Experiments with auto-parallelizing SPEC2000FP benchmarks

Guansong Zhang, Priya Unnikrishnan, James Ren

IBM Toronto Lab
Toronto
ON, L6G 1C7, Canada

Abstract. In this paper, we document the experimental work in our attempts to automatically parallelize SPEC2000FP benchmarks for SMP machines. This is not purely a research project. It was implemented within IBM's software laboratory in a commercial compiler infrastructure that implements OpenMP 2.0 specifications in both Fortran and C/C++. From the beginning, our emphasis is on using simple parallelization techniques. We aim to maintain a good trade-off between performance, especially scalability of an application program and its compilation time. Although the parallelization results show relatively low speed up, it is still promising considering the problems associated with explicit parallel programming and the fact that more and more multi-thread and multi-core chips will soon be available even for home computing.

Keywords: automatic parallelization, parallelizing compiler, SMT machine, OpenMP, parallel do

1 Introduction

Automatic parallelization or auto-parallelization involves automatically identifying and translating serial program code into equivalent parallel code to be executed by multiple threads. Both OpenMP parallelization and auto-parallelization try to exploit the benefits of shared memory, multiprocessor systems. But the main difference between OpenMP and auto-parallelization is that, for OpenMP, the user has complete knowledge of the program behavior and is aware of all the code segments that can benefit from parallelization. For auto-parallelization, however, the challenge is to pick the right parallelizable code from the limited information available within the compiler without any modification to the original source program.

1.1 Motivation

It is widely accepted that automatic parallelization is difficult and less efficient than explicit parallel programming. For years, people have been seeking better parallel programming models that can expressively describe the parallelism in an application. The existing models include, just to name a few, HPF[1], MPI[2], OpenMP[3], UPC[4], etc. It is also a known fact that parallel programming is difficult. No matter what kind of parallel programming model is used, creating efficient, scalable parallel code still requires

a significant degree of expertise. Parallelization tools can be used as an expert system to help users to find potential parallelizable code. However, to use the tools effectively, an understanding of dependence analysis and the application code is still needed.

On the other hand, more and more multi-thread and multi-core chips are becoming available in the market and parallel programming is no longer a privilege available only for high-performance computing. If not now, in the near future, parallel architecture will be a common feature in *desktop* and even *laptop computing*. Keeping this growing trend in mind, auto-parallelization looks like an irresistible option. Auto-parallelization relieves the user from having to analyze and modify the source program with explicit compiler directives. The user is shielded from low-level details of iteration space partitioning, data sharing, thread scheduling and synchronization. Auto-parallelization also saves the user the burden of ensuring correct parallel execution.

However, providing compiler support for automatic parallelization is not easy or straightforward. The biggest critique is that the resultant performance gain is typically limited, particularly in terms of scalability. In addition, accurate analysis can be restricted by the time constraints a compiler must meet for commercial acceptability, where in-depth analysis may be sacrificed to allow rapid operation. Taking these factors into account, we believe that using auto-parallelization is still justified by the following considerations:

- For parallel machines with a small number of node processors, scalability is not a big concern. While auto-parallelization may not give us the speedup that a massively parallel processing (MPP) application can achieve, it can still be used to take advantage of the extra silicon available.
- Even an MPP machine is most likely multi-layered where each node is a tightly coupled SMP machine, as in a cluster. Explicit message passing parallel programs, such as MPI, will still benefit from a nested automatic parallel execution.
- Most of the time-consuming applications are data parallel applications. With the growing problem size outpacing the improvements in hardware, data parallel applications can benefit from simple parallelization techniques.
- As more and more desktop parallel machines are available, users will be willing to try a parallel approach to solve their computational problems, if only for a small speedup as long as the effort to achieve that is reasonable.
- As hardware performance keeps improving, more complicated analysis will be tolerable in the compilation process.

1.2 Organization of the paper

The rest of the paper is organized as follows. In Section 2 we introduce our existing compilation and runtime environment and also briefly describe the structure of the auto-parallelizer. Section 3 and 4 describes in detail some optimization techniques used by the parallelizer to enhance the parallel performance of the application code. Section 5 looks at a few challenging parallelization cases that we encountered during the course of our work. Finally, in Section 6 we summarize our results and future work.

