Parallelizable C and Its Performance on Low Power High Performance Multicore Processors

Masayoshi Mase, Yuto Onozaki, Keiji Kimura, and Hironori Kasahara

Department of Computer Science and Engineering, Waseda University
3-4-1 Ohkubo, Shinjuku-ku, Tokyo, 169-8555, Japan
{mase,yonozaki,kimura,kasahara}@kasahara.cs.waseda.ac.jp

Abstract. As multicore processors are getting popular, demands for automatic parallelization achieving high performance, low power, and short software development period are increasing. However, automatic parallelization of C programs has been difficult due to the pointer analysis. This paper proposes Parallelizable C, which is a kind of programming methods for sequential C programs that allows a parallelizing compiler to analyze pointers precisely, and its compilation framework. Coding with Parallelizable C mitigates application programmers from complicated parallel tuning. Parallelizable C allows ordinary programmers to leverage full benefits of exploiting multi level parallelism and data locality optimization for effective use of cache or local memory without hard programming efforts. Application programs written in Parallelizable C can be automatically parallelized by OSCAR (Optimally Scheduled Advanced Multiprocessor) parallelizing compiler. The OSCAR compiler automatically generates parallel C source code with OSCAR API opened in http://www.kasahara.cs.waseda.ac.jp/ which is a parallel processing API designed for real-time embedded low-power multicores and shared memory multiprocessor servers from different vendors. The generated parallel programs with OSCAR API are compiled into executable binary for the target architecture using a sequential compiler with an API analyzer or an OpenMP compiler. In our experience, sequential C programs such as AAC encoder and MPEG2 encoder from the multimedia application domain and SPEC2000 "art", "equake", SPEC2006 "hmmer" and "lbm" from the numerical application domain require modifying only a few percentages of the program code to achieve automatic parallelization. The evaluation on an 8 cores server IBM p5 550Q with Power5+ processors shows that the compiler automatic parallelization gives us 5.43 speedup in average against sequential execution. Also, on a 4 cores PC with Intel Core i7, automatic parallelization gives us 2.34x speedup, and on an embedded multicore Renesas/Hitachi/Waseda RP2 with SH-4A processor cores in 4 cores SMP execution mode, automatic parallelization gives us 2.80x speedup in average against sequential execution, respectively.

1 Introduction

Multicore processors have been widely used in various computer systems such as PCs, workstations, and high-performance computers as well as embedded systems including consumer electronics. Though parallel software is necessary to leverage full potential of these multicore processors, parallel programming is still a challenging issue that requires long development period to obtain expected performance even of expert parallel programmers. While current multicores have several architectural features responsible for achieving high performance with low power, the cost for application development is increasing since programmers must deal with the machine specific tuning such as memory management, synchronization, and power management. Therefore, automatic parallelization of conventional sequential programming languages is desired to improve software productivity through achieving both of high performance and low power in a short software development period on various multicore platforms by a compiler.

The programming language C is one of the most popular programming lauguages often adopted for various kinds of applications including numerical computation and media processing. Especially in embedded domain, C language is the de fact standard programming language because of capability of generating fast executables in straight forward compilation process as well as its flexibile pointer usage capable of accessing arbitrary memory address space convinient for manipulating hardware devices.

Automatic parallelizing compiler such as Polaris compiler[1], SUIF compiler [2], and OSCAR compiler[3] have been developed and its performance was demonstrated on various parallel machines for FORTRAN programs in scientific computation. However, automatic parallelization of C programs has been difficult due to the difficulty of pointer analysis. The specification of C language allows flexible descriptions with extensive use of pointers while such flexible pointer usages prevent the compiler to analyze where a pointer variable points to. Spurious pointer alias information interferes dependence analysis, which hampers automatic parallelization by a compiler. There are no perfect pointer analysis which can statically determine all the objects pointed-to by all the pointers for all coding patterns allowed in the specification of C language, while extensive studies on pointer analysis[4] have shown potential to optimize C programs at some degree of precision if we combine with other program analyses for parallelism extraction[5].

Coding guidelines for C programs are widely used in industry such as MISRA-C[6], which is a guideline for critical systems that requires high reliability, dependability and portability. They limit the flexibility of C language specification to utilize software tools including compilers while improving software quality and visibility for programmers. Although guidelines for multicore processors such as Clean C[7] by IMEC and implicitly parallel programming model which encourages hint information from user assertions by Hwu et.al[8] are proposed, automatic parallelization performance and amounts of user effort were not evaluated.

