Pseudo-code of PEO

performance evaluation. Thus PEO can find an approximate optimal choice in a less cost than SEO. The number of iterations spent for performance evaluation in PEO is almost $1/P$ of that of SEO. Thus, PEO will perform well if the number of the candidates M is large, and if the number of the iterations K is relatively small.

The disadvantage of PEO is that the chosen candidate (m in the pseudo-code) is not always the very optimal on each processor. Thus the asymptotic performance of PEO may be slightly lower than that of SEO. In cases the number of iterations K is large, the accumulated performance difference may be larger than the difference of the costs of performance evaluation. If K is much larger than M, then SEO will perform better than PEO.

3.3 MPEO (Modified Parallel Experiments Once)

MPEO is based on PEO. MPEO has a disadvantage that the chosen candidate m may not be optimal on some processors. In MPEO, each processor compares m (the chosen candidate) and the candidates known to the processor, and chooses the latter if it seems better than m. Figure 6 is a pseudo-code of MPEO.

The only differences of MPEO from PEO are the four lines from the line with the comment "local improvement," and the line with comment "record performance." After $K_0 + 1$st iteration, the performance of m (which is regarded

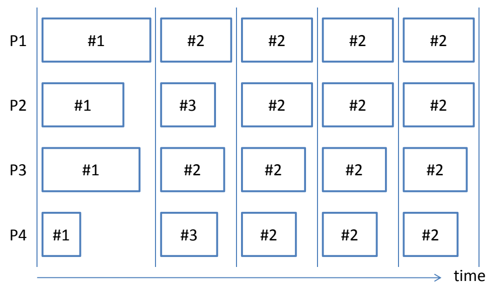

Fig. 5. PEO — all candidates are distributed, and the choice is done based on the performance relative to the reference.

as the global optimum) is evaluated ("record performance"). Then it is compared with the other candidates that have already been evaluated on that processor, and if the latter is better than the former, the latter is chosen on that processor ("local improvement"). Thus MPEO fully utilizes the available performance data on each processor. Again note that the only communication that MPEO requires is a single call of MPI_Allreduce.

Figure 7 depicts an example execution of MPEO. For the first three iterations, it goes in the same way as PEO. However before the fourth iteration, the observed performance of the candidate #2 is compared with the other candidates whose performance has been observed on each processor. In some processors, candidates other than #2 can be chosen in the fourth and the later iterations.

PEO will work well if the computations on the processors are identical. In that case, each candidate shows the same performance on every processor, thus it is enough to evaluate it on one of the processors. However, if the computation parts on the processors are different, performance measured on a processor may not be very informative for the other processors. To obtain accurate performance information, one has to pay the costs of experimental executions. MPEO tries to reduce this difficulty with the minimum additional costs, without doing any additional experiments, by choosing the best candidate from those whose performance on each processor is already known.

Considering only of the execution times of the computational parts, MPEO will perform better than PEO. Looking at more precisely, MPEO requires a little more computations, one more performance measurement, and change of the tunable parameter. Thus there can be cases where MPEO performs slightly lower than PEO.

4 Application to Parallel Sparse Matrix Data Structure

In the previous section, we have shown three methods, SEO, PEO and MPEO for online automatic tuning of parallel programs. From the discussion, we can

```
(on the $p$th processor)
$K_0 = \lceil (M-1)/P \rceil + 1$ // number of iterations for experiments
for $k = 1$ to $K$ do
    if ($k = 1$) then
        Assign 1st (reference) candidate to the tunable parameter
    else if ($k \le K_0$) then
        $i = (((k-1)P + p) \bmod (M-1)) + 1$ // cyclic assignment of candidates
        Assign the $i$th candidate to the tunable parameter
    else
        if ($k = K_0 + 1$) then // approximate global minimum
            $m_p = \mathrm{argmin}_m \{T_{mp}/T_{rp}\}$
            $\mathrm{MPI\_Allreduce}((T_{m_p p}/T_{rp}, m_p), (R, m), \mathrm{MPI\_MINLOC})$
        else if ($k = K_0 + 2$) then // local improvement
            if ($T_{m_p p} < T_{mp}$) then
                $m = m_p$
            endif
        endif
        Assign $m$ to the tunable parameter
    endif
    Do the computation
    if ($k \le K_0 + 1$) // record performance
        $c$ = the chosen candidate
        $T_{cp}$ = the execution time of the computation
    endif
    Do the communication and synchronization
endfor
```

Fig. 6. Pseudo-code of MPEO

conclude that, MPEO will perform no worse than PEO, and MPEO will perform better than SEO when the number of iteration K is relatively small and the number of candidates M is relatively large.