2 Parallelizer Structure and Position

The auto-parallelization work done here is based on the available OpenMP compilers, i.e. IBM®XL Fortran and VisualAge®C/C++. Our auto-parallelizer is essentially a loop parallelizer. For simplicity, the auto-parallelizer avoids complicated parallelization constructs and focuses on identifying and parallelizing expensive loops in the program. The parallelizer simply marks parallelizable loops as OpenMP `parallel do` constructs. The compiler then generates code similar to parallel loops marked by the user. Our auto-parallelizer does not support nested parallelization of loops unlike OpenMP. Auto-parallelization can also be done on an OpenMP program. In this case the auto-parallelizer skips the loop nests with OpenMP constructs and scans for other loops that can potentially benefit from parallelization.

2.1 Compilation and Runtime Infrastructure

A typical compiler and runtime structure to support OpenMP is shown in Figure 1 [5]. Figure 2 shows the Compile and Link phases of the compiler. *IR* is the intermediate representation. The parallelizer is delayed until the link phase of the compiler to make maximum utilization of the inter-procedural analysis available during linking. This is shown by the dashed lines in Figure 2

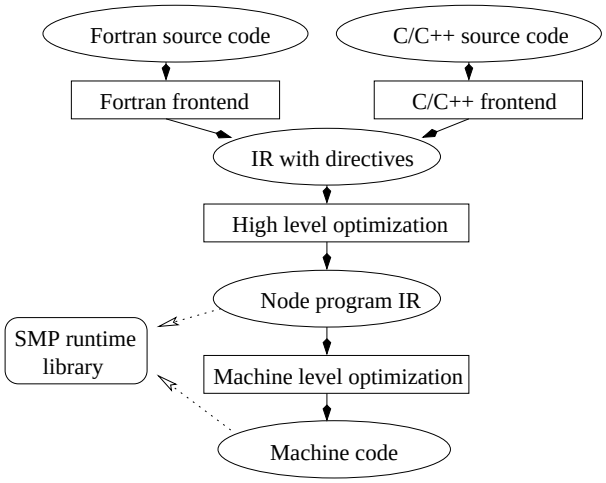

Fig. 1. Compile and runtime environment

SPEC2000 CPU benchmark suite was used to evaluate our techniques, as it is one of the best indicators of the performance of the processor, memory and compiler. Our tests focus on Spec2000FP, considering that it is where most of the data parallel processing exists. The hardware system used for testing in this paper is 1.1GHz POWER4™ system, with 1 to 8 nodes available.

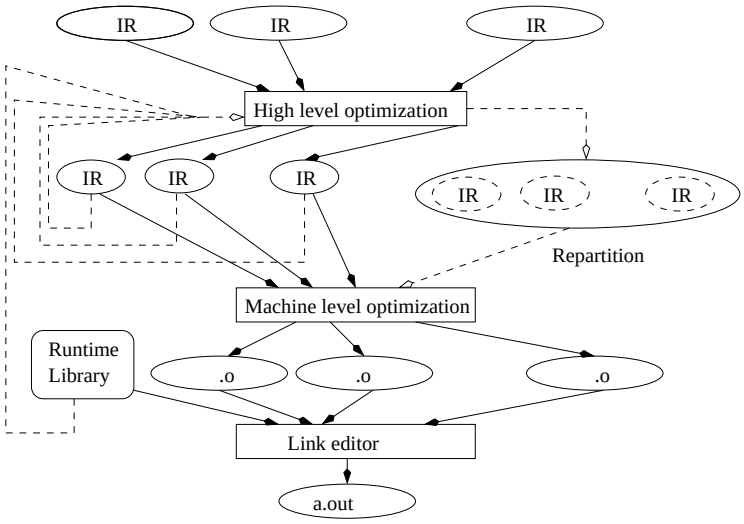

Fig. 2. Compile and link phase optimization

2.2 Basic Parallelizer Structure

Dependence analysis is a core part of the parallelizer. Data *dependence vectors*[6] are still our principal tools to analyze and verify the transformation applied on a loop nest. Suppose $\boldsymbol{\delta} = \{\delta_1 \dots \delta_n\}$ is a hybrid distance/direction vector with the most precise information derivable.

