Experiments with auto-parallelizing SPEC2000FP benchmarks

Guansong Zhang

CASCON 2004

Authors

Guansong Zhang, Priya Unnikrishnan, James Ren

Compiler Development Team

IBM Toronto Lab, Canada

{guansong,priyau,jamesren}@ca.ibm.com

(most slides from Priya's LCPC2004 presentation)

Overview

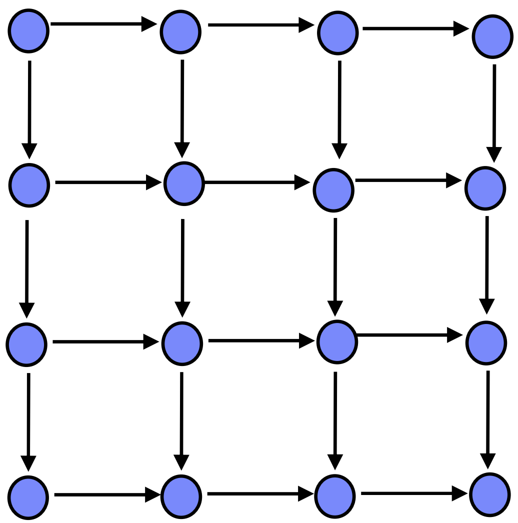
- Introduction and motivation
- Structure of our auto-parallelizer
- Performance results
- Limitations and future work

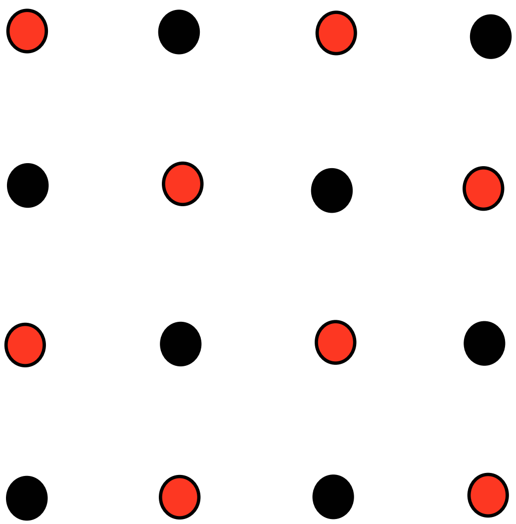
Auto parallelization, again?

- HPF (Fortran D)
  - V 1.0 1992; V 2.0 1997
- MPI
  - V 1.0, 1994; V 2.0 2002
- OpenMP
  - V 1.0, 1997; V 2.0 2000; V 2.5 2005
- Other parallel programming tools/models
  - Global array (1994), HPJava(1998), UPC(2001),
- Auto parallelization tools
  - ParaWise (CAPtools, 2000), Other efforts: Polaris, SUIF, …

The most effective way in parallelization

Discover parallelism in the algorithm





So, why?

- User expertise required
  - Knowledge of parallel programming, dependence analysis
  - Knowledge of the application, time and effort
- Extra cycles in desktop, even laptop computing
  - hyperthread
  - multicore

