

Efficiency & Optimisation for the IBM POWER4 (p690)

...the shifting sands

John Hague, IBM Hursley Lab

...the shifting sands

- Hardware is much more complex
- Compiler is much more sophisticated
- Compiler writers have spent months matching code to hardware
- So optimisation requirements are changing

Complexity of p690

- p690 Power4 CPU
 - 1.3 GHz
 - Two floating point pipes
 - multiply add every cycle per pipe
 - 6 to 7 cycle pipe length
 - 32 architected FP registers
 - 72 hardware “rename” registers
 - Cache to FP register takes 4 to 5 cycles
 - Up to 100 instructions look ahead
- Compiler knows all this
 - Does application programmer?

Complexity of p690

- Cache
 - L1: 32KB, 2-way associative, FIFO replacement, “store through”
 - L2: 1.44MB shared between 2 CPUs, 4-way associative, latency ~12 cycles, “store in”
 - L3: 128MB shared between 8 CPUs, 8-way associative, latency ~100 cycles
- Memory
 - Latency ~300 cycles
 - Bandwidth (measured on 8GB node) about 3.5 GB/sec for 1 processor
 - Bandwidth (measured) about 10GB/sec for 8 processors
 - Hardware for each processor can handle 8 outstanding cache misses, and 8 prefetch streams
- Compiler knows all this
 - Does programmer know all this?

References

- IBM J of R&D, Vol 46, No 1, Jan 2002, IBM Power4 System
<http://www.research.ibm.com/journal>
- The POWER4 Processor Introduction and Tuning Guide, SG24-7041
<http://www.ibm.com/redbooks>

So now you understand the p690?

Compiler Capabilities

- Compiler can do much more basic tuning
 - unroll loops, split loops, fuse loops, invert loops
 - move invariants outside loop
 - can schedule instructions to match CPU registers and pipes
- Difficult to optimise for memory subsystem
 - can issue prefetch instructions...
 - ...but does not know where data is
 - ...but neither does programmer!
- Still plenty left to do
 - emphasis moves towards high level application optimisation
 - sands are shifting!

Top Tuning Activities

- Add Timing
 - timef() across all OpenMP parallel regions, and all MPI Communications
 - Profiling: gprof and xprofiler
 - hardware monitor?
- Don't copy (or zero out) arrays
- Optimise Data Access
- Combine multiple passes through data
- -qstrict implications
 - Remove divides and parenthisise common expressions
- Use MASS Library
- Use ESSL Library
- Use OpenMP
- Don't overlap communication with CPU
- Use MPI global communication