If we have

```
L1    DO i1 = 1, U1
L2      DO i2 = 1, U2
           ...
Ln         DO in = 1, Un
S1           A(f1(i1,...,in),...,fm(i1,...,in))=···
S2           ···=A(g1(i1,...,in),...,gm(i1,...,in))
T1           B(u1(i1,...,in),...,um(i1,...,in))=···
T2           ···=B(v1(i1,...,in),...,vm(i1,...,in))
           END DO
           ...
         END DO
       END DO
```

as a loop nest from L_1 to L_n, we may have two dependences $\boldsymbol{\delta}_A$ and $\boldsymbol{\delta}_B$ characterizing the nested loops, which were caused by S_1/S_2 and T_1/T_2, where f, g, u, v are linear functions of the loop induction variables. *Dependence matrix*, denoted as $\mathcal{D}$, includes $\boldsymbol{\delta}_A$ and $\boldsymbol{\delta}_B$ as two rows.

The Algorithm 1 shows the basic structure of the parallelizer. The parallelizer scans through all the loop nests in a procedure looking for parallelizable loops. Nested par-

Algorithm 1: Basic loop parallelizer

```
begin
    for each loop nest in a procedure do
        for each loop in the nest in the reverse depthfirst order (outer first) do
            if the loop is user parallel then
                break
            if the loop is marked sequential, has side-effects etc then
                continue
            if the loop has loop carried dependence then
                try splitting the loop to eliminate dependence
                if dependence not eliminated then
                    continue
            if loop cost is known at compile time then
                if the loop has not enough cost then
                    break
            else
                Insert code for run-time cost estimate
            Mark this loop auto parallel
            break
end
```

allelism is avoided by parallelizing only one loop in every nest. The loops in a nest are scanned from outer to inner as outer loops have a greater benefit from parallelization than inner loops. As a first step, loops that are not normalized and loops that are explicitly marked as sequential or have residuals, side-affects or with loop carried dependences are discarded by the parallelizer at compile time. For loops with loop carried dependences, the parallelizer tries to split the loop to eliminate the dependences. The loops are then parallelized independently. This preliminary step eliminates all non-parallelizable loops. However, naively executing all the parallelizable loops in an application may not be a good idea as we will soon discover. Parallelization is a very expensive operation with high overhead and the compiler has to be very judicial in identifying loops that can benefit from parallelization.

This paper omits discussions about basic techniques of computing dependence matrix, privatizing scalar variables in a loop, finding reduction variables, peeling or splitting the loop to eliminate the loop carried dependence, etc. which are all parts of the preparation phase for the parallelizer. We focus instead on the advanced techniques used by the parallelizer to intelligently select loops for parallelization and the optimizations that can be done after identifying a parallelizable loop. Section 3 deals with loop transformation techniques that can enhance the parallel performance of loops selected for parallelization. Section 4 explores the use of loop cost as an advanced loop selection technique for parallelization.

3 Balancing coarse-grained parallelism and locality

There are many loop transformation techniques that exists today[7]. And almost all of them can be applied to more than one kind of code optimizations. For example, loop interchange is widely used to improve data locality in cache management. Meanwhile, it also can be used to exploit parallelism, such as vectorization, and even coarse-grained parallelism. Unfortunately, the transformations for different optimization purposes do not always work in harmony.

For instance, in a loop nest, maximum spatial cache reuse can be achieved by moving the loop with stride-one access of the references to the innermost position. So a loop permutation algorithm will try to calculate such a loop order while keeping the dependence constraints of the original loop nest. On the other hand, in an automatic parallelizing compiler, one would like to parallelize the outermost loop to reduce the parallel region setup overhead, and leave more code in the inner loop for other optimization opportunities.

There has been significant research to study different approaches to solve this problem. A well known technique is *Unimodular transformation* [8–10], a compound loop transformation algorithm aiming at integrating different loop transformations for a specific goal and target machine, thereby reducing the problem of finding the best transformation combination to finding the best unimodular matrix. Another technique is *Loop selection*[7], which parallelizes all the parallelizable loops, then uses heuristics to select the next one as sequential, expecting to find more parallelizable loops after that. This is easier to implement when compared to unimodular transformation.

Our work is based on the idea of loop selection, but is different from the original one in two aspects: a) given our target is an OpenMP node program, we only need to find one parallelizable loop per nest, to avoid generating code with nested parallelism; b) our heuristics are directly linked to data locality, unlike the original loop selection technique which picks a loop with most "$<$" directions to parallelize.

3.1 Analyzing model

Using the notation in Section 2, we introduce two theorems.