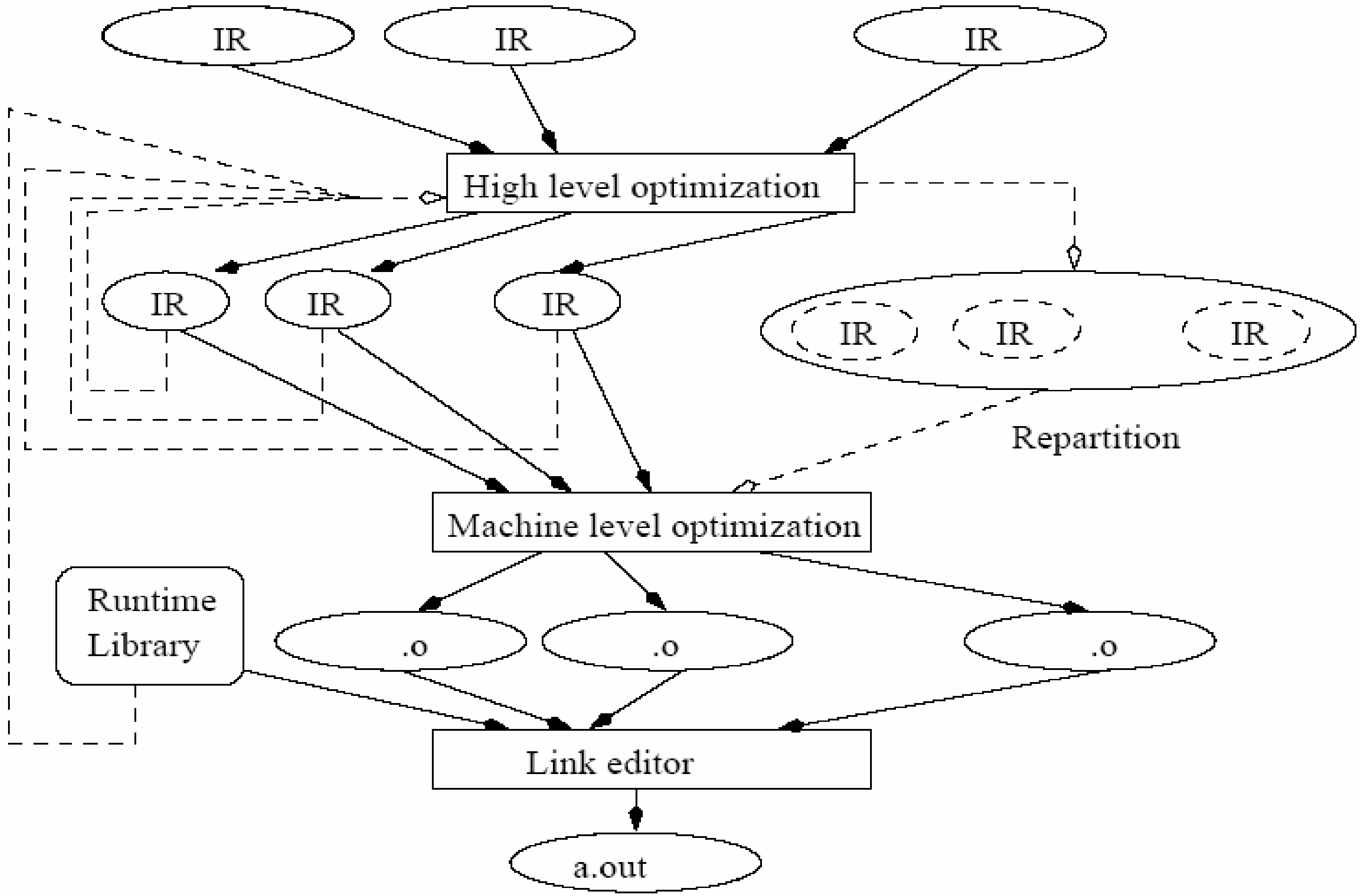
SPEC, from CPU2000 to OMP

- Parallel programming is difficult.
  - Even for just getting it right
- Parallelizable problem exist
  - Amdahl's Law vs. Gustafson's Law
  - 10 of the14 SPEC CPU FP tests are in SPECOMPM
  - 9 of the 11 SPECOMPM tests are in SPECOMPL

Strategy

- Make parallelism accessible to general users
  – Shields users from low-level details
- Take advantage of extra hardware
  – Do not waste the cycle and the power
- Our emphasis
  – Use simple parallelization techniques
  – Balance performance gain and compilation time