+ Bits and Pieces

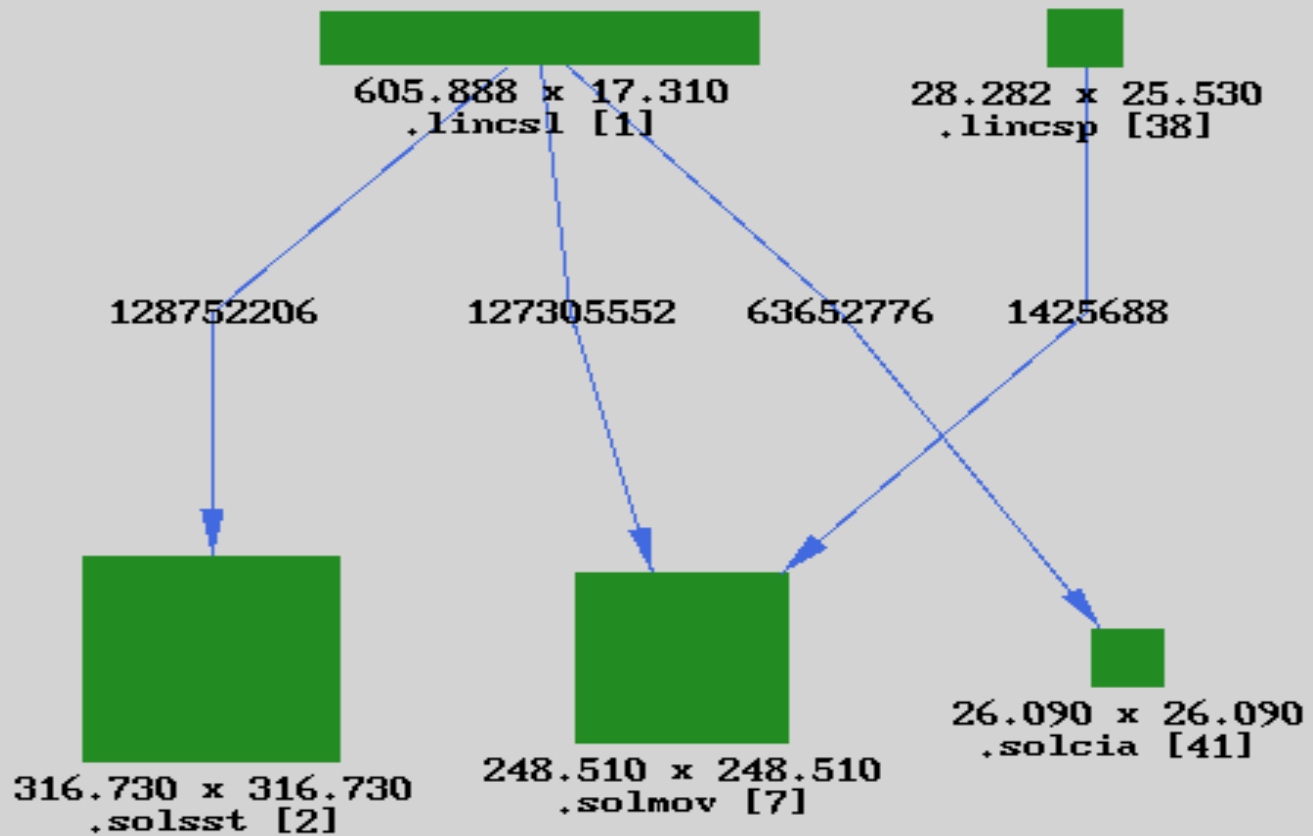
Profiling

- gprof
 - Use -pg when compiling to get number of calls between subroutines
 - Small overhead (unless many small subroutines)
 - gmon.out file produced
 - Analyse with

```
gprof <binary> gmon.out > <fn>
```

- xprofiler
 - GUI: use xprofiler instead of gprof
 - Provides source statement profiling

xprofiler



xprofiler

Source statement profiling

State ment Number	Ticks 1/100th sec	
134		DO JK=ITOPP1,ILEVM1
135	16	DO JL=KIDIA,KFDIA
136	3	ZQDP=1.0_JPRB/(PAPHM1(JL,JK+1)-PAPHM1(JL,JK))
137	16	ZFAC=ZTCOE(JL State-)*ZQDP
138	14	ZTCOE(JL)=PCFM(JL,JK)
139	42	ZDISC=1.0_JPRB/(1.0_JPRB+ZFAC*(1.0_JPRB- ZEBSM(JL,JK-1))+PCFM(JL,JK)*ZQDP)
140		ZEBSM(JL,JK)=ZDISC*(PCFM(JL,JK)*ZQDP)
141		PUDIF(JL,JK)=ZDISC*(PUDIF(JL,JK)+ZFAC*PUDIF(JL,JK-1))
142	104	PVDIF(JL,JK)=ZDISC*(PVDIF(JL,JK)+ZFAC*PVDIF(JL,JK-1))
143	38	ENDDO
144		ENDDO

Profiling with MPI

- With N MPI tasks, profile output (gmon.out.n) takes a long time and a lot of disk space
- So use mpmd to profile just one or two MPI-tasks
 - Add following to command line
 - `-pgmmodel mpmd -cmdfile <filename>`
- Create two binaries
 - ifsMASTER (normal)
 - ifsMASTER.pg (linked with `-pg`)
- Create filename with one line for each MPI task like:
 - ifsMASTER.pg
 - ifsMASTER.pg
 - ifsMASTER
 - ifsMASTER
 -
- Use xprofile to obtain output
 - `xprofiler ifsMASTER.pg gmon.out.0`

Hardware Monitor

- Link with libhpm.a -lpmapi
- Fortran interface from Bob Walkup to count nth block:
 - hpm_begt(n)
 - hpm_endt(n)
 - hpm_prnt()
- ~60 blocks with 8 counters each
- Typical Block
 - Data loaded from L3
 - Data loaded from memory
 - Data loaded from L3.5
 - Data loaded from L2
 - Data loaded from L2.5 shared
 - Processors cycles
 - Instructions Completed
 - Data loaded from L2.5 modified