Theorem 1 (C-level interchangeable). *Loop L_c and L_{c-1} is interchangeable if and only if $\forall \boldsymbol{\delta} \in \mathcal{D}$, $\boldsymbol{\delta} \neq (=^{(c-2)}<> \ldots)$, where "$=^{(c-2)}$" indicates that there are $(c-2)$ directions as "$=$" before the first "$<$".*

Theorem 1 and its variant forms can be found in [11] and other literatures. We will not prove it here, but use it directly for our theorem later.

Theorem 2 (P-level parallelizable). *Loop L_p can be parallelized as a parallel do if and only if $\forall \boldsymbol{\delta} \in \mathcal{D}$, $\boldsymbol{\delta} \neq (=^{(p-1)}< \ldots)$.*

The proof for Theorem 2 is trivial. By the definition of parallel do, it cannot carry any dependence.

Next, we define a *multiply* operation on a dependence vector. Let $\boldsymbol{\sigma} = \boldsymbol{\sigma}_a \times \boldsymbol{\sigma}_b$, where $\forall \sigma_j \in \boldsymbol{\sigma}, \sigma_j = \sigma_{a_j} \times \sigma_{b_j}$. The multiplication of the two dependence distances is

defined as following. It is unknown if one of the distances is unknown. Otherwise, it will be the regular multiplication on two integers. If one of the distances is only a direction, it is treated as 1 or -1, and after the multiplication, converted back to a direction .

Let $\boldsymbol{\sigma}_p$ and $\boldsymbol{\sigma}_{p-1}$ be the p^{th}, $(p-1)^{th}$ column vector of dependence matrix $\mathcal{D}$, then we have

Theorem 3 (Keeping it parallelizable). *A p-level parallelizable loop L_p can be interchanged to L_{p-1} and becomes (p-1)-level parallelizable if and only if $\forall \sigma_j \in \boldsymbol{\sigma}$, where $\boldsymbol{\sigma} = \boldsymbol{\sigma}_{p-1} \times \boldsymbol{\sigma}_p$, either*

- *$\sigma_j = 0$ or*
- *the j^{th} dependence in $\mathcal{D}$ is not carried by L_{p-1}*

Proof

We start by proving the "*if*" part of the theorem.

Suppose $\forall \sigma_j = 0$.

First, we can assert that L_{p-1} and L_p is interchangeable. Otherwise, according to Theorem 1 there is a dependence vector $\boldsymbol{\delta}$ that has the form of $(=^{(p-2)}<> \ldots)$. This will conflict with the fact that all σ_j is zero. Secondly, we prove that the L_p will become $(p-1)$-level parallelizable after interchanging with L_{p-1}. Otherwise, from Theorem 2, there will a dependence vector $\boldsymbol{\delta}$, of the format $(=^{(p-2)}< \ldots)$, in the dependence matrix preventing it from being parallelized after the interchange. Considering the format of the $\boldsymbol{\delta}$ before the interchange, given that all σ_j is zero, it should be in the form of $(=^{(p-1)}< \ldots)$. This conflicts with the fact that L_p is parallelizable according to Theorem 2.

If $\exists \sigma_j \neq 0$, we let $\boldsymbol{\delta}$ be the dependence vector on the j^{th} row of the matrix. It should have one of the following formats $(\cdots^{(p-2)} << \ldots)$, $(\cdots^{(p-2)} <> \ldots)$, $(\cdots^{(p-2)} >< \ldots)$, $(\cdots^{(p-2)} >> \ldots)$, where $\cdots^{(p-2)}$ is the leading $(p-2)$ positions. Since L_{p-1} does not carry any dependences as in the given condition, $\cdots^{(p-2)}$ has to be in the format of $(= \ldots =< \ldots)$, in order for the δ to be valid. Therefore, L_p can be interchanged with L_{p-1} and still be kept parallelizable.

Similarly, the "*only if*" part of the theorem can be proved. This is not important in our algorithm, so we will omit it here.

End of Proof

Theorem 4 (Outermost parallelizable). *Loop L_o can be parallelized at the outermost level if and only if $\boldsymbol{\sigma} = \mathbf{0}$, where $\boldsymbol{\sigma}$ is the o^{th} column vector of dependence matrix $\mathcal{D}$.*

This is a direct conclusion from Theorem 3. It is actually used in [7] for loop selection.