Compiler infrastructure



Our auto-parallelizer features

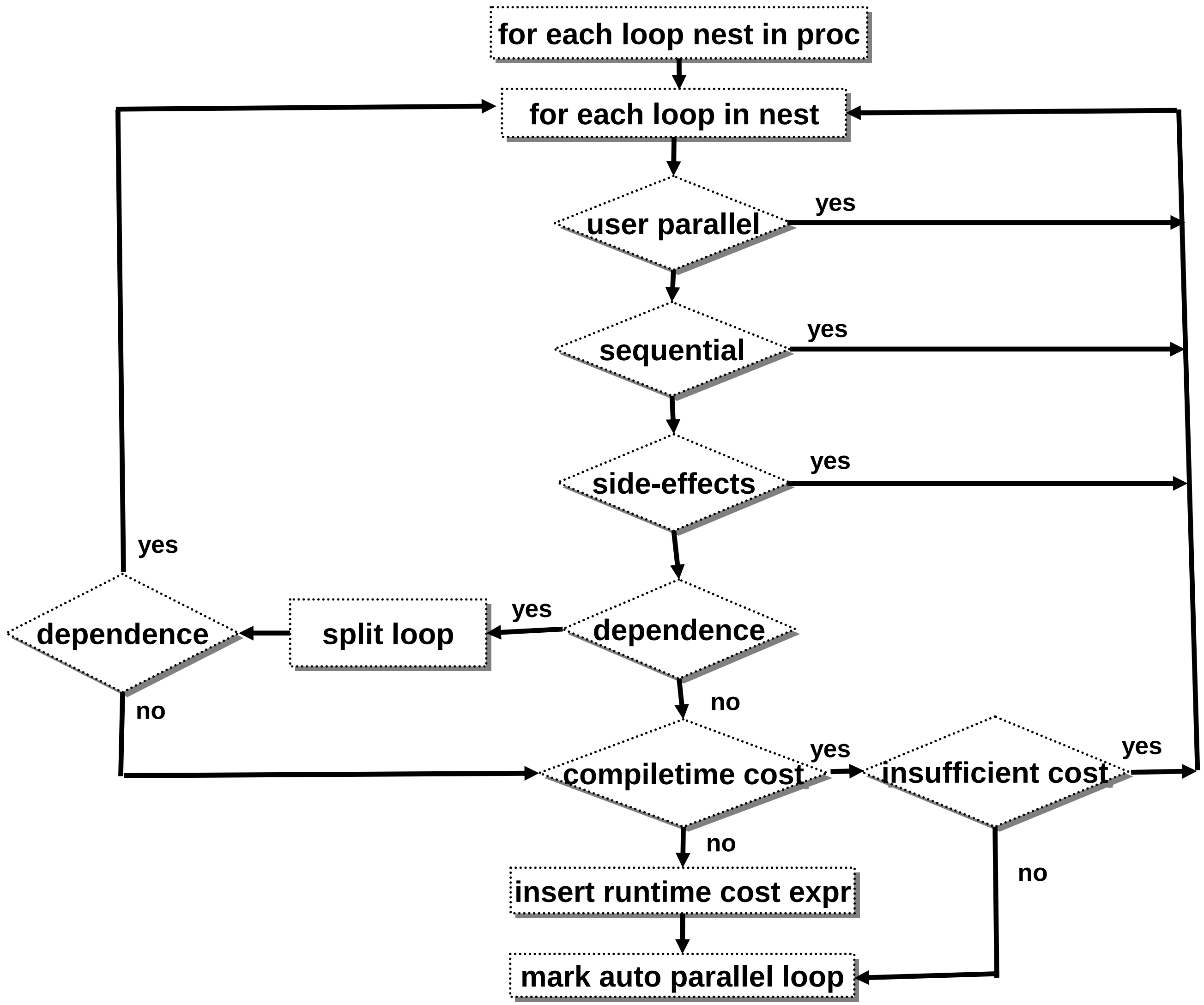
- Use OpenMP compiler and runtime infrastructure
  - for parallelization and parallel execution.
- Essentially a Loop parallelizer,
  - inserting “Parallel do” directives
- Can further optimize an OpenMP programs
  - general optimization specific to an OpenMP program
- Depending on dependence analyzer
  - core of the parallelizer, shared with other loop optimizations

Pre-parallelization Phase

- Induction variable elimination
- Scalar Privatization
- Reduction finding
- Loop transformations favoring parallelism

