



OpenMP Benchmark using PARKBENCH

Mitsuhisa Sato

Kazuhiro Kusano and Shigehisa Satoh
Real World Computing Partnership, Japan



OpenMP Benchmark

- ◆ **To understand the performance of OpenMP program, the following factors are important**
 - ◆ **The overhead of OpenMP constructs**
 - EPCC microbenchmark
 - ◆ **The performance of the target SMP's shared memory**
 - PARKBENCH for OpenMP
- ◆ **Give guidance to the programmer:**
 - ◆ How much performance to be obtained on shared memory hardware for typical arithmetic loops.
 - Mainly for parallel loops, not test full range of OpenMP directives.
 - ◆ The overheads of OpenMP loop constructs are evaluated with respect to the loop performance.
 - Overhead = The half-performance length



Outline

- ◆ **Introduction and Motivation**
 - ◆ OpenMP benchmark
- ◆ **PARKBENCH for OpenMP**
 - ◆ RINF1: arithmetic operations
 - ◆ POLY1: memory and cache bottleneck
- ◆ **Results**
- ◆ **Conclusion**



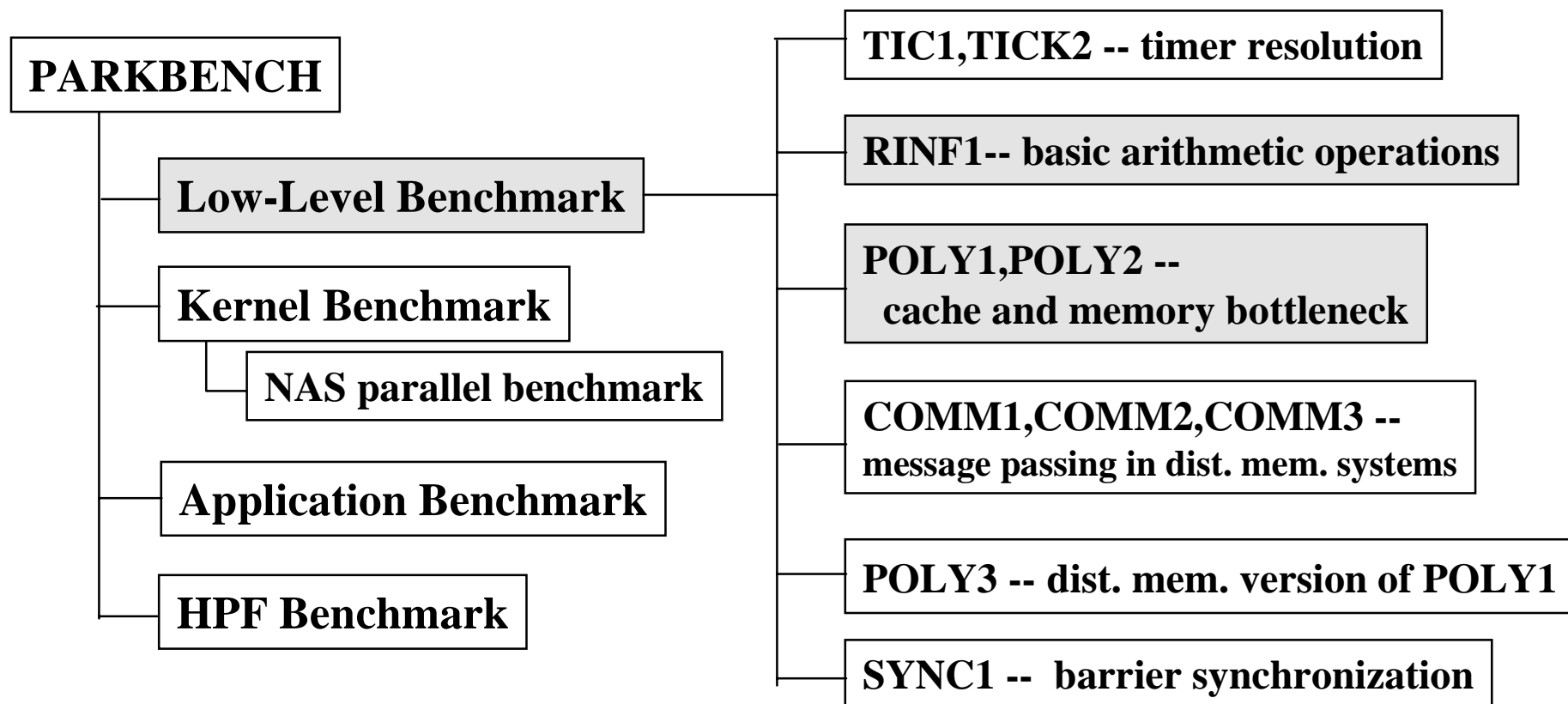
OpenMP benchmark

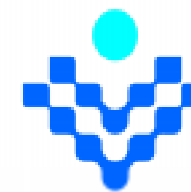
- ◆ **Synthetic benchmark**
 - ◆ EPC microbenchmark
 - ◆ Low level benchmark of PARKBENCH
- ◆ **Kernel benchmark**
 - ◆ NAS parallel benchmark in OpenMP (NPB 3.0?)
- ◆ **Application benchmark**
 - ◆ SPEC HPC benchmark (SPEC_{hpc}96)
 - ◆ SPEC OpenMP benchmark (coming soon?)