3.2 Loop permutation with examples

Based on our discussion in the previous section, we try to find a permutation favoring both data locality and parallelism. To do this, we first assign each loop induction variable with a weight in terms of memory access. We will favor those having maximum

memory spatial reuse. A more comprehensive way of evaluating the weights is called profitability-based methods, introduced in [7], which has a cache model to estimate cache misses. Either way, a memory ordering $O = (L_1, L_2, ..., L_n)$ of the nested loops is calculated, where L_1 has the least reuse and L_n the most.

Next we build up a nearby legal permutation in P by first testing to see if the loop L_1 is legal in the outermost position. If it is legal, it is added to P and removed from O. If it is not legal, the next loop in O is tested. Once a loop L_i is positioned, the process is repeated starting from the beginning of $O - \{L_i\}$ until O is empty. P is considered as having the best data locality for the loop nest. The algorithm is summarized in [12],

Finally, we can adjust the other loops further, based on the following conditions to decide if L_i can be moved before L_j,

- The interchange should be legal,
- L_i is still parallelizable after moving, and
- The amount of data locality we are willing to sacrifice.

The first two questions are answered by Theorem 3. We will use heuristics including the induction variable weights calculated previously to control how aggressive the parallelizer should be. Apart from estimating the cache miss, a good heuristic could also include estimations of the loop cost and parallel setup overhead. The topics deserve some discussions on their own, and will not be included here. *Strip-mining* can also be considered, if no loop can be moved at all.

We illustrate the benefits of using our loop permutation algorithm using a simple example. Loop permutation for data locality will transform the Fortran code shown below to `K` loop as the outermost and `I` loop as the innermost in the nest. This will lead to the middle `J` loop getting parallelized. With our loop permutation algorithm we can have the `J` loop moved to the outermost position, thus striking a balance between locality and the desired coarse-grained parallelism.

```
DO I = 1, N
  DO J = 1, N
    DO K = 1, N
      A(I,J,K)=A(I,J,K+1);
    END DO
  END DO
END DO
```

Figure 3 shows the performance difference of the generated code on the POWER4 system for both the cases discussed. `N` is set to 100 and the loop nest is executed 100 times. In the figure, "Baseline" and "Improved" use exactly the same compiler, with the only difference being that the parallel loop is in the middle for the "Baseline" case and the outermost for the "Improved" case. We can see that the parallel setup overhead completely offsets the gain from parallelization in the first case, while the improved version sees reasonable speedup even for a loop with small computation cost (the loop body has a single assignment statement).

In the SPEC2000FP suite, `mgrid` was negatively affected by this transformation, further experiments are being carried out to derive better heuristics.

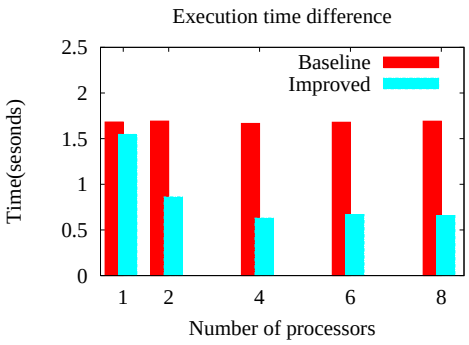

Fig. 3. Performance difference

4 Restricting parallelization using cost estimation

After a preliminary filtering out of loops unsuitable for parallelization using Algorithm 1, the auto-parallelizer evaluates the cost of a loop to further refine the selection of parallel loops.

4.1 Loop Cost Model

The cost of a loop is the approximate execution time of the loop and is given by

$$LoopCost = (IterationCount * ExecutionTimeOfLoopBody)$$

The loop cost algorithm is a recursive algorithm that computes the approximate execution time of the loop including all nested loops inside the current loop. To evaluate loops based on the loop cost, we define a set of *Threshold* values as the basis for comparison. The threshold values are carefully chosen values based on heuristics and take into account the number of processors, overhead arising from the setup required for parallelization, etc.

For loops whose cost can be estimated at compile-time, the parallelizer simply compares the cost against the precomputed threshold value and rejects loops with low costs. However for loops whose cost cannot be determined at compile-time (which is mostly the case), the auto-parallelizer computes a cost expression for the loop in terms of the values known only at runtime. This expression is then evaluated at runtime to determine whether the loop is worth parallelizing. The runtime cost expression is kept as lightweight as possible because of the overhead incurred in computing the expression at runtime. The auto-parallelizer also has a mechanism to limit the number of threads used to parallelize a loop depending on the cost of the loop. Table 1 shows the parallel loops selected at different stages of filtering.