Basic loop parallelizer

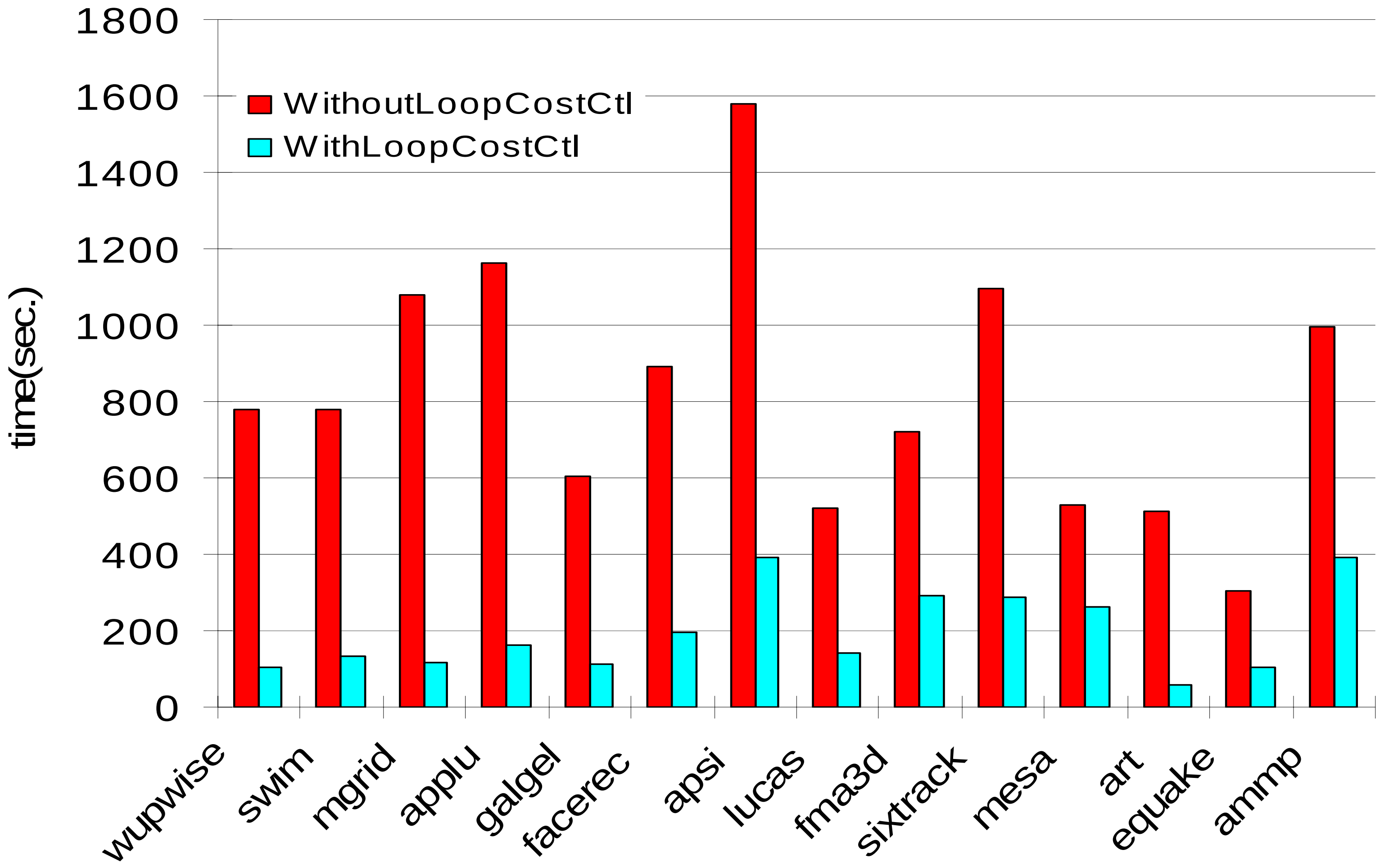
```
begin
  for each loop nest in a procedure do
    for each loop in the nest in the depthfirst order (outer first) do
      if the loop is user parallel then
        break
      if the loop is marked sequential, has side-effects etc then
        continue
      if the loop has loop carried dependence 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
```


for each loop nest in proc
for each loop in nest
user parallel
yes
sequential
yes
side-effects
yes
dependence
yes
split loop
dependence
yes
no
no
compiletime cost
yes
insufficient cost
yes
no
insert runtime cost expr
no
mark auto parallel loop

Loop Cost

- LoopCost = (Iterationcount * ExecTimeOfLoopBody)
- Compile time cost
- (LoopCost < Threshold) ← reject
- Runtime loop cost expression – extremely light-weight
- Runtime profiling – finer granularity filtering