PARKBENCH

- ◆ A set of benchmark programs proposed by PARKBENCH committee, for distributed memory systems
- ◆ Use Low-level Benchmarks as an OpenMP benchmark





RINF1: Arithmetic operations

No	Operation	Flops	Comment
1	DYADS, $A(I)=B(I)*C(I)$	1.0	contiguous
2	DYADS, $A(I)=B(I)*C(I)$	1.0	stride=8
3	TRIADS, $A(I)=B(I)*C(I)+D(I)$	2.0	Contiguous
4	TRIADS, $A(I)=B(I)*C(I)+D(I)$	2.0	stride=8
5	Random Scatter/Gather	2.0	
6	$A(I)=B(I)*C(I)+D(I)*E(I)+F(I)$	4.0	contiguous
7	Inner Product, $S=S+B(I)*C(I)$	2.0	single, reduction
8	First Order Recurrence	2.0	not parallelized
9	Charge Assignment; $A(J(I))=A(J(I))+S$	1.0	atomic
10	Transposition: $B(I,J)=A(J,I)$	N	
11	Matrix Mult BY Inner Product	$2*N*N$	
12	Matrix Mult BY Middle Product	$2*N*N$	
13	Matrix Mult BY Outer Product	$2*N$	
14	DYADS, $A(I)=B(I)*C(I)$	1.0	stride=128
15	DYADS, $A(I)=B(I)*C(I)$	1.0	stride=1024
16	DAXPY, $A(I)=S*B(I)+C(I)$	2.0	contiguous
17	DAXPY, $A(J(I))=S*B(K(I))+C(L(I))$	2.0	Indirect
18	DAXPY, $A(I)=S*B(I)+C(I)$	2.0	contiguous, parallel do
19	DAXPY, $A(I)=S*B(I)+C(I)$	2.0	contiguous, schedule(static,10)
20	DAXPY, $A(I)=S*B(I)+C(I)$	2.0	contiguous, schedule(dynamic,10)

Extended for OpenMP



RINF1: Metrics

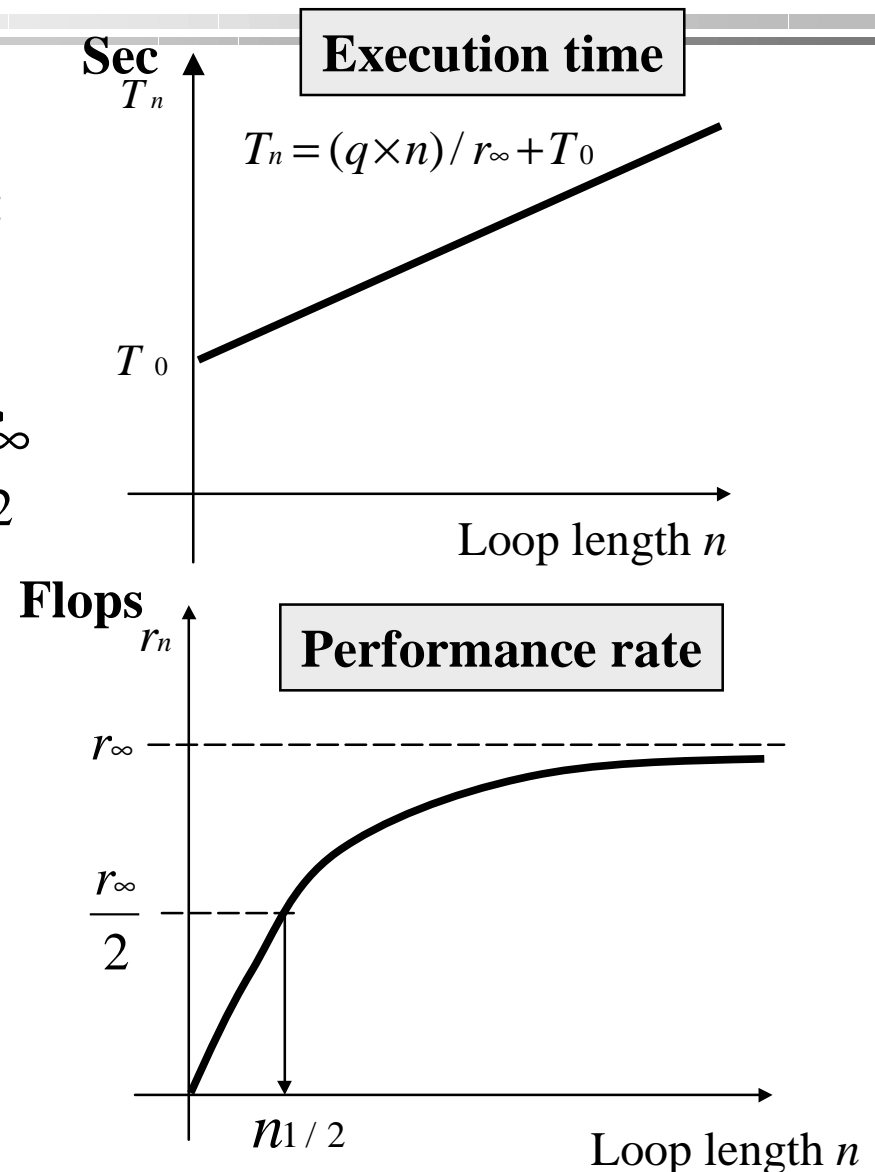
- ◆ The execution time of the loop (length= n , q floating point operations) is approximated by:

$$T_n = (q \times n) / r_\infty + T_0$$

- ◆ Asymptotic performance rate: r_∞
- ◆ Half-performance length: $n_{1/2}$
- ◆ Loop startup overhead T_0 is represented by $n_{1/2}$, which is the quantity known to the programmer.

$$T_0 = \frac{n_{1/2} \times q}{r_\infty}$$

- ◆ T_0 includes the OpenMP loop scheduling overhead.