This paper proposes Parallelizable C, which enables programmers to leverage the full benefits of an automatic parallelizing compiler such as exploiting multi level parallelism and data locality optimization for effective use of cache or local memory. Parallelizable C is a subset of C language so that application programmers need not to learn a new programming language. Coding with Parallelizable C mitigates application programmers from complicated parallellization techniques, thus programmers can concentrate on algorithm tuning itself. Furthermore, even if a programmer is not familiar with parallel processing, the Parallelizable C enables scalable performance improvement on multiprocessor or multicore systems.

In this paper, we clarify the effectiveness of Parallelizable C by insights on code rewriting process and speedup by automatic parallelization on real multicore platforms for several application programs. For scientific and multimedia applications, the amount of rewriting sufficient for automatic parallelization by the OSCAR compiler is small. The main contribution of this paper are the evaluation of the parallel processing performance on existing multicore platforms with an automatic parallelizing compiler, on top of the code rewriting according to coding rules, and defining the coding guideline considering the compiler analysis precision.

The rest of the paper is organized as follows. First, Section 2 describes the OSCAR parallelizing compiler and OSCAR API to explain the target parallel processing model. Second, section 3 describes pointer analysis essential for automatic parallelization of C programs. Then, section 4 shows the concept and specification of Parallelizable C. Finally, section 5 demonstrates the rewriting of programs to Parallelizable C and section 6 gives the parallel processing results on multicore systems.

2 OSCAR Parallelizing Compiler and OSCAR API

In this section, we briefly explain how the multigrain parallelism and data locality are extracted through the compilation process in the OSCAR parallelizing compiler[3].

2.1 Overview of Multigrain Parallelization

Multigrain parallel processing exploits multiple grains of parallelism such as coarse grain task parallel processing, loop iteration level parallel processing, and statement level near fine grain parallel processing. In this study, loops, function calls, and basic blocks are defined as coarse grain tasks.

Macro Task Generation: In order to apply multigrain parallel processing to an ordinary sequential program, the OSCAR compiler firstly decomposes a source C or Fortran program into coarse grain tasks, namely macro-tasks (MTs), such as basic block (BPA), loop (RB), and function call or subroutine call (SB).

Coarse Grain Task Parallel Processing: Then, the compiler analyzes both the control flow and the data dependencies among MTs and represents them as a macro-flow-graph (MFG). Next, the compiler applies the earliest executable condition analysis[9], which can exploit parallelism among MTs associated with both the control dependencies and the data dependencies. The analysis result is represented as a macro-task-graph (MTG). If an MT is a subroutine call or a loop that has coarse grain task parallelism, the compiler hierarchically generates inner MTs inside that MT. Also, loops that has loop-level parallelism is translated to coarse grain task parallelism by loop decomposition. Then, the compiler groups processor cores into processor groups (PG) logically and hierarchically considering the parallelism in each layer of hierarchical MTG. These MTs are assigned to processor cores by the compiler. If the MTG has conditional branches or runtime fluctuations, dynamic scheduling is applied to it. Otherwise, static scheduling is applied[10].

Data Locality Optimization: Data locality optimization over the whole program and data transfer optimization using DMA controller are applied to MTGs. If multiple MTs share same data, whose size is greater than that of the cache memory or the local memory, the OSCAR compiler decomposes these MTs into smaller MTs in order to fit the shared data accessed by each MT into the cache or the local memory by loop aligned decomposition[11]. Then, these decomposed MTs are scheduled onto processor cores in order to assign MTs, which access same smaller data, successively as much as possible[12].

2.2 OSCAR API and Its Compilation Framework

The OSCAR API[13] is a parallel processing API which consists of annotations to C and FORTRAN programs. The API is designed on a subset of OpenMP for preserving portability over a wide range of multicore architectures, while original features are added to support emerging low power real-time multicore processors such as power management, data assignment to various memories, and explicit data transfer among memories. Technical details of OSCAR API is shown in [14] and its specification (ver 1.0) for heterogeneous multicore has been open in http://www.kasahara.cs.waseda.ac.jp/.

The compile flow of the OSCAR compiler with the OSCAR API is described in Fig. 1. First, a sequential C (or Fortran) program are parallelized by the OSCAR compiler. The OSCAR compiler generates a C (or Fortran) code with the OSCAR API. This parallelized code is then compiled by an OpenMP compiler for a server platform, or it is translated into C (or Fortran) program with runtime library calls generated by the API analyzer in front of the backend compiler of the target multicore. Finally, the backend compiler generates an executable object for the target multicore.