Figure 4 shows the effectiveness of using the Loop Cost Model to restrict parallelization. The results show that careful restriction and appropriate selection of loops

for parallelization is extremely crucial for parallel performance. Evaluation of our auto-parallelization techniques for Spec2000FP benchmarks indicate that the auto-parallelizer is quite precise in selecting high-cost loops for parallelization as shown by Table 2. The Spec2000FP benchmarks were analyzed using Profile Directed Feedback techniques [13] [14] to identify the high-cost loops in the program. The loops selected by the auto-parallelizer are then compared with the list of expensive loops obtained from profile directed feedback. From Table 2 we can see that the auto-parallelizer is quite accurate within a small error margin. Column 4 of Table 2 indicates the number of loops incorrectly picked by the parallelizer, i.e., loops that have a low cost and hence are not worth parallelizing. It is important to keep this number a minimum as selecting the wrong loop can adversely impact the parallel execution performance. The zero values in Column 4 indicate that the parallelizer never picks the wrong loops.

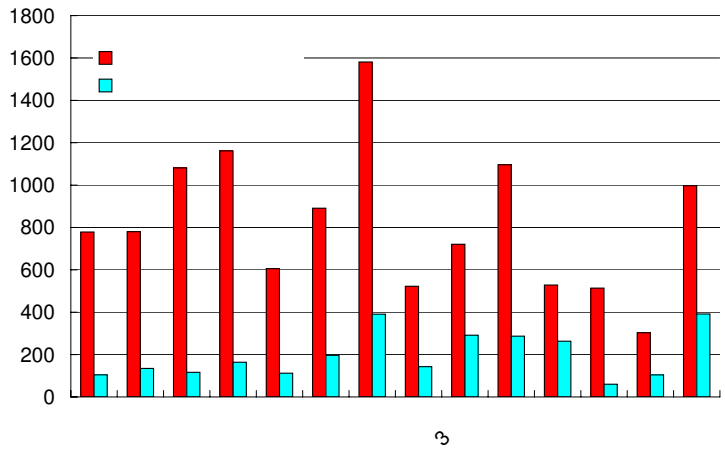

Fig. 4. Benchmark performance with and without Loop Cost using two CPU's

4.2 Runtime Profiling

In addition to using loop cost for restricting parallel loops, the auto-parallelizer employs runtime profiling[15] to further filter out loops at a finer granularity. The runtime profiling is a more accurate way of measuring the execution time of a chunk of code. The execution time of a chunk of a loop executed in parallel (as measured by the runtime profiler) is compared with selected *serial* and *parallel* thresholds. If the execution time of the chunk is less than the serial threshold, the next chunk of the loop is serialized dis-

Benchmark	#Input Loops	#Loops Processed	#After Stage1	#After Stage2/	#After Stage3
Stage1: Preliminary compile-time filtering Stage2: Compile-time loop cost filtering Stage3: Runtime loop cost filtering					
swim	23	18	9	9	5
wupwise	58	58	6	6	4
mgrid	47	30	11	11	7
applu	180	104	27	27	11
galgel	642	538	280	280	36
lucas	123	123	16	16	4
mesa	777	777	118	118	0
art	83	83	18	18	5
equake	62	64	0	0	0
ammp	283	282	18	18	0
apsi	303	284	92	92	5
sixtrack	1157	1135	347	347	11
fma3d	1648	1635	272	272	33
facerec	202	133	69	69	9

Table 1. Stages of Loop Selection for Auto-parallelization of Spec2000 FP Benchmarks

Benchmark	#HighCostLoops from PDF	#Parallelizable HighCostLoops from PDF	#HighCostLoops selected by Parallelizer	#LowCostLoops selected by Parallelizer
swim	8	5	5	0
wupwise	7	4	4	0
mgrid	9	7	7	0
applu	41	11	11	0
galgel	84	49	36	0
lucas	47	4	4	0
mesa	74	3	0	0
art	37	5	5	0
equake	8	0	0	0
ammp	31	0	0	0
apsi	28	11	5	0
sixtrack	44	13	11	0
fma3d	61	33	33	0
facerec	43	25	9	0

Table 2. Evaluation of Loop Selection for Auto-parallelization of Spec2000 FP Benchmarks

regarding the decision from the runtime loop cost evaluation. The decision to parallelize or serialize changes dynamically depending on previous executions of the loop.