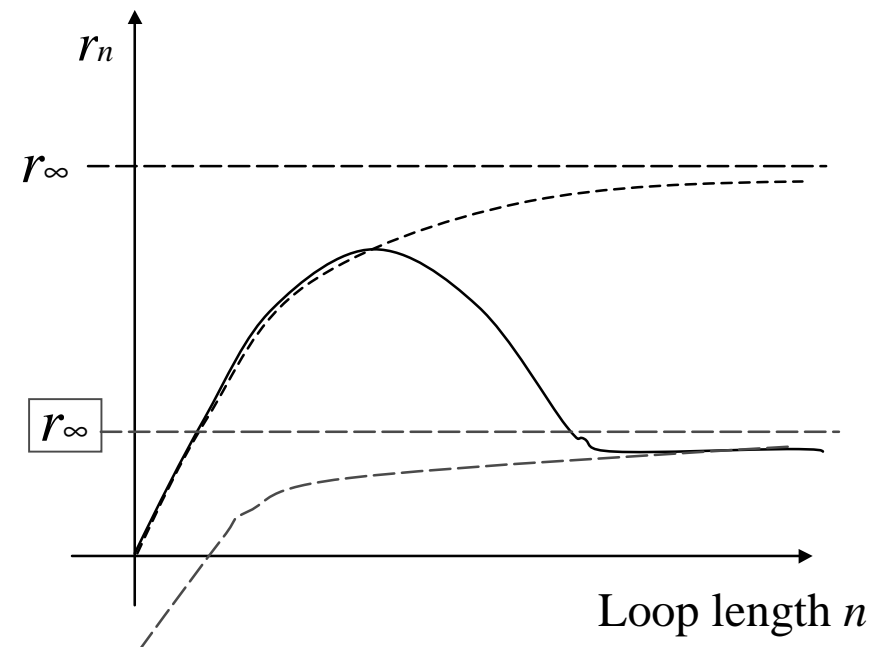
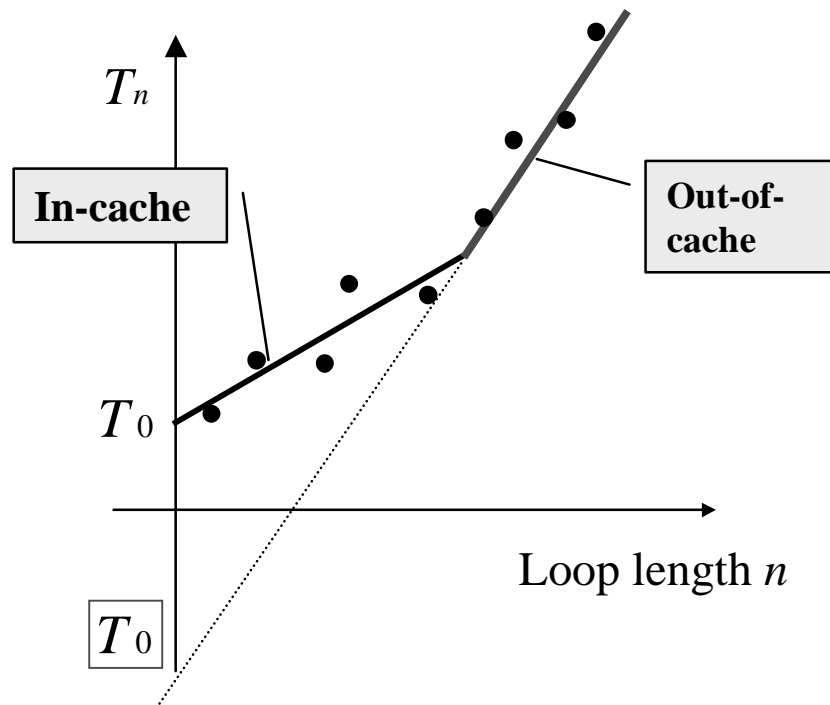




RINF1: cache effect

- ◆ Two sets of r_∞ and $n_{1/2}$ are reported for both in-cache and out-of-cache cases.
- ◆ The parameters are determined by a least square fit of the data to the straight line defined by:
 - ◆ If the error become large, restart the fitting.