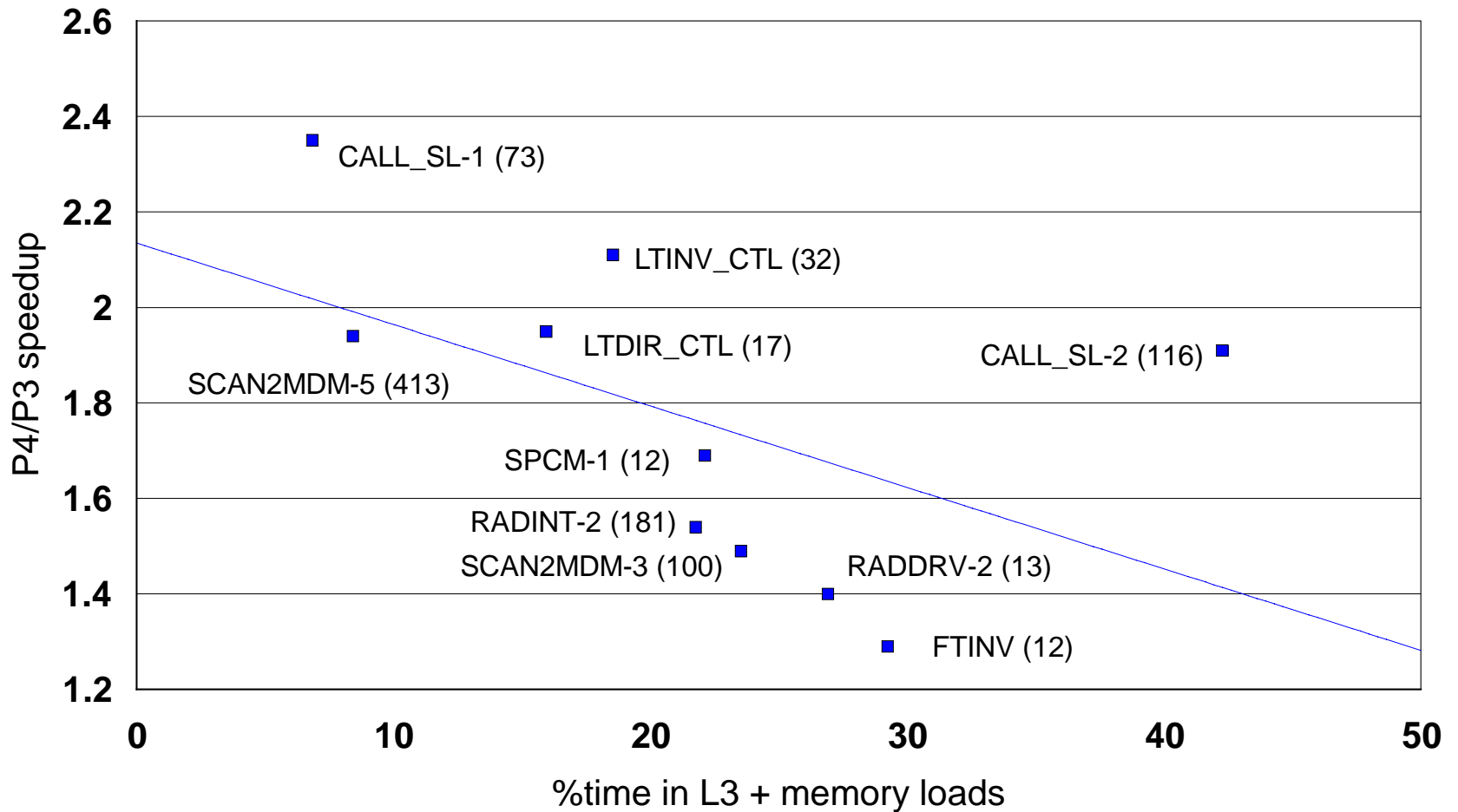
Hardware Monitor

Example of data compiled from hpm blocks

Routine	Sec	P4/P3	Cycles	-----Loads from-----		
				Memory	L3	L2
SCAN2MDM-5	413	1.94	534G	69M	93.7M	3655M
RADINT-2	181	1.54	228G	20M	358M	1741M
CALL_SL-2	116	1.91	147G	83M	185M	623M
SCAN2MDM-3	100	1.49	115G	47M	32.6M	747M
CALL_SL-1	73	2.35	92G	7.3M	23.9M	413M
LTINV_CTL	32	2.11	40G	12.4M	10.9M	721M
LTDIR_CTL	17	1.95	22G	5.4M	7.3M	403M
RADDR-2	13	1.40	14.9G	7.0M	5.1M	65M
FTINV	12	1.29	13.6G	7.3M	3.1M	62M
SPCM-1	12	1.69	14.8G	5.3M	5.6M	235M

Hardware Monitor

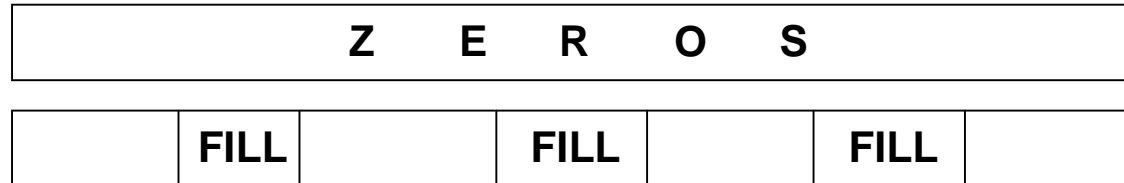
P4/P3 speedup correlated to memory loads



Don't Copy or Zero Arrays

- Don't copy arrays
 - Scalar cache based machine hates it
 - Sometimes done for safety – to make sure data not overwritten or not initialised