This section shows an application of those methods to parallel sparse matrix implementation. The objective of this section is to confirm the above properties of the tuning methods.

4.1 Sparse Matrix Data Structures

The candidates in the following experiments are the data structures of sparse matrices. Sparse matrices are matrices whose elements are mostly zeros. In matrix operations, such as the product of a matrix and a vector, zero elements have no effect on the result. In such cases, it is enough to store and to access data regarding only non-zero elements. By doing so, both the computational time and the memory usage can be saved.

Several data structures for sparse matrices have been proposed. If the places of the non-zero elements of the given matrix has some characteristics — for

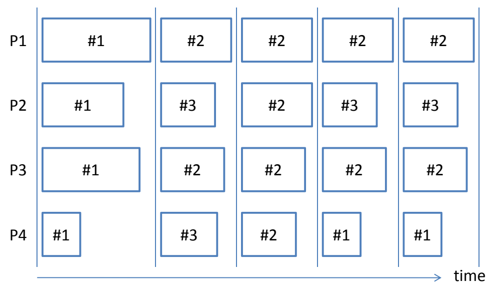

Fig. 7. MPEO — similar to PEO, but the candidate of the best performance observed on each processor is chosen before the 4th iteration.

example, all non-zero elements are on a few lines parallel to the diagonal, or, the non-zero elements appear in a 3×3 dense block in the matrix — then a data structure that utilize such characteristics will be efficient. For matrices for which none of the known characteristics applies, general data structures should be used.

Let us predefine several data structures as candidates, and choose one for each given matrix. There are several research works on automatic tuning of sparse matrix data structures (e.g. [11, 12]). They build a mathematical model formula that predicts the performance of each data structure when applied to the given matrix. However, in this paper, we do not assume any performance model. We rely on experiments, and use the online automatic tuning methods described in Section 3.

4.2 Candidate Data Structures

First we discuss data structure on each processor, and then discuss how parallel computation is implemented.

We employ a subset of sparse matrix data structures defined in a sparse solver library lis[13]. We implemented those data structures by ourselves, rather than using the lis library. The reason is this: lis is a sparse matrix solver, so it assumes square matrices. However, we assume parallel processing, where matrices are distributed into the processors with block distribution in the row direction (which is standard in parallel iterative solvers). Then each processor treats a rectangular submatrix. Although it is not impossible to treat rectangular matrices in data structures for square matrices, it is better to use data structures for rectangular matrices.

We implemented the following data structures.

1. Coordinate (COO),
2. Compressed Row Storage (CRS),
3. Compressed Column Storage (CCS),

4. Modified compressed Sparse Row (MSR),
5. Modified compressed Sparse Column (MSC),
6. Diagonal (DIA),
7. Jagged Diagonal Storage (JDS),
8. Block compressed Sparse Row (BSR),
9. Block compressed Sparse Column (BSC).

We omit the detailed definitions of those data structures, partly because none of them is new, and partly because we do not build any performance model for them. They are described in [13]. Just let us mention some characteristics of those data structures.

COO, CRS and CCS are general purpose data structures. They do not store any zero elements. Usually CRS outperforms COO and CCS in cache-based architectures. MSR and MSC are modifications of CRS and CCS, respectively, and perform slightly better if the diagonal elements are nonzero (as is often seen in scientific computing). They treat diagonal elements in a way different from the other elements. We modified the data structure (from the implementation of lis) so that an arbitrary line parallel to the diagonal can be the treated in such a special way. This is required when it is used in parallel processing: The diagonal of the whole matrix may not be the diagonal of the submatrix assigned to a processor. DIA works well if the non-zero elements are on a few lines parallel to the diagonal. JDS is known to perform well on vector processors. BSR and BSC assume that the non-zero elements are in submatrices with size $r \times c$, where r and c is constants. In our implementation, if the number of rows is not a multiple of r, we add a few (less than r) rows in CRS form. Similarly, if the number of columns if not a multiple of c, we add a few (less than c) columns in CCS form. We wrote routines for the product of a matrix in BSR or BSC data structure and a vector, where the computations with $r \times c$ blocks are fully unrolled for $1 \leq r \leq 4$ and $1 \leq c \leq 4$ (thus 16 routines for each of BSR and BSC, but we don't use those with $r = c = 1$). We have $7 + 15 + 15 = 37$ candidates. Figure 8 plots the execution times of matrix-vector products with those data structures, for matrix af23560 distributed over 32 processors. For this matrix BSR performs best, but the best block sizes varies.