$$r_n = \frac{r_\infty}{1 + n_{1/2}/n}$$





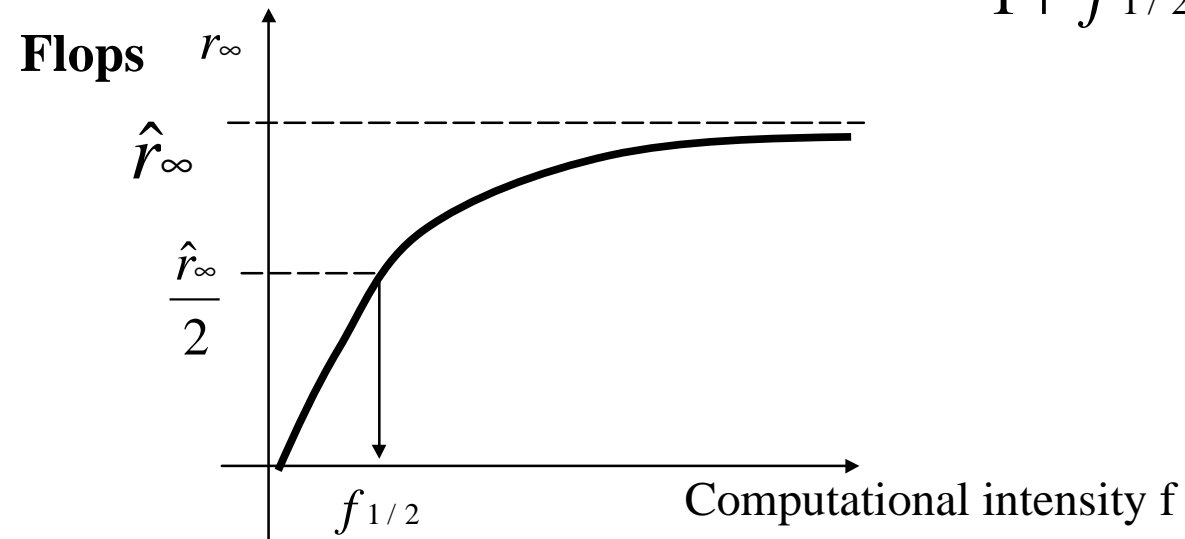
POLY1,2: Metrics

- ◆ Measure the basic hardware performance on shared memory bottleneck.
- ◆ Computational intensity f is defined by the number of floating point operations per memory reference.
- ◆ The asymptotic performance rate r_∞ is given by:

- ◆ The peak performance of arithmetic unit: \hat{r}_∞

- ◆ The half computational intensity: $f_{1/2}$

$$r_\infty = \frac{\hat{r}_\infty}{1 + f_{1/2}/f}$$





POLY1,2: Metrics

- ◆ **Use the evaluation of polynomials by Horner's rule.**
 - ◆ To control the computational intensity
 - ◆ POLY1 for in-cache, loop length=10,000
 - ◆ POLY2 for out-of-cache, loop length=100,000
- ◆ **If memory access and arithmetic operation are not overlapped, $f_{1/2}$ shows the ratio of arithmetic speed to the memory speed.**



Performance measurement

- ◆ Each kernel loops are parallelized by OpenMP directives.
- ◆ The function `DWALLTIME00` returns the wall clock time.
- ◆ The execution time is computed by:

$$T = T2 - T1 - T0$$

- ◆ `T0` is the execution time of dummy loop
- ◆ `DUMMY` is inserted to prevent optimizations.
- ◆ Change the number of threads (processors)

```
T1 = DWALLTIME00()  
!$OMP parallel private(JT)  
  DO JT=1,NTIM  
    CALL DUMMY(JT)  
!$OMP do  
  DO I=1,N  
    ...vector computation ...  
  END DO  
!$OMP end do  
END DO  
!$OMP end parallel  
T2 = DWALLTIME00()
```



Platforms

◆ **Omni-Linux**

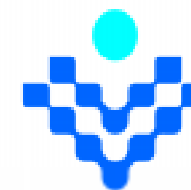
- ◆ **COMPAQ ProLiant 6500(Intel Pentium II Xeon 450MHz*4CPU, 1MB cache/CPU, 1GB memory)**
- ◆ **Linux Red-hat 6.0**
- ◆ **Omni OpenMP Compiler (backend:egcs 2.91.66, option -O3 -malign-double)**

◆ **PGI-Linux**

- ◆ **COMPAQ ProLiant 6500**
- ◆ **Linux Red-hat 6.0**
- ◆ **PGI OpenMP Fortran Compiler(pgf 3.1.2)**

◆ **Omni-Sun**

- ◆ **Sun Enterprise 450(UltraSparc 300MHz*4CPU,1GB memory)**
- ◆ **Solaris 2.6**
- ◆ **Omni OpenMP Compiler(backend: SUN Wspro 4.2, options: -fast)**



RINF1 summary

- ◆ Change the number of threads (processors).
- ◆ As a summary of RINF1, two sets of r_∞ and $n_{1/2}$ are reported for each kernel loop.