- Don't zero out arrays before filling

Don't do



Do



Optimise Data Access (1)

- Access data sequentially (stride 1)
 - Left most Fortran index in array
 - Reorder Loops

```
DO I=1,N
  DO L=1,LEV
    C(IND(I),L)=A(I,L)
  ENDDO
ENDDO
```

```
DO L=1,LEV
  DO I=1,N
    C(IND(I),L)=A(I,L)
  ENDDO
ENDDO
```

- Minimise stride non 1

```
DO J=1,N
  DO I=1,N
    C(I)=A(I)*B(J,I)
  ENDDO
ENDDO
```

```
DO I=1,N
  DO J=1,N
    C(I)=A(I)*B(J,I)
  ENDDO
ENDDO
```

Optimise Data Access (2)

- Data re-use with blocking

A1	A2
A3	A4

 \times

B1	B2
B3	B4

 =

A1xB1+ A2xB3	A1xB2+ A2xB4
A3xB1+ A4xB3	A3xB2+ A4xB4

Minimise passes through data

Don't Do

```
DO L=1,KLEV
  DO J=KIDIA,KFDIA
    T=A(I(J),L)
    ...
  ENDDO
ENDDO
```

```
DO L=1,KLEV
  DO J=KIDIA,KFDIA
    Z=A(I(J),L)
    ...
  ENDDO
ENDDO
```