For parallel processing, we use MPI (so-called flat MPI). The matrix is divided into several submatrices. To each processor consecutive rows are assigned. The matrix is divided so that each processor has approximately equal number of rows. This is known as block distribution, and one of the standard methods of parallel sparse matrix computations. Each processor stores the submatrix assigned to it, and the parts of vectors which are accessed in matrix-vector product operations. In this experiment, the vector values are not communicated among the processors. Instead, MPI_Barrier is called after the computation in each iteration.

4.3 Online Automatic Tuning Methods

The computations in the performance evaluation are done in the following way. First, the matrix is loaded from a file on the memory in COO data structure. The

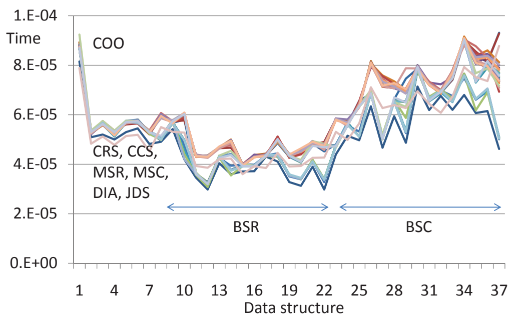

Fig. 8. Performance of matrix data structures for af23560

matrix file is formatted in MatrixMarket format, from which COO is most convenient to convert. So we also use COO data structure as the *reference candidate* in PEO and MPEO.

The COO data structure on each processor is kept in the computation. When a data structure other than COO is used, the data structure is generated from COO data structure. We call this operation *conversion*. To convert a sparse matrix in the COO data structure into another data structure, computations and memory are needed. We implemented conversion routines from COO to the other data structures. To convert from COO to BSR and BSC, CRS and CCS are used temporarily, respectively. There can be several possible algorithms of conversions, and we employ an algorithm which requires memory and computations linear to the output data structure. Conversions are independently done on each processor.

For the set of the candidates, we employ 37 data structures discussed in the previous section. As is mentioned above, COO data structure is kept throughout the iterations. In addition to COO, we keep a second data structure, say, CRS, on memory. If another data structure than them, for example CCS, is to be used, then CRS is destroyed, and CCS is constructed. The time for a conversion is typically 5 to 20 times longer than the time of a matrix-vector product. Thus trying a new data structure is quite costly.

As methods of online automatic tuning, we employ SEO, PEO, and MPEO discussed in Section 3. As is already mentioned, COO data structure is used as the reference candidate in PEO and MPEO. In addition, for a comparative purpose, we implemented NoTune, where only COO data structure is used throughout the iterations. It requires no conversion, so it can be efficient if the number of iterations (K in Section 2) is very small.

4.4 Experimental Results

We use TSUBAME supercomputer in Tokyo Institute of Technology. Each node has eight sockets of dual core AMD Opteron CPU (16 cores on a node) in 2.4

GHz, connected with 4× SDR InfiniBand. Compiler is PGI C compiler version 7.2 and the used option is -O4. OS is SUSE Linux Enterprise Server 10 SP2.

Table 1 shows the tested matrices. They are obtained from the University of Florida Sparse Matrix Collection [15]. It is a collection of thousands of matrices, and from that we chose a few real unsymmetric matrices which show different characteristics. We have also tried many other matrices, and those matrices listed in Table 1 showed representative performance characteristics. The best data structure in Table 1 is the data structure chosen on the majority of processors in SEO. For torso2, various data structures were chosen on different processors in SEO.