In-cache

Out-of-cache

No.	#P	vlen(1)	r_∞	$n_{1/2}$	vlen(2)	r_∞	$n_{1/2}$	r_n min	r_n max
1	seq	<= 400	52.95	4.8	>= 800	9.74	-4414.7	9.64	53.67
	2	<= 8000	53.02	35.5	>= 30000	15.41	-15515.9	5.96	58.41
	4	<= 8000	81.03	0.1	>= 30000	18.38	-21211.0	4.31	97.46
2	seq	<= 50	49.53	1.4	>= 90	3.83	-554.4	3.58	48.95
	2	<= 90	63.42	92.7	>= 400	6.93	-879.4	6.17	33.33
	4	<= 1000	35.75	18.2	>= 5000	15.12	-2910.6	4.81	61.32
3	seq	<= 80	80.56	0.8	>= 300	15.27	-2981.9	15.26	87.98
	2	<= 800	114.20	79.9	>= 3000	26.71	-6478.7	11.99	102.97
	4	<= 3000	162.49	97.1	>= 7000	41.11	-9602.1	8.78	180.82
4	seq	<= 40	81.25	2.1	>= 80	5.35	-491.8	5.33	77.42
	2	<= 80	116.20	83.5	>= 300	8.37	-967.8	7.59	59.34
	4	<= 900	62.70	17.6	>= 4000	17.59	-3300.3	9.69	94.25
5	seq	<= 600	46.56	1.1	>= 1000	15.87	-4791.7	15.17	46.79
	2	<= 8000	52.64	17.5	>= 30000	22.81	-17599.7	6.08	58.86
	4	<= 8000	91.91	117.5	>= 30000	52.35	-491.9	4.60	96.01
6	seq	<= 90	102.62	1.1	>= 400	19.78	-2588.6	20.20	101.83
	2	<= 800	149.79	37.0	>= 3000	31.33	-6960.7	23.46	151.86
	4	<= 1000	331.29	165.1	>= 5000	47.27	-8693.8	17.85	278.73

Omni-Linux
RINF1 summary



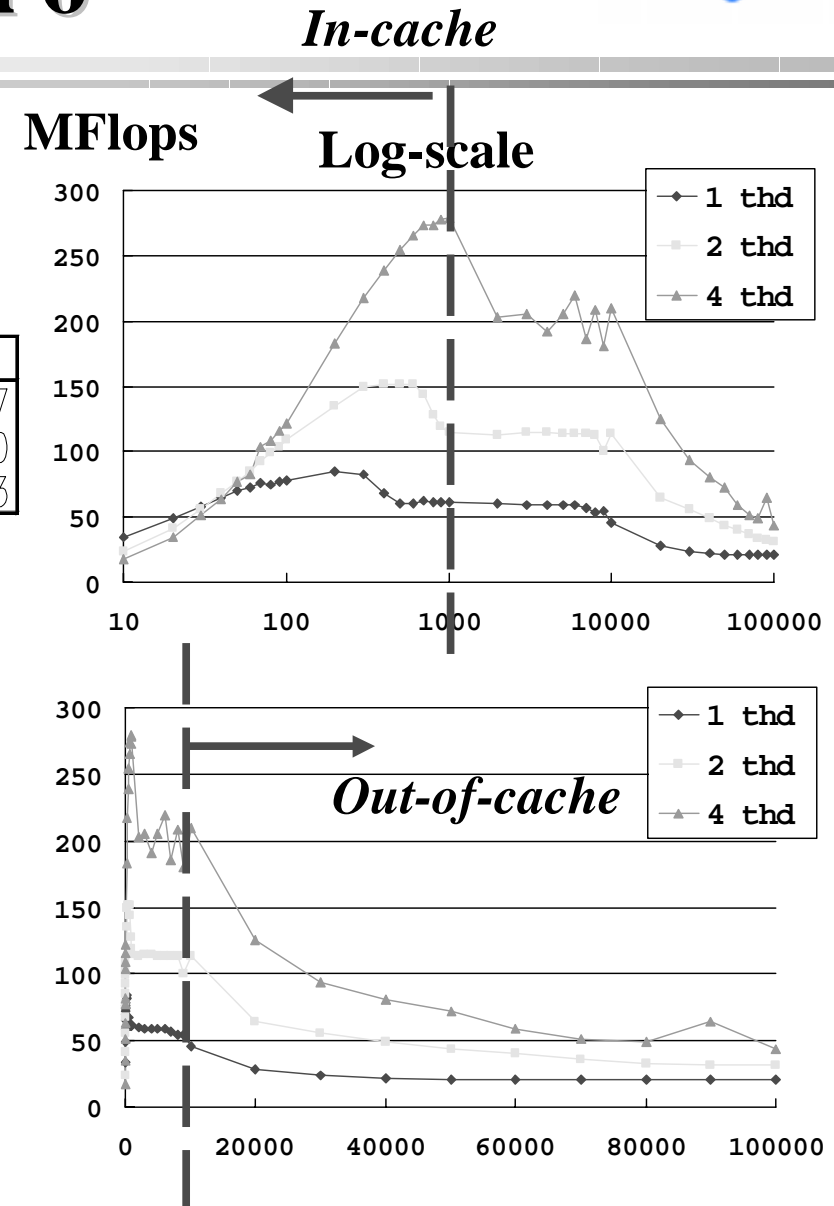
Kernel 6

- ◆ Kernel6 result on Omni-Linux
 - ◆ $A(I)=B(I)*C(I)+D(I)*E(I)+D(I)$

Reported summary

#thd	vlen(1)	r_inf	n_half	vlen(2)	r_inf	n_half
1	<=90	102.62	1.1	>=400	19.78	-4797
2	<=800	149.79	37.0	>=3000	31.33	-6960
4	<=1000	331.29	165.1	>=5000	47.27	-8693