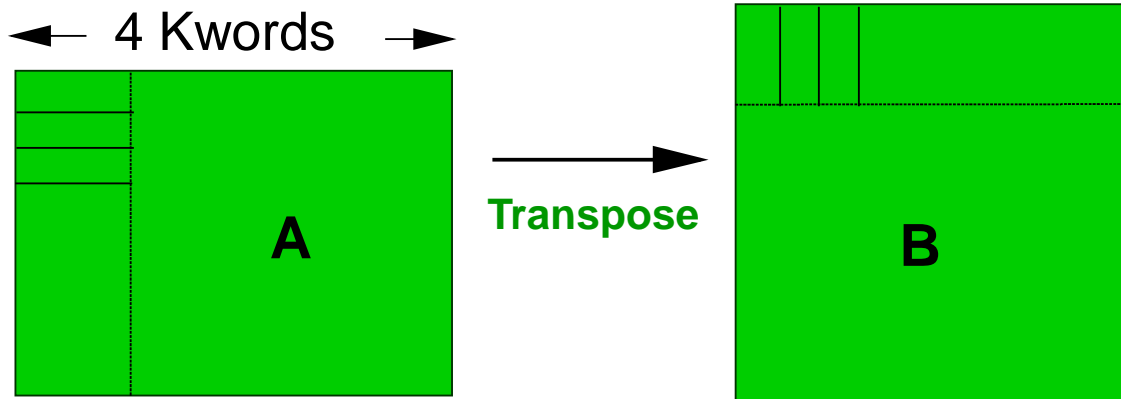
Transpose Array

Recursive algorithm in y direction

Transpose

Recursive algorithm in x direction

Transpose



```
DO II=1,N,NB
  DO J=1,N
    DO I=II,II+NB-1
      B(I,J)=A(J,I)
    ENDDO
  ENDDO
ENDDO
```

Effect of -qstrict

- Use of `-qstrict`
 - Many software vendors like to use this
 - Forces strict order of computation
 - Prevents certain optimisations
 - Particularly for common expressions, divide, and dependencies
- ECMWF need bitwise identical results with different number of MPI tasks
 - Loop count (e.g. over grid points) varies with number of nodes
- And results not bitwise identical if loop count varies:
 - unrolled part may be reciprocal instead of divide
 - tidy up at end may be divide
 - more stringent than required
 - `-qnounroll` not sufficient

Common Expressions

- Because of `-qstrict`, common expressions need parenthesis to prevent re-computation

```
DO I=1,N
```

```
  A(I)=B(I)*C(I)*D(I)
```

```
  X(I)=Y(I)*C(I)*D(I)
```

```
ENDDO
```

```
DO I=1,N
```

```
  A(I)=B(I)*(C(I)*D(I))
```

```
  X(I)=Y(I)*(C(I)*D(I))
```

```
ENDDO
```

Replace Divides

- Replace repeated divides, by repeated reciprocal multiplies
- Saves about 25 cycles per divide

```
DO I=1,N
  A(I)=B(I)/X(I)
  C(I)=D(I)/X(I)
ENDDO
```



```
DO I=1,N
  Z=1.0/X(I)
  A(I)=B(I)*Z
  C(I)=D(I)*Z
ENDDO
```

```
DO I=1,N
  E(I)=A(I)/C(I)
  +B(I)/D(I)
ENDDO
```



```
DO I=1,N
  E(I)=(A(I)*D(I)+B(I)*C(I))
  /(C(I)*D(I))
ENDDO
```

- Compiler would do this with `-O3` without `-qstrict`

Remove Dependencies

- Remove dependencies forced by `-qstrict`

```
DO I=1,N          → DO I=1,N,4
  S = S + A(I)    S1 = S1 + A(I  )
ENDDO             S2 = S2 + A(I+1)
                  S3 = S3 + A(I+2)
                  S4 = S4 + A(I+3)
                  ENDDO
                  S = S1+ S2 + S3 + S4
```

- Compiler would do this with `-O3` without `-qstrict`

Use MASS Library

Mathematical Accelerator Subsystem*

- Least significant bit accuracy sacrificed for better performance
- For scalar intrinsics (e.g. exp, log, **, cos, sin etc)
 - No change to code or recompilation required
 - Relink only
 - Typically 2 x speedup
- For vector intrinsics
 - Code change may be required
 - Compiler will generate with `-qhot` (and `-qnostrict`)
 - For example
 - call `vexp(y,x,N)`
 - Typically 5 x speedup for vector

* <http://www.rs6000.ibm.com/resource/technology/MASS>

Use ESSL Libraries

Engineering and Scientific Subroutine Library

- Particularly good for routines where cache re-use is important. E.g.:
 - FFTs, Matrix multiplication, Linear Equation solvers
- More basic routines may be better in Fortran. E.g:
 - Daxpy + Dot Product

```
CALL DAXPY(N,A,P,1,R,1)
S=DDOT(N,R,1,P,1)
```



```
DO I=1,N
  R(I)=R(I)+A*P(I)
  S=S+P(I)*R(I)
ENDDO
```

- Matrix vector multiply

```
CALL DGEMX(N,.,M,.,X,.,Y)
CALL DGEMX(N,.,M,.,A,.,B) →
```

```
DO I=1,N
  DO J=1,N
    Y(I)=Y(I)+X(J)*M(I,J)
    B(I)=B(I)+A(J)*M(I,J)
  ENDDO
ENDDO
```