Impact of Loop cost on performance



Accuracy of loop cost algorithm

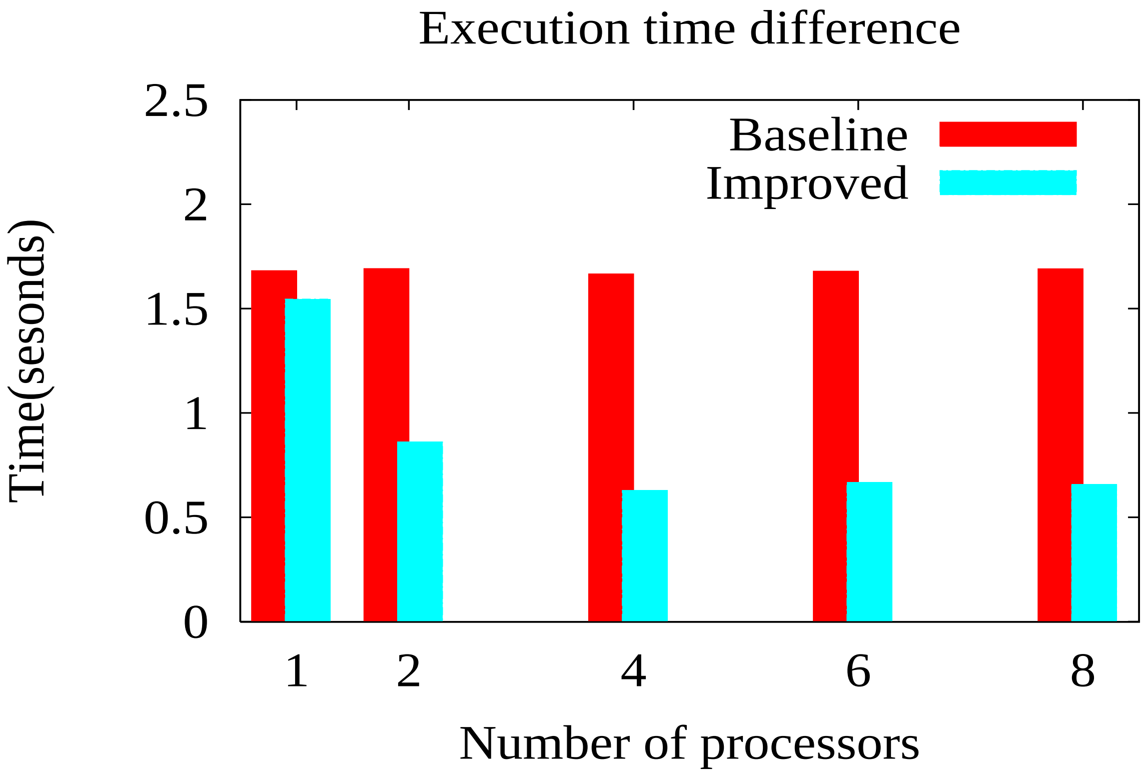
Benchmark	#Parallelizable HighCostLoops from PDF	#HighCostLoops selected by Parallelizer	#LowCostLoops selected by Parallelizer
swim	5	5	0
mgrid	7	7	0
applu	11	11	0
galgel	49	36	0
sixtrack	13	11	0
fma3d	33	33	0