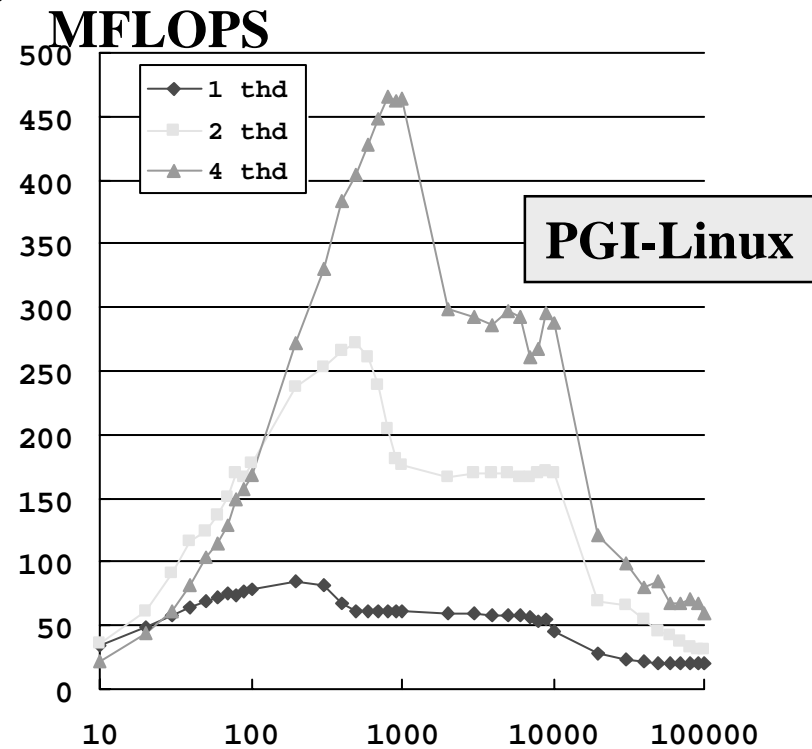
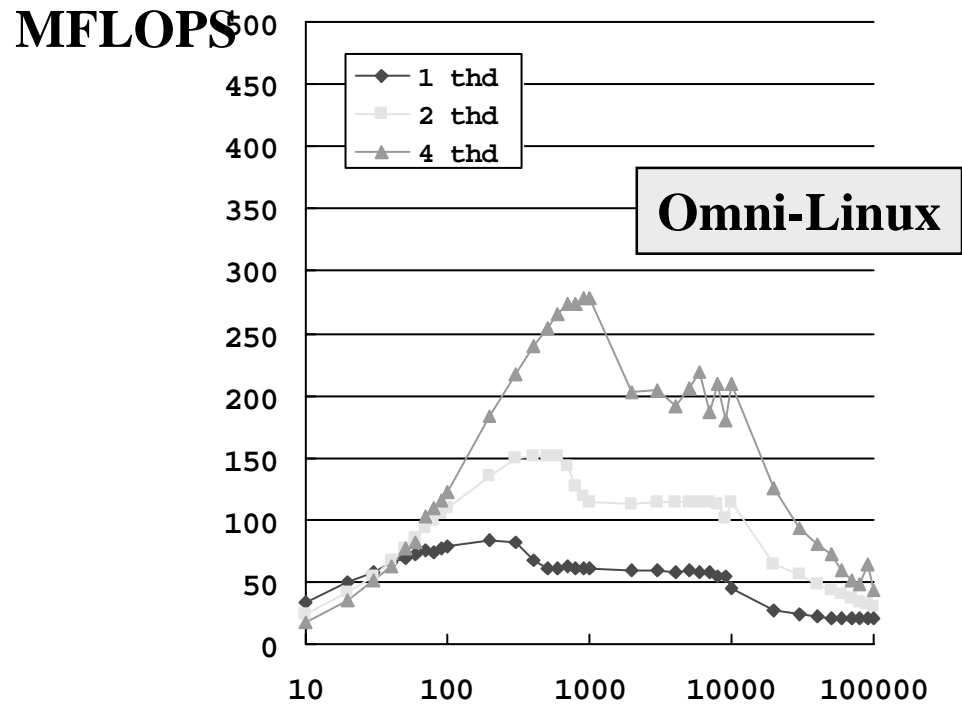
- ◆ vlen(1) and vlen(2) shows the break point of least square fitting.
 - ◆ vlen(1) for in-cache
 - ◆ vlen(2) for out-of-cache
- ◆ The loop overheads are reported in term of the half performance length.





Kernel 6

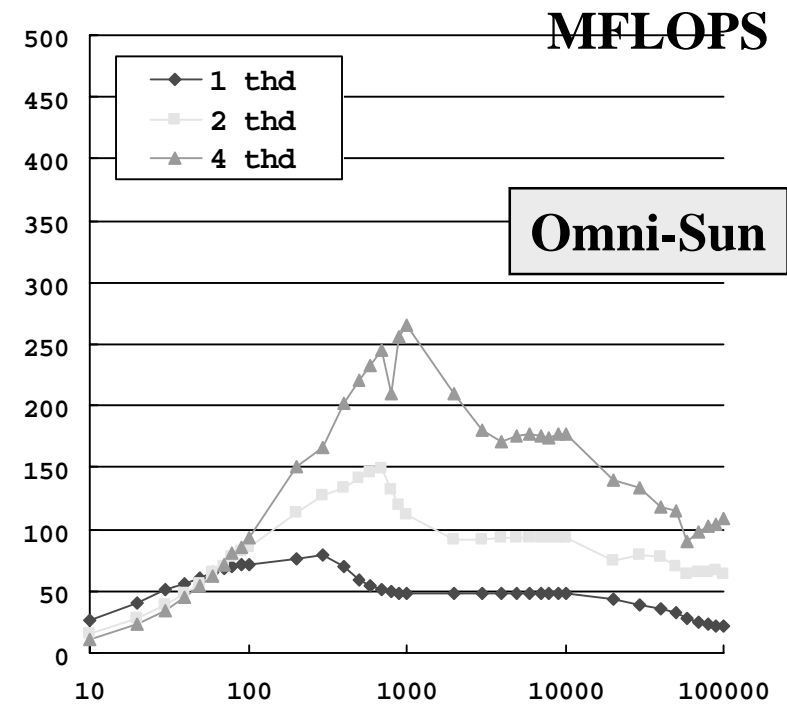
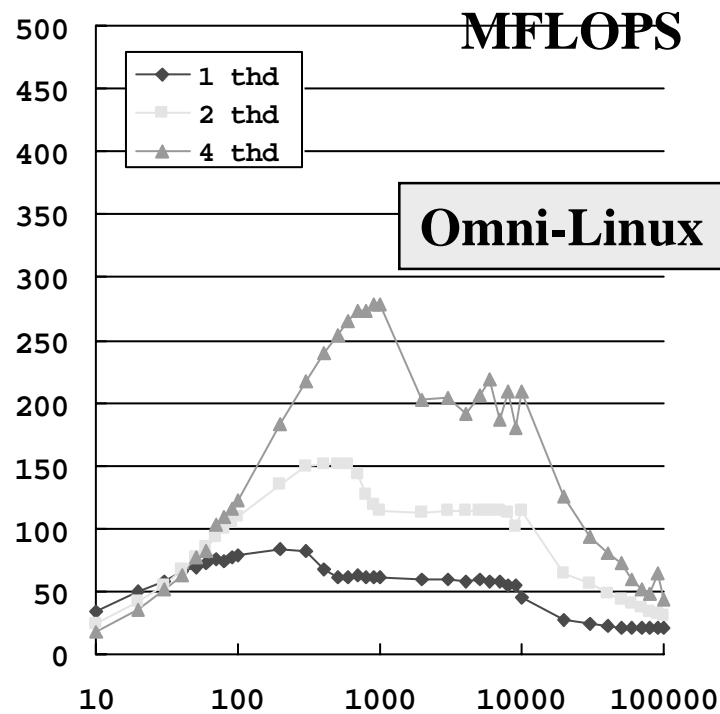
- ◆ The same SMP, different compilers
- ◆ Because Omni is a translator to C code, the results reflect the performance of backend C compiler(gcc).
- ◆ PGI does good instruction scheduling(?).

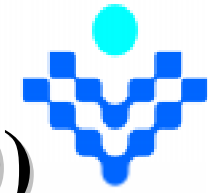




Kernel 6

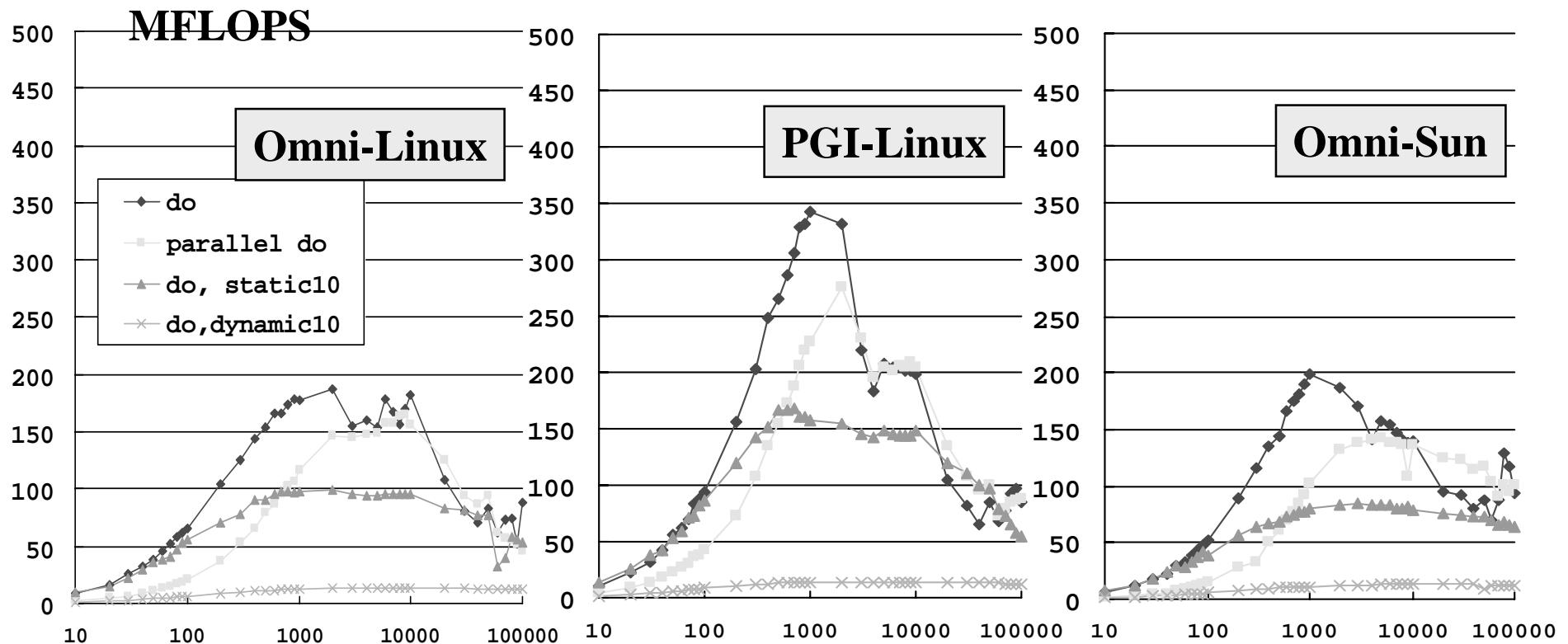
- ◆ The same compiler, different SMPs
- ◆ The in-cache rates are similar.
- ◆ In Sun, the difference in-cache rate and out-of-cache rate is small. Main memory bandwidth is larger than x86's for CPU performance.



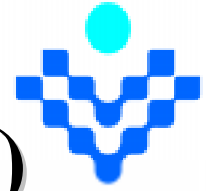


Loop scheduling(kernel 16,18,19,20)

- ◆ Daxpy, $A(I)=S*B(I)+C(I)$, 4 threads
- ◆ Compare different loop scheduling loops
- ◆ The performances of “do” and “do parallel” become closer in larger n .



Loop scheduling(kernel 16,18,19,20)



Kernel 16

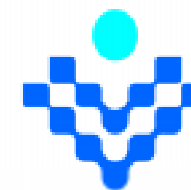
```
T1 = DWALLTIME00()
!$OMP parallel private(JT)
  DO JT=1,NTIM
    CALL DUMMY(JT)
!$OMP do
  DO I=1,N
    A(I)=S*B(I)+C(I)
  END DO
!$OMP end do
  ENDDO
!$OMP end parallel
T2 = DWALLTIME00()
```

Kernel 18

```
T1 = DWALLTIME00()
DO JT=1,NTIM
  CALL DUMMY(JT)
!$OMP parallel do
  DO I=1,N
    A(I)=S*B(I)+C(I)
  END DO
!$OMP end parallel do
ENDDO
T2 = DWALLTIME00()
```

Kernel 19

```
T1 = DWALLTIME00()
!$OMP parallel private(JT)
  DO JT=1,NTIM
    CALL DUMMY(JT)
!$OMP do schedule(static,10)
  DO I=1,N
    A(I)=S*B(I)+C(I)
  END DO
!$OMP end do
  ENDDO
!$OMP end parallel
T2 = DWALLTIME00()
```



POLY1,2: results

- ◆ \hat{r}_∞ shows the (estimated) peak hardware performance.
 - ◆ In both in-cache and out-of-cache, \hat{r}_∞ are almost the same.
- ◆ $f_{1/2}$ shows the ratio of the arithmetic speed and the memory speed.

- ◆ In in-cache case, memory speed is fast enough. Cache performance is independent on the number of threads.
- ◆ When CPU performance increases, the ratio become larger in POLY2(out-of-cache)
- ◆ Sun's memory performance is more well-balanced than x86's performance.

	#thd	POLY1(in-cache)		POLY2(out-of-cache)	
		\hat{r}_∞	$f_{1/2}$	\hat{r}_∞	$f_{1/2}$
Omni-Linux	1	80.28	-0.15	75.86	1.24
	2	160.65	0.01	152.6	1.27
	4	314.13	-0.09	409.09	5.84
PGI-Linux	1	189.81	0.39	233.66	8.3
	2	380.14	0.38	472.96	8.63
	4	761.34	0.43	978.93	9.25
Omni-Sun	1	98.71	0.23	95.89	1.2
	2	197.12	0.2	197.78	2.5
	4	390.89	0.19	349.96	2.02



Comments

◆ Other kernels

- ◆ PGI compiler achieves good performance in in-cache case.
- ◆ The performances of PGI and Omni become close in the kernel that needs memory access (e.g. kernel 10,17)
- ◆ The performance of Sun shared memory is well-balanced than x86's one.

◆ Problems

- ◆ The proposed low-level benchmark is mainly for arithmetic loops. Tested OpenMP directives are limited.
 - Application benchmarks are also required.
- ◆ Does the benchmark reflect actual usage of OpenMP?
 - Typical arithmetic loops, but too simple?
- ◆ The least square fitting sometimes mis-estimates?
 - L1 and L2 cache



Conclusion

- ◆ **The low-level benchmark of PERKBENCH is used as a OpenMP benchmark.**
 - ◆ **Report: shared memory hardware performance and memory/cache bottleneck**
 - ◆ **The overhead of OpenMP is measured with respect to loop performances.**
- ◆ **C version is available.**
- ◆ **We have a plan to make our benchmarks available at Netlib.**