Best if Matrix M too
large for cache

Best if Matrix M fits
in cache

Use OpenMP

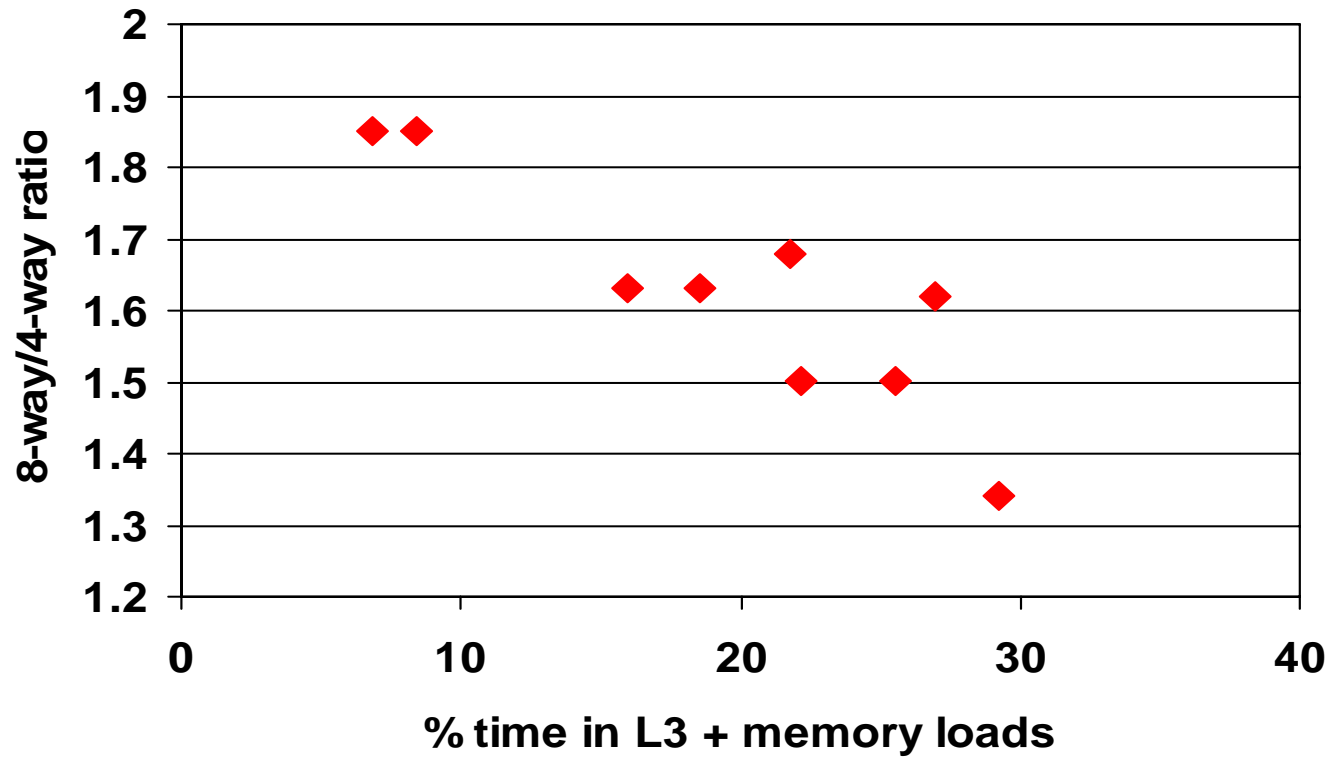
- Specific parallel regions (-qsmp=omp)

```
!OMP$ DO PARALLEL PRIVATE(J...)  
DO J=1,N  
    A(J)=B(J)  
ENDDO
```

- Need to parallelise high percentage of loops
 - Scalability affected by
 - Thread dispatching (~20 usec overhead)
 - Memory bandwidth (3.5GB/s single, 10 GB/s 8-way LPAR)
 - Data in wrong cache
 - Cache line interference
 - Load imbalance
- but can use DYNAMIC, GUIDED or AFFINITY Scheduling

OpenMP Scalability

8-way/4-way memory dependence



Expect less memory interference with MPI

MPI Communication