Balance coarse-grained parallelism and locality

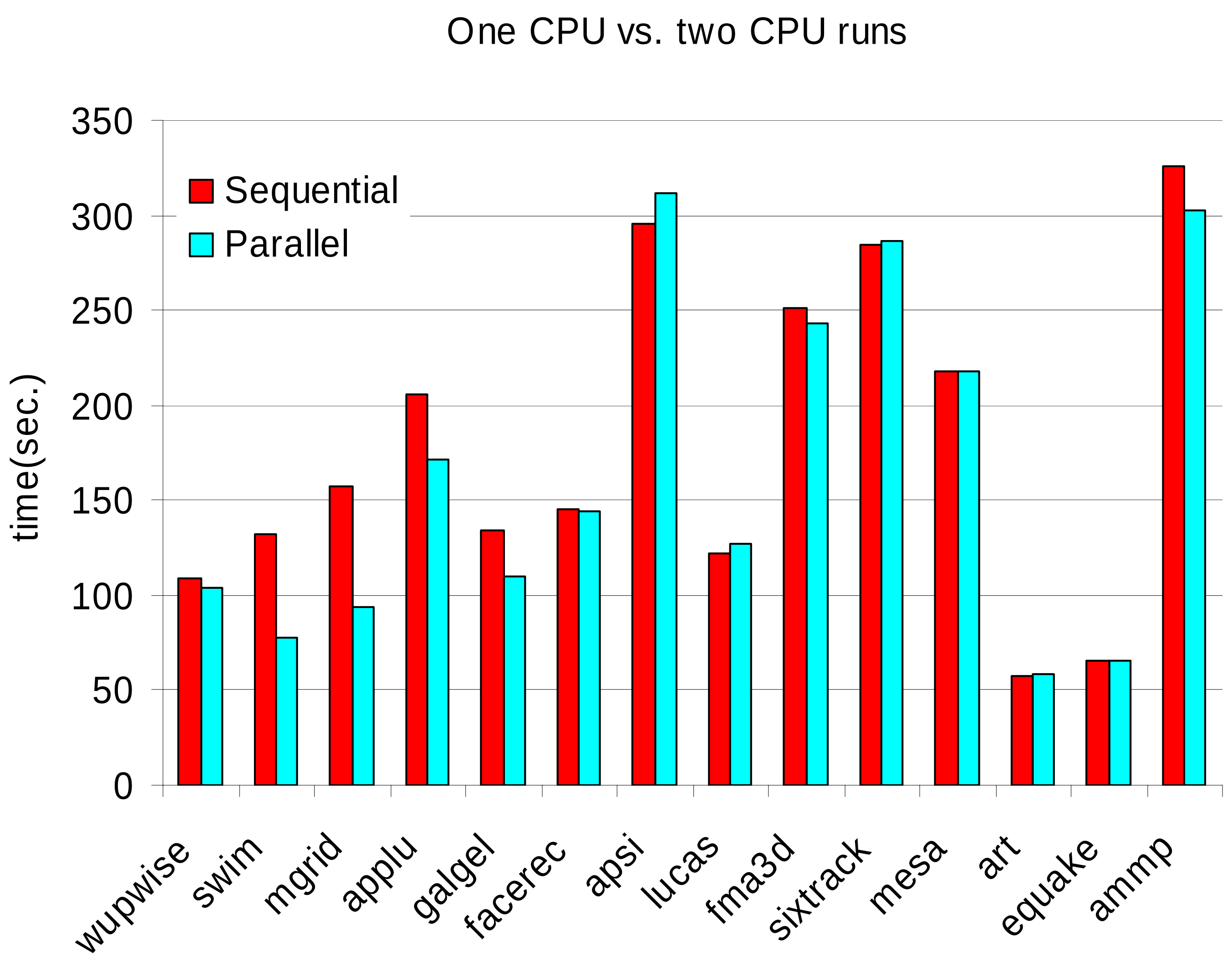
- Loop interchange for data locality
- Loop interchange to exploit parallelism
- Transformations do not always work in harmony

```
DO I = 1, N          ← FORTRAN loop nest
   DO J = 1, N
      DO K = 1, N
         A( I, J, K) = A( I, J, K+1)
      END DO
   END DO
END DO
```

Performance: loop permutation for parallelism



Auto-parallelization performance – 10%



Expose limitations

- Compare SPEC2000FP and SPECOMP
- SPECOMP achieves good performance and scalability
  - – Disparity between explicit and auto-parallelization
- Expose missed opportunities
- 10 common benchmarks
  - – Compare on a loop-to-loop basis

Limitations

- Loop body contains function calls
- Array privatization

```
        COMPLEX*16   AUX1(12),AUX3(12)
        ....
        DO 100 JKL = 0, N2 * N3 * N4 - 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 (1,1,1,I,J,K,L),'N',AUX1,AUX3)
        ....
100     CONTINUE
```