5 Missed parallelization opportunities

As expected, the performance improvements from auto-parallelization are limited (refer to Section 6 for the actual data). To understand the auto-parallelization results, we compare our results with the SPECOMP benchmarks. For an SMP machine with a small number of processors, SPECOMP achieves relatively good performance and scalability. With this comparison, we hope to understand the reasons for the disparity in the results between explicit and auto parallelization and also expose opportunities missed by the auto-parallelizer.

Among the 14 SPEC2000FP benchmarks, 10 of them are also included in SPECOMP test suite. The two versions of the benchmarks were compared on a loop-to-loop basis. This analysis exposes limitations in the auto-parallelizer and also led to some very interesting observations. We list here some of the cases where the auto-parallelizer failed to parallelize a loop that was explicitly parallelized in the SPECOMP version. We believe that improvements to the auto-parallelizer to handle these cases can result in significant performance improvements for at least some of the benchmarks. To prove this, we try to manually parallelize loops that were explicitly parallelized in SPECOMP and compare the performance gain (manual parallelization is not possible in all cases).

5.1 Case 1: Loop body contains function calls

The auto-parallelizer fails to parallelize loops that have function calls in the loop body. Function calls could result in side-effects; the current auto-parallelizer is not capable of handling such loops. Shown here is a code snip found in *wupwise* where the loop body has function calls. *wupwise* has four such case, two in muldeo.f and two in muldoe.f. Manual parallelization of such loops shows good speedup. To manually parallelize the loops, array's AUX1 and AUX3 were privatized. The execution time is about 114 seconds with one thread, 61.8 seconds for two threads and 46.5 seconds with four threads, i.e., a 46% improvement with 2 threads and 59% improvement with 4 threads over using 1 thread. More complicated forms of this case exists in *apsi*, *galgel* and *applu*

```
      COMPLEX*16   AUX1(12),AUX3(12)
      ....

      DO 100 JKL = 0, N2 * N3 * N4 - 1
        L = MOD (JKL / (N2 * N3), N4) + 1
        LP=MOD(L,N4)+1
        K = MOD (JKL / N2, N3) + 1
        KP=MOD(K,N3)+1
        J = MOD (JKL, N2) + 1
        JP=MOD(J,N2)+1
        DO 100 I=(MOD(J+K+L,2)+1),N1,2
          IP=MOD(I,N1)+1
```

```
            CALL GAMMUL(1,0,X(1,(IP+1)/2,J,K,L),AUX1)
            CALL SU3MUL(U(1,1,1,I,J,K,L),'N',AUX1,AUX3)
            CALL ZCOPY(12,AUX3,1,RESULT(1,(I+1)/2,J,K,L),1)
100    CONTINUE
```

5.2 Case 2: Array privatization

Several loops in SPECOMP benchmarks have arrays that are explicitly marked as private, enabling the loops to be parallelized. Array privatization analysis is not yet implemented in our parallelizer preventing it from exploiting these opportunities. The code snip in Case 1 illustrates an example where array privatization is required in order to parallelize the loop. 3 such nested loops can be found in subroutine *rhs* of *applu*. Manual parallelization of such loops resulted in a 12% performance gain for *applu*. Similar situations exist in *galgel* and apsi

5.3 Case 3: Zero trip loops

Zero trip loops are loops in which the number of iterations calculated from the parameters of the loop is less than 1. There are 8 such loops in *apsi*, one of which is shown below. There exists loop carried dependence on the induction variable L, which the compiler tries to eliminate by rewriting the induction variable in terms of the loop parameters. However, in the case shown here, the parallelizer cannot identify L as an induction variable because of the possibility that the value of NX or NY is zero, thereby preventing the outer loop from getting parallelized. By manually parallelizing the outermost loop of such loops, the performance of *apsi* was improved by 3%.

```
      DO 30 K=1,NZTOP
        DO 20 J=1,NY
          DO 10 I=1,NX
            L=L+1
            DCDX(L)=-(UX(L)+UM(K))*DCDX(L)-(VY(L)+VM(K))*DCDY(L)
     *              +Q(L)
 10       CONTINUE
 20     CONTINUE
 30   CONTINUE
```

6 Summary and future work

We measured the performance of both the sequential and the auto-parallel versions of SPEC2000FP benchmarks. The results are shown in Figure 5[1].The figure shows three execution times for each benchmark,

- *Sequential* : sequential execution time of the benchmark code compiled by our compiler at the highest optimization level.