3 Pointer Analysis for Automatic Parallelization of C Programs

Pointer analysis is an essential program analysis in C compiler, which statically determines where a pointer variable points to. The precision of pointer analysis influences succeeding analyses such as loop dependence analysis, data flow

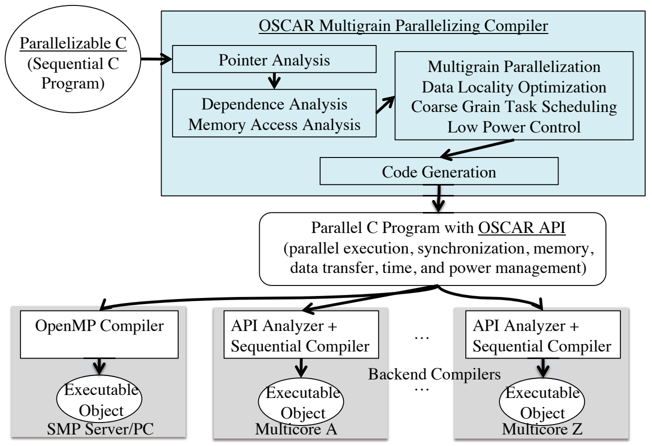
Parallelizable C
(Sequential C Program)
OSCAR Multigrain Parallelizing Compiler
Pointer Analysis
Dependence Analysis
Memory Access Analysis
Multigrain Parallelization
Data Locality Optimization
Coarse Grain Task Scheduling
Low Power Control
Code Generation
Parallel C Program with OSCAR API
(parallel execution, synchronization, memory, data transfer, time, and power management)
OpenMP Compiler
API Analyzer + Sequential Compiler
API Analyzer + Sequential Compiler
Backend Compilers
Executable Object
Executable Object
Executable Object
SMP Server/PC
Multicore A
Multicore Z

Fig. 1. Compile flow using OSCAR compiler and OSCAR API

analysis, and data access region analysis which are necessary for automatic parallelization.

This section summarizes the current status of pointer analysis to provide base knowledge for deciding the specification of the Parallelizable C. Pointer analysis is discussed from viewpoints of various sensitivities correspond to control flow as well as data structure. The trade off between precision and efficiency on these sensitivities are often discussed considering the feature of target optimization[4].

For automatic parallelization, highly precise pointer analysis is necessary to achieve high performance. Putting precise pointer analysis techniques into altogether has potential to construct powerful pointer analysis framework helpful for automatic parallelization.

3.1 Sensitivity for Control Flow

Flow-sensitivity and context-sensitivity are major categories that are concerned about control flow. Flow-sensitive analysis generates information for each statement and propagate it along the control flow throughout the program. Context-sensitive analysis generates information for each function call site for a function.

Classically, flow-sensitive context-sensitive pointer analysis algorithm is formulated in dataflow framework [15]. In flow-sensitive and context-sensitive pointer analysis, huge amount of memory and analysis time are often required. Various schemes[16–19] are proposed to realize practical analysis time for flow-sensitive and context-sensitive pointer analysis.

3.2 Sensitivity for Data Structure

Pointer information for data structure is also an essential factor for pointer analysis, so that field-sensitivity and heap-modeling are often discussed.

Field-sensitive pointer analysis[20] handles each structure member as an independent object and is necessary to analyze aplications whose data are composed in structures.

Regarding to heap modeling, resent pointer analysis distinguishes heap name by each function call statement corresponds to its memory allocation function like malloc(). However, heap is often obtained via user defined wrapper functions like xmalloc(), therefore heap cloning[21] is adopted to improve precision by distinguishing call sites of wrapper functions.

Furthermore, there are another problems caused by the limitation of the property of flow-sensitive dataflow analysis. In the fixpoint calculation of dataflow analysis, the heap object allocated in the current loop iteration and heap objects allocated in previous iterations are aggregated into an amortized object. A technique called aging[22] can cope with this problem by naming heap with age that indicates whether a heap object is allocated in the current iteration or previous iterations.

Another important sensitivity on data structure is elements in array of pointers. Element-wise points-to set[23] maps the iteration information of heap allocation on elements in an array of pointer object. This kind of sensitivity for array elements is important to distinguish array-like data structures on heap in order to apply automatic parallelization and data locality optimization techniques targetting arrays and loops.

4 Parallelizable C

The main contribution of the Parallelizable C is clarifying what kind of description can be analyzed and parallelized by a compiler.

Implementing pointer analysis described in section 3 can increase the optimization level of automatic parallelization by a compiler. However, it is impossible to implement the perfect pointer analysis which covers all patterns of descriptions allowed in the C language specification. Thus, writing sequential programs in the capability of pointer analysis is the essential point for the practical use of an automatic parallelizing compiler.