Table 1. Test Matrices

Matrix	Size	Nonzeros	Best Data Str.	Problem
af23560	$23,560 \times 23,560$	460,598	BSR	Fluid
ted_A	$10,605 \times 10,605$	424,587	DIA	Thermal
xenon1	$48,600 \times 48,600$	1,181,120	BSR	Materials
torso2	$115,967 \times 115,967$	1,033,473	depends	2D/3D

Table 2 shows the performance results. The results are shown in the form of $\alpha + \beta n$. This approximates the execution time of n iterations in ms. The figures α and β are calculated as follows. The number of iterations is chosen as 50. Since we have 37 candidates, the chosen method is fixed before 38th iteration in SEO, and the last 10 iterations are executed with the finally chosen candidate. The execution time per iteration β is determined as the average execution time of the last 10 iterations. The startup time α is determined so that $\alpha + 50\beta$ equals the total execution time of 50 iterations.

First let us discuss about β, which is the execution time per iteration. In most cases, β is larger for NoTune (without automatic tuning) than SEO and MPEO (with automatic tuning). In many cases, SEO outperforms PEO in terms of β. However, β of SEO and MPEO are comparable. Only from Table 2, we can hardly say that β of PEO is consistently smaller than that of NoTune.

Next investigate on α, which represents the overheads of the experiments for automatic tuning. It is natural that $\alpha \approx 0$ for NoTune. The overheads of SEO, PEO and MPEO are significant. In many cases α is larger than β for two or three orders of magnitude. This suggests that the automatic tuning is effective when the number of iterations n is larger than 100 or 1000. Differences of PEO and MPEO from SEO are clear. Overheads of SEO are much higher than those of PEO and MPEO.

In summary, the proposed MPEO successfully tunes the parallel data structure of sparse matrices with much lower overheads than SEO.

As is shown in Table 2, the measured performance is unstable in some cases. Thus our assumption that no perturbation is observed in performance evaluations is not true. Sometimes the performance figures are superlinear, but we

Table 2. Performance Results — total execution time approximated as $\alpha + \beta n$ (ms) where n is the number of iterations, 2 significant figures

Matrix	P	NoTune	SEO	PEO	MPEO
af23560	4	$0.0 + 2.9n$	$730 + 1.3n$	$230 + 2.1n$	$200 + 1.5n$
af23560	8	$0.89 + 0.66n$	$390 + 0.25n$	$80 + 0.32n$	$110 + 0.39n$
af23560	16	$0.0 + 0.27n$	$130 + 0.20n$	$17 + 0.20n$	$26 + 0.18n$
af23560	32	$1.1 + 0.23n$	$70 + 0.34n$	$4.4 + 0.34n$	$5.1 + 0.38n$
ted_A	4	$0.0 + 3.5n$	$1100 + 1.2n$	$140 + 29n$	$250 + 1.1n$
ted_A	8	$1.4 + 1.5n$	$690 + 0.46n$	$88 + 0.57n$	$82 + 0.41n$
ted_A	16	$0.61 + 0.35n$	$250 + 0.27n$	$39 + 0.24n$	$47 + 0.20n$
ted_A	32	$0.0 + 0.40n$	$71 + 0.43n$	$4.7 + 0.47n$	$12 + 0.62n$
xenon1	4	$17 + 4.9n$	$2000 + 2.6n$	$630 + 3.0n$	$640 + 2.4n$
xenon1	8	$1.8 + 3.1n$	$1100 + 1.6n$	$460 + 3.9n$	$260 + 1.5n$
xenon1	16	$16 + 1.2n$	$720 + 0.32n$	$140 + 1.3n$	$140 + 0.82n$
xenon1	32	$1.5 + 0.34n$	$490 + 0.36n$	$33 + 0.26n$	$62 + 0.23n$
torso2	4	$13 + 4.9n$	$3900 + 3.6n$	$2200 + 3.6n$	$1400 + 3.7n$
torso2	8	$3.4 + 3.5n$	$1200 + 2.0n$	$380 + 2.1n$	$380 + 2.8n$
torso2	16	$7.2 + 1.4n$	$660 + 0.58n$	$120 + 0.38n$	$130 + 0.49n$
torso2	32	$1.4 + 0.26n$	$290 + 0.26n$	$49 + 0.32n$	$72 + 0.25n$

guess that they come from the instability of the performance. We have measured the execution times of the computational parts and the communication (synchronization) parts separately, and found that the unstable execution time is mostly due to the communication parts. The instability must be the reason why sometimes SEO performs slightly worse than MPEO.

5 Conclusion

In this paper, we have proposed methods of parallel experiments for online automatic tuning of parallel programs. As far as the author knows, this is the first discussion of parallel online automatic tuning of parallel programs with parallel experiments. In our experiments, MPEO has been able to choose sub-optimal data structures in parallel sparse matrix-vector products, while SEO can give the optimal choice and thus be the best method for asymptotically many iterations.

In this paper, we assume that no a priori information about the performance of the candidates is available, and that no perturbation is observed in performance evaluation. Those factors are considered in the works such as [9] in a Bayesian framework. Our future plans include application of Bayesian framework in parallel experimental design for parallel processing.

Our algorithms discussed in this paper assume a very simple model of parallel processing as shown in Figure 1. This model is appropriate to several important applications of high performance computing. But it will be beneficial to investigate automatic tuning methods for other parallel processing models such as task parallel models.

Acknowledgements

This work is partially supported by Grant-in-Aid for Scientific Research "Research on Mathematical Core for Robust Auto-Tuning System in Information Explosion Era" of MEXT Japan, and Core Research of Evolutional Science and Technology (CREST) project "ULP-HPC: Ultra Low-Power, High-Performance Computing via Modeling and Optimization of Next Generation HPC Technologies" of Japan Science and Technology Agency (JST).

References

1. Whaley, R. C. and Dongarra, J. J.: Automatically Tuned Linear Algebra Software, *Proceedings of SC98*, (CD-ROM), 1998.
2. Frigo, M. and Johnson, S. G.: FFTW: an adaptive software architecture for the FFT, *Proceedings of ICASSP '98*, Vol. 3, pp. 1381–1384, 1998.
3. Katagiri, T., Kise, K., Honda, H., and Yuba, T.: ABCLibScript: A directive to support specification of an auto-tuning facility for numerical software, *Parallel Computing*, Vol. 32, No. 1, pp. 92–112, 2006.
4. Ribler, R. L., Simitci, H., Reed, D. A.: The Autopilot performance-directed adaptive control system, *Future Generation Computer Systems*, Vol. 18, No. 1, pp. 175–187, 2001.
5. Kulkarni, P., et al.: Practical exhaustive optimization phase order exploration and evaluation. *ACM TACO*, Vol. 6, No. 1, 2009.
6. Chen, C, et al.: Model-Guided Empirical Optimization for Multimedia Extension Architectures: A Case Study. *IPDPS 2007.* pp. 1-8, 2007.
7. Fursin, G., et al.: MiDataSets: Creating the Conditions for a More Realistic Evaluation of Iterative Optimization. *HiPEAC 2007*, pp. 245-260, 2007.
8. Suda, R.: Mathematical Models for Software Automatic Tuning on Parallel Processors, *Proc. Ann. Meeting of JSIAM*, pp. 13–14, 2009.
9. Suda, R.: A Bayesian Method for Online Code Selection: Toward Efficient and Robust Methods of Automatic Tuning, *Proc. 2nd Int'l Workshop on Automatic Performance Tuning (iWAPT2007)*, pp. 23–32 (2007).
10. Suda. R.: Mathematics of Software Automatic Tuning, *Proc. IPSJ*, Vol. 50, No. 6, pp. 487–493, 2009.
11. Vuduc, R., Demmel, J. W. and Yelick, K. A.: OSKI: A library of automatically tuned sparse matrix kernels, *Proc. SciDAC 2005, Journal of Physics: Conference Series*, 2005.
12. Im, E.-J., Yelick, K., and Vuduc, R.: SPARSITY: An optimization framework for sparse matrix kernels, *Int'l J. of High Performance Computing Applications*, Vol. 18, No. 1 pp. 135–158, 2004.
13. Kotakemori, H., et. al: Performance Evaluation of Parallel Sparse Matrix–Vector Products on SGI Altix3700, *First International Workshop on OpenMP (IWOMP2005)*, LNCS Vol. 4315, pp. 153–163, 2008.
14. The Matrix Market,
    `http://math.nist.gov/MatrixMarket/`
15. The University of Florida Sparse Matrix Collection,
    `http://www.cise.ufl.edu/research/sparse/matrices/`