[1] The numbers shown here are based on a snap shot of our development compiler.

– *Parallel* : parallel (2 threads) execution time of the benchmark parallelized by the auto-parallelizer for SMP machines.
– *Improved* : parallel (2 threads) execution time with manual changes to the source code as described in Section 5

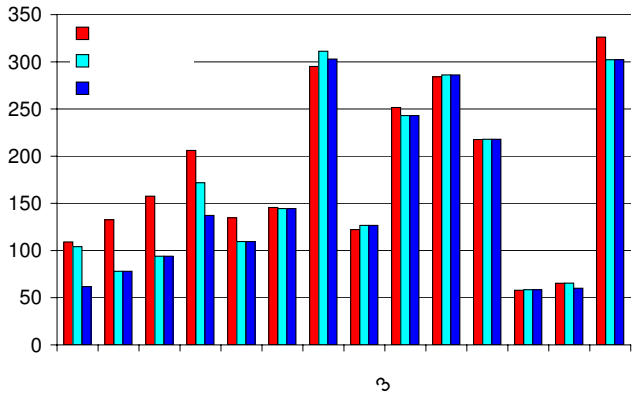

Fig. 5. Performance of auto-parallelization

We also emphasize that for simplicity, the modification to individual benchmarks for the *Improved* results address only one specific problem among the 3 identified in Section 5. For instance, `apsi` is affected by zero trip loops, function calls in the loop body as well as array privatization. However, the source modification includes only source changes for zero trip loops. Among those 10 benchmarks in the SPECOMP test suite, `swim`, `mgrid`, `applu`, `galgel` and `wupwise` see reasonable speedup, with improvement up to 40%.

The results from auto-parallelization as shown by Figure 5 are not very impressive, especially considering that the hardware resource has actually doubled. However, we argue that the focus in this paper has been the utilization of simple techniques to achieve parallelization with minimal impact on the compilation time. This paper has also shown that much better performance is obtainable using sophisticated techniques at the cost of increased compilation time. In addition, this paper also presents a simple loop permutation algorithm to balance data locality and coarse-grained parallelism on SMP machines.

Our auto-parallelizer has several limitations as shown in Section 5. Implementing array privatization analysis and improving the existing dependence analysis are part of

the planned future work. We also plan to fine-tune several heuristics and guidelines used in the parallelizer. With all these improvements, we hope that the auto-parallelizer will be able to detect inherent parallelism in the application code on par with explicit parallelization if not better.

7 Trademarks and copyright

References

1. Charles H. Koelbel, David B. Loveman, and Robert S. Schreiber. *The High Performance Fortran Handbook*. MIT Press, 1993.
2. Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1996.
3. Rohit Chandra et al. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers, 2001.
4. Tarek A. El-Ghazawi, William W. Carlson, and Jesse M. Draper. Upc language specification (v 1.1.1), 2003. http://upc.gwu.edu.
5. Guansong Zhang, Raul Silvera, and Roch Archambault. Structure and algorithm for implementing OpenMP workshare. In *WOMPAT*, Lecture Notes in Computer Science. Springer, 2004.
6. U. Banerjee. *Dependence Analysis for Supercomputing*. Boston MA: Kluwer, 1988.
7. Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures*. Morgan Kaufmann Publishers, 2002.
8. Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, 1991.
9. Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, 1991.
10. U. Banerjee. Unimodular transformations of double loops. In *Proc. of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.
11. Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-wesley, 1990.
12. K.S.Mckinley, S.Carr, and C.-W Tseng. Improving data locality with loop transformations. *ACM Trans. on Programming Language and Systems*, 18(4), 1996.
13. Robert Cohn and P. Geoffrey Lowney. Feedback directed optimization in Compaq's compilation tools for Alpha. 2nd ACM Workshop on Feedback-Directed Optimization, 1999.
14. W. J. Schmidt et. al. Profile-directed restructuring of operating system code. In *IBM Systems Journal, 37(2)*, 1998.
15. Sagnik Nandy, Xiaofeng Gao, and Jeanne Ferrante. TFP: Time-sensitive, Flow-specific Profiling at Runtime. In *LCPC 2003*, 16th Workshop on Languages and Compilers for Parallel Computing, 2003.

[2] The opinions expressed in this paper are those of the authors and not necessarily of IBM.