4.1 Target Optimization

We are assuming extracting full potential of conventional framework for parallelization and data locality optimization mainly targeting arrays and loops [11, 24]. Thus, parallelization of recursive data structure such as lists and trees are out of scope for the Parallelizable C guideline proposed in this paper.

4.2 Assumed Compiler Analysis Precision

The Parallelizable C coding guidelines assumes the automatic parallelizing compiler have pointer analysis and field-sensitivity for structures in data dependence analysis as described in following:

– The pointer analysis is along control flow and generates points-to information for each statement. (flow-sensitive)
– The pointer analysis processes for each invocation site of function call. (context-sensitive)
– The pointer analysis distinguishes members of a structure. (field-sensitive)
– The pointer analysis distinguishes heap objects by the call path to the allocation sites. (heap-cloning)
– The pointer analysis distinguishes an iteration of memory allocation site for heap object. (cycle-sensitive)
– The pointer analysis distinguishes whether each element in an array of pointers points-to different memory region. (element-sensitive)
– The pointer analysis distinguishes whether a poiter variable points-to the head of the pointed object.

Furthermore, succeeding data dependence analysis also requires these kinds of sensitivities. Especially, nested relationships between arrays and structures are complicated for array-based optimizations. The Parallelizable C also assumes the sensitivity of the analysis on structure and array as follows.

– The data dependence analysis distinguishes members of a scalar structure.
– The data dependence analysis does not distinguish members of an array of structures.

4.3 Coding Rules

In the Parallelizable C, programmers are encouraged to write code complying with the rules described in this section. This specification consists of coding rules and hint information for external function call. Deviation from the coding rules is allowed because the Parallelizable C accepts an ordinary C program to be compiled though some program parts may be difficult to parallelize. The Parallelizable C coding rules are as follows. These limitations are due to the program analysis precision described in the Sec. 3.

4.4 Coding Rules Related to Pointer Usage

Following pointer usages are not recommended:

– Accessing out of declared boundary on members of structure or each dimension of multi-dimensional array.
– Pointer casts except for heap allocation site.
– Pointer arithmetic. (When we access the region pointed to by a pointer, use dereference operator (*) or index operator ([ ]) on the pointer variable without updating pointer value.)
– Updating pointer value in a conditional branch or a loop except for heap allocation site.
– Passing arguments of function as pointer to different offset of same dimension of an array. (Thus, regions pointed to by function arguments in callee never overlapped.)

- Reusing a heap object as a temporal buffer. (Reuse of heap can be an obstacle of parallelism extraction since it is more difficult to analyze the reuse of a heap object than reuse of an ordinaly variable.)
- Index accesses to an array member of an array of structures. (If an array of structures has an array member, pass one element of the array of structures as an argument of a function and access the member in the callee function.)

4.5 Coding Rules Related to Function Call

Following function calls are not recommended:

- Recursive function calls. (A function does not call itself neither directly nor indirectly.)
- Function pointers. (Use switch-case statements instead of function pointers.)

4.6 Hint for External Functions

The OSCAR compiler assumes whole program analysis on Parallelizable C. When library functions are required, a hint annotation on the behavior of the external function is used to give an assumption to the pointer analyses in the compiler. The assumptions are as follows.

- Assuming all pointer values do not update in all external functions.
- Assuming the pointer return value points to a new heap object for all external functions.

5 Code Modifications to Parallelizable C

This section shows examples of code modification to the Parallelizable C. SPEC2000 “art”, “equake”, SPEC2006 “lbm”, “hmmer”, and MediaBench “mpeg2encode” are used as benchmark application programs.

5.1 Amount of Code Modifications

The amount of modifications is measured in the metric of how many lines of code (LOC) are modified from the original source code. The original code of the program is rewritten into the Parallelizable C, then they are simply compared with the original C code by *diff* command ignoring white spaces and indents. The amount of modifications is the greater one when the amount of deleted lines and added lines are compared. The amout of code modification for each application program is shown in Table 1.

SPEC2000 “art” and “equake” are compliant to the Parallelizable C code in original version so that code modification is not necessary. SPEC2006 “lbm” and “hmmer” requires code modification related to Parallelizable C. Regarding to “mpeg2encode”, code modification is applied from the standpoint of algorithm, in order to extract parallelism and data locality among macroblocks that is not available in the original program structure in MediaBench[25]. Thus, the amount of rewritten code is more than previous examples due to the modification of data structures and control structures.

Table 1. Amount of code modifications to the Parallelizable C

application program	total (LOC)	subject	deleted (LOC)	added (LOC)	modified portion(%)
SPEC2000 art	1270	N/A	0	0	0%
SPEC2000 equake	1513	N/A	0	0	0%
SPEC2006 lbm	1155	pointer to middle of object	-7	+7	1.8%
		pointer swap in a loop	-10	+ 14	
SPEC2006 hmmer	35992	reuse of heap	-9	+8	0.03%
Mediabench mpeg2encode	3750	algorithm improvement	-640	+864	23.5%

```
for( t = 1; t <= param.nTimeSteps; t++ ) {
   ...
   LBM_performStreamCollide( *srcGrid, *dstGrid ); /* main computation */
   LBM_swapGrids( &srcGrid, &dstGrid ); /* swap pointer value of srcGrid and dstGrid */
   ...
}
```

(a) Original C

```
flg = 0;
for( t = 1; t <= param.nTimeSteps; t++ ) {
    ...
    if (flg % 2)
        LBM_performStreamCollide( *dstGrid, *srcGrid ); /* main computation */
    else
        LBM_performStreamCollide( *srcGrid, *dstGrid ); /* main computation */
    flg++;
    ...
}
```

(b) Parallelizable C

Fig. 2. Removing pointer update inside a Loop in "lbm"

5.2 Parallelizable C Code Example

Here, the code examples of the Parallelizable C are shown using application programs rewritten in this evaluation.

SPEC2006 lbm: "lbm" has a pointer variables which points to the middle of an array and a pointer is updated in a loop. They are removed in the Parallelizable C version of the program. Pointer update in a loop like Fig. 2 (a) is rewritten with a flag variable and conditional branch as in Fig. 2 (b).

SPEC2006 hmmer: "hmmer" has a buffer reuse codes among several loop iterations as shown in Fig. 3 (a). This is modified by freeing and allocating heap for each iteration as shown in Fig. 3 (b).

6 Performance Evaluation on Multicore Processors

This section gives performance evaluation results of automatic parallelization by the OSCAR compiler on server, PC, and embedded multicores for the Parallelizable C programs.

```
static void main_serial_loop() {
    /* malloc the data structure used in P7Viterbi() */
    mx = CreatePlan7Matrix(1, hmm->M, 25, 0);
    for (idx = 0; idx < nsample; idx++)
    {
        ...
        score = P7Viterbi(..., mx);  /* main computation */
        ...
    }
    ...
}
static float P7Viterbi(char *dsq, int L, struct plan7_s *hmm, struct dpmatrix_s *mx) {
    ResizePlan7Matrix(mx, ...); /* reuse the data structure by realloc */
    ... /* scoring */
    return score;
}
```

(a) Original C

```
static void main_serial_loop() {
    for (idx = 0; idx < nsample; idx++)
    {
        ...
        score = P7Viterbi(...);  /* main computation */
        ...
    }
}
static float P7Viterbi(char *dsq, int L, struct plan7_s *hmm) {
    struct dpmatrix_s *mx;
    mx = AllocPlan7Matrix(...); /* malloc the data structure for each iteration*/
    ... /* score computation */
    FreePlan7Matrix(mx); /* free the data structure for each iteration*/
    return score;
}
```

(b) Parallelizable C

Fig. 3. Removing buffer reuse in "hmmer"

6.1 Evaluation Environment

Application programs rewritten in Parallelizable C are automatically parallelized by OSCAR parallelizing compiler. The OSCAR compiler generates parallel C source code with OSCAR API. The generated parallel programs with OSCAR API are compiled into executable binaries for the target architecture using a sequential compiler with an API analyzer or an OpenMP compiler.

In this evaluation, multicore systems such as IBM p5 550Q server having 8 cores, Intel Core i7 PC having 4 cores, and Renesas, Hitachi, Waseda University RP2 having 4 cores in SMP mode, are selected. Parameters in each environment is shown in Table 2.

6.2 Target Application Programs

As described in section 5, SPEC2000 "art", "equake", SPEC2006 "lbm", "hmmer", and MediaBench "mpeg2encode" are translated into Parallelizable C and automatically parallelized by the OSCAR compiler. The parallel processing performance by automatic parallelization for their original versions and Paralleliz-

Table 2. Evaluation environment

System	IBM p5 550Q (Server)	Intel Core i7 920 (PC)	Renesas/Hitachi/Waseda RP2 (Embedded)
CPU	Power5+ (1.5GHz × 2 × 4)	Nehalem (2.66GHz × 4)	SH-4A (600MHz × 4)
L1 D-Cache	32KB for 1 core	32KB for 1 core	16KB for 1 core
L1 I-Cache	64KB for 1 core	32KB for 1 core	16KB for 1 core
L2 cache	1.9MB for 2 cores	256KB for 1 core	
L3 cache	36MB for 2 cores	8MB for 4 cores	
Native Compiler	IBM XL C/C++ for AIX Compiler V10.1	Intel C/C++ Compiler verion 11.0	SH C Compiler + OSCAR API Parser
Compile Option	OSCAR: -O5 -qsmp=noauto Native: -O5 -qsmp=auto	OSCAR: -fast -openmp Native: -fast -parallel	
other features	SMT: disabled	Hyper-Threading: disabled Turbo Boost: disabled	

able C versions are evaluated. Although it is not rewritten in this paper, AAC encoder originallly written in the Parallelizable C is also evaluated. This AAC encoder program is developed by Renesas Electronics Corporation which is written without pointers and structures except for pointer arguments of functions.

6.3 Processing Performance by Automatic Parallelization

This section shows the processing performance of automatic parallelization by the OSCAR compiler for both original code and Parallelizable C code. Performance on IBM p5 550Q is shown in Fig. 4, Intel Core i7 is shown in Fig. 5, and Renesas/Hitachi/Waseda RP2 is shown in Fig. 6, respectively. In these figures, vertical axis represents application and code version, and horizontal axis represents speedup compared with sequential execution of original code using 1 core. Bars in each entry in vertical axis correspond to the number of processor cores used, 1PE, 2PEs, 4PEs, and 8PEs, from left to right, respectively.

IBM p5 550Q server with 8 cores: As shown in Fig. 4, automatic parallelization of original code by the OSCAR compiler achieves 4.96x speedup for "art", 5.61x speedup for "equake", respectively. By rewriting code into Parallelizable C, it achieves 5.35x speedup for "lbm", 6.06x speedup for "hmmer", and 5.12x speedup for "mpeg2encode", respectively, though in "lbm" and "hmmer" are no speedup. For "AACencode", it also achieves 6.16x speedup. To summarize these results, automatic parallelization of the Parallelizable C programs on 8 cores IBM p5 550Q by OSCAR compiler achieves 5.54x speedup against sequential execution in average.

Intel Core i7 PC with 4 cores: As shown in Fig. 5, automatic parallelization of original code by the OSCAR compiler achieves 1.51x speedup for "art", 1.45x speedup for "equake", 1.81x speedup against sequential execution, respectively. By rewriting code into Parallelizable C, it achieves 1.45x speedup for "equake", 1.28x speedup for "lbm", 3.34x speedup for "hmmer", 3.60x speedup for "mpeg2encode", respectively. For "AACencode", it also

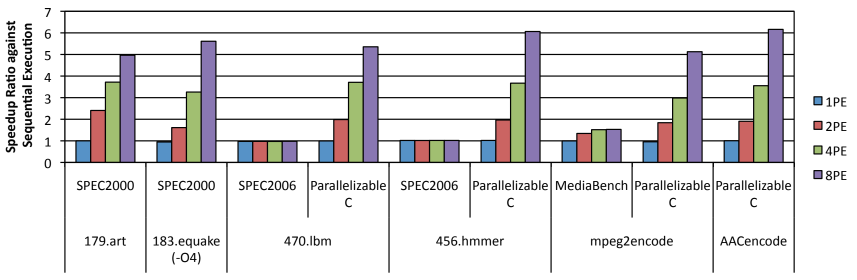

Fig. 4. Automatic parallelization results by OSCAR Compiler on IBM p5 550Q

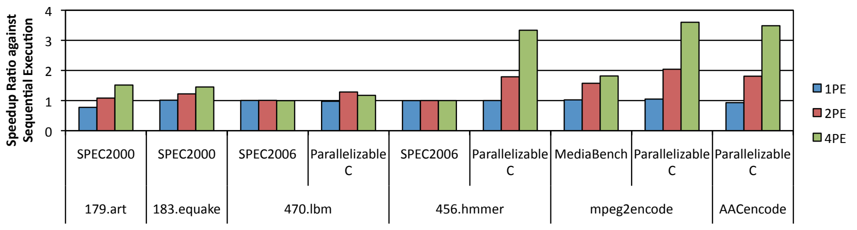

Fig. 5. Automatic parallelization results by OSCAR Compiler on Intel Core i7 920

achieves 3.48x speedup. To summarize these results, automatic parallelization of Parallelizable C programs by the OSCAR compiler achievs 2.34x speedup against sequential execution in average.

Renesas/Hitachi/Waseda RP2 in 4 cores SMP mode As shown in Fig. 6, automatic parallelization of original code by the OSCAR compiler achieves 2.54x speedup for "art" and 1.95x speedup for "equake" against sequential execution, respectively. By rewriting code into Parallelizable C, automatic parallelization achieves 2.19x speedup for "lbm", 3.47x speedup for "hmmer", and 3.27x speedup for "mpeg2encode", respectively. For "AACencode" it also achieved 3.35x speedup. To summarize results, automatic parallelization of Parallelizable C programs by the OSCAR compiler achieves 2.80x speedup against sequential execution in average.

6.4 Automatic Parallelization Results by OSCAR Compiler

This section shows the summary of automatic parallelization by the OSCAR compiler for translated application programs such as "art", "equake", "lbm", "hmmer", and "mpeg2encode".

SPEC2000 art: Automatic parallelization by the OSCAR compiler achieves 4.96x speedup on IBM p5 550Q using 8 cores. 95% of execution time is

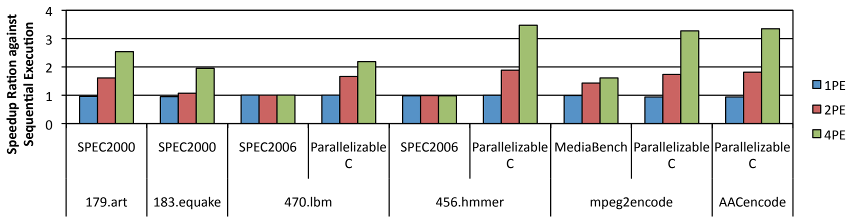

Fig. 6. Automatic parallelization results by OSCAR Compiler on Renesas/Hitachi/Waseda RP2

consumed in two functions, namely train_match() and match(). Each of them are consists of 7 parallelizable loops in major portion. The OSCAR compiler parallelized these loops to achieve speedup.

SPEC2000 equake: Automatic parallelization achieves 5.61x speedup on IBM p5 550Q using 8 cores. The original code of "equake" is compliant to Parallelizable C and can be parallelized by the compiler. Major computation part in a function smvp() consists of a reduction loop for whole array. That leads the speedup according to the number of cores.

SPEC2006 lbm: Code modification into the Parallelizable C achieves 5.36x speedup using 8 cores on IBM p5 550Q by automatic parallelization of the OSCAR compiler, while speedup for the original code is not obtained. "lbm" consists of 90% of execution time in a function LBM_performStreamCollide which has one parallelizable loop. In original code, a loop having the call statement for this function contains pointer updates (Fig. 2), that causes spurious points-to information for the arguments of the function. This ambiguity interferes the parallelism extraction. Thus, removing this pointer updates leads to the automatic parallelization to achieve speedup.

SPEC2006 hmmer: Code modification into the Parallelizable C achieved 6.06x speedup using 8 cores on IBM p5 550Q by automatic parallelization of OSCAR compiler, while speedup for the original code is not obtained. "hmmer" consists of 90% of execution time in a sequential loop in main_loop_serial(). On the loop body, there are some part which has dependency among loop iterations and some part which does not have dependency. A few kinds of loop restructuring, such as loop distribution and stripmining, is leveraged for automatic parallelization. After that, data localization in coarse grain task parallelization is applied to obtain speedup. In original code, the reuse of a heap buffer throughout iterations in a loop interferences the parallelism extraction.

MediaBench mpeg2encode: Code rewriting in Parallelizable C achieves 5.13x speedup using 8 cores on IBM p5 550Q by the OSCAR compiler, while speedup for original code is limited to 1.53x. In the Parallelizable C code, the OSCAR compiler extracted parallelism among macroblock data structures and applied data localization to improve performance.

7 Conclusion

This paper proposes the Parallelizable C, a coding gudeline for automatic parallelization of C programs by the OSCAR compiler. Automatic parallelization of numerical and multimedia application programs written in Parallelizable C is evaluated on multicore systems. The evaluation results show that automatic parallelization of Parallelizable C achieves, 5.54x sppedup on average using 8 cores on IBM p5 550Q server, 2.43x speedup on average using 4 cores on Intel Core i7, and 2.80x speedup on average using 4 cores on Renesas/Hitachi/Waseda RP2, against sequential execution, respectively.

The collabolation of Parallelizable C and an automatic parallelizing compiler with OSCAR API such as the OSCAR compiler improves software productivity and code portability for multicore processors, that also encourages the future research and development on multicore and manycore technology.

Acknowledgments

A part of this research was supported by METI/NEDO projects "Heterogeneous Multicore Technology for Realtime Consumer Electronics" and "Leading Research on Low Power Manycore Processor Systems". The authors specifically thanks to the members of "Realtime Consumer Electronics Multicore Architecture and API Comittee" in these projects, from Fujitsu Laboratory, Hitachi, NEC, Panansonic, Renesas Electronics, and Toshiba. A part of this research was supported by Waseda University global COE program "International Research and Education Center for Ambient SoC".

References

1. Eigenmann, R., Hoeflinger, J., Padua, D.: On the automatic parallelization of the perfect benchmarks. IEEE Trans. on parallel and distributed systems **9**(1) (Jan. 1998)
2. Hall, M.W., Anderson, J.M., Amarasinghe, S.P., Murphy, B.R., Liao, S., Bugnion, E., Lam, M.S.: Maximizing multiprocessor performance with the suif compiler. IEEE Computer (1996)
3. Kasahara, H., et al.: A multi-grain parallelizing compilation scheme on oscar. Proc. 4th Workshop on Language and Compilers for Parallel Computing (1991)
4. Hind, M.: Pointer analysis: Haven't we solved this problem yet? In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (2001) 54–61
5. Ryoo, S., Ueng, S.Z., Rodrigues, C.I., Kidd, R.E., Frank, M.I., Hwu, W.W.: Automatic discovery of coarse-grained parallelism in media applications. Transactions on High-Performance Embedded Architectures and Compilers (January 2007)
6. : MISRA-C:2004 - Guidelines for the use of the C language in critical systems. (2004)
7. : Clean C. http://www.imec.be/CleanC/.
8. Hwu, W., et al.: Implicitly parallel programming models for thousand-core microprocessors. DAC 2007 (2007)

9. Honda, H., Iwata, M., Kasahara, H.: Coarse grain parallelism detection scheme of a fortran program. Trans. of IEICE **J73-D-1**(12) (Dec. 1990) 951–960
10. Obata, M., Shirako, J., Kaminaga, H., Ishizaka, K., Kasahara, H.: Hierarchical parallelism control for multigrain parallel processing. In: Proc. of 15th International Workshop on Languages and Compilers for Parallel Computing. (Aug. 2002)
11. Yoshida, A., Koshizuka, K., Kasahara, H.: Data-localization for fortran macro-dataflow computation using partial static task assignment
12. Ishizaka, K., Obata, M., Kasahara, H.: Coarse grain task parallel processing with cache optimization on shared memory multiprocessor. In: Proc. of 14th International Workshop on Languages and Compilers for Parallel Computing. (Aug. 2001)
13. : Optimaly Scheduled Advanced Multiprocessor Application Program Interface (OSCAR API) version 1.0. http://www.kasahara.cs.waseda.ac.jp/.
14. Kimura, K., Mase, M., Mikami, H., Miyamoto, T., Shirako, J., Kasahara, H.: Oscar api for real-time low-power multicores and its performance on multicores and smp servers. In: Proc. of The 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC2009). (Oct. 2009)
15. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: PLDI'94. (1994)
16. Whaley, J., Lam, M.: Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. PLDI'04
17. Lattner, C., Lenharth, A., Adve, V.: Making context-sensitive points-to analysis with heap cloning practical for the real world. PLDI'07 (2007)
18. Kahlon, V.: Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. PLDI'08 (2008)
19. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. POPL'09 (2009)
20. Pearce, D.J., Kelly, P.H.J., Hankin, C.: Efficient field sensitive pointer analysis for c. PASTE '04 (2004)
21. Nystrom, E.M., Kim, H.S., Hwu, W.M.W.: Importance of heap specialization in pointer analysis. PASTE'04 (2004)
22. Ryoo, S., Rodrigues, C.I., Hwu, W.M.W.: Iteration disambiguation for parallelism identification in time-sliced applications. LCPC'07 (October 2007)
23. Wu, P., Feautrier, P., Padua, D.A., Sura, Z.: Instance-wise points-to analysis for loop-based dependence testing. ICS'02 (2002)
24. Lim, A.W., Liao, S., Lam, M.S.: Blocking and array contraction across arbitrarily nested loops using affine partitioning. Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (June 2001)
25. Kodaka, T., Nakano, H., Kimura, K., Kasahara, H.: Parallel processing using data localization for mpeg2 encoding on oscar chip multiprocessor. In: Proc. of International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems. (Jan. 2004)