Limitations … contd

- Zero trip loops

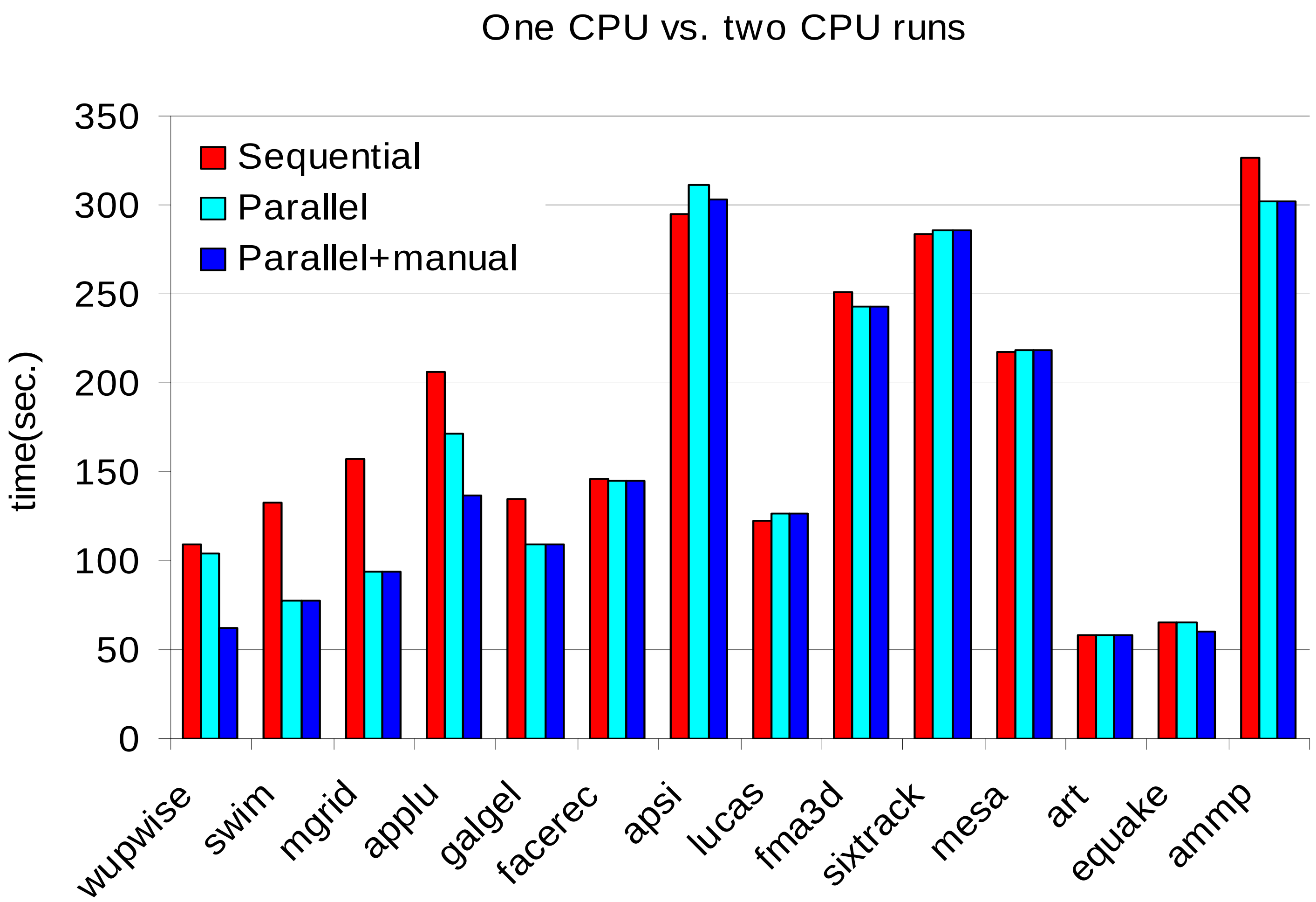
```
IV=0
DO J=1, M
  DO I=1,N
    IV=IV+1
    A(IV) = 0
  ENDDO
ENDDO
```

  – Induction variable 'IV=I+(J-1)*N'

  – Valid if N is positive.

  – Cannot parallelize outer-loop if N is zero

Improved auto-parallelization performance



System Configuration

- SPEC2000 CPU FP benchmark suite
- IBM XL Fortran/C/C++ commercial compiler infrastructure which implements OpenMP 2.0
- Hardware : 1.1GHz POWER4 with 1-8 nodes
- Compiler options: -O5 –qsmp
  – Comparing to –O5 as sequential

Future Work

- Fine tune the heuristics
  - Loop cost, permutation, unroll.
- Further loop parallelization
  - Array dataflow analysis, array privatization
  - Do across, loop with carried dependence
  - Interprocedural, runtime dependence analysis
- Speculative execution
  - OpenMP threadprivate, sections, task queue
- Keep reasonable increase in compilation time
  - not to compete with auto-par tools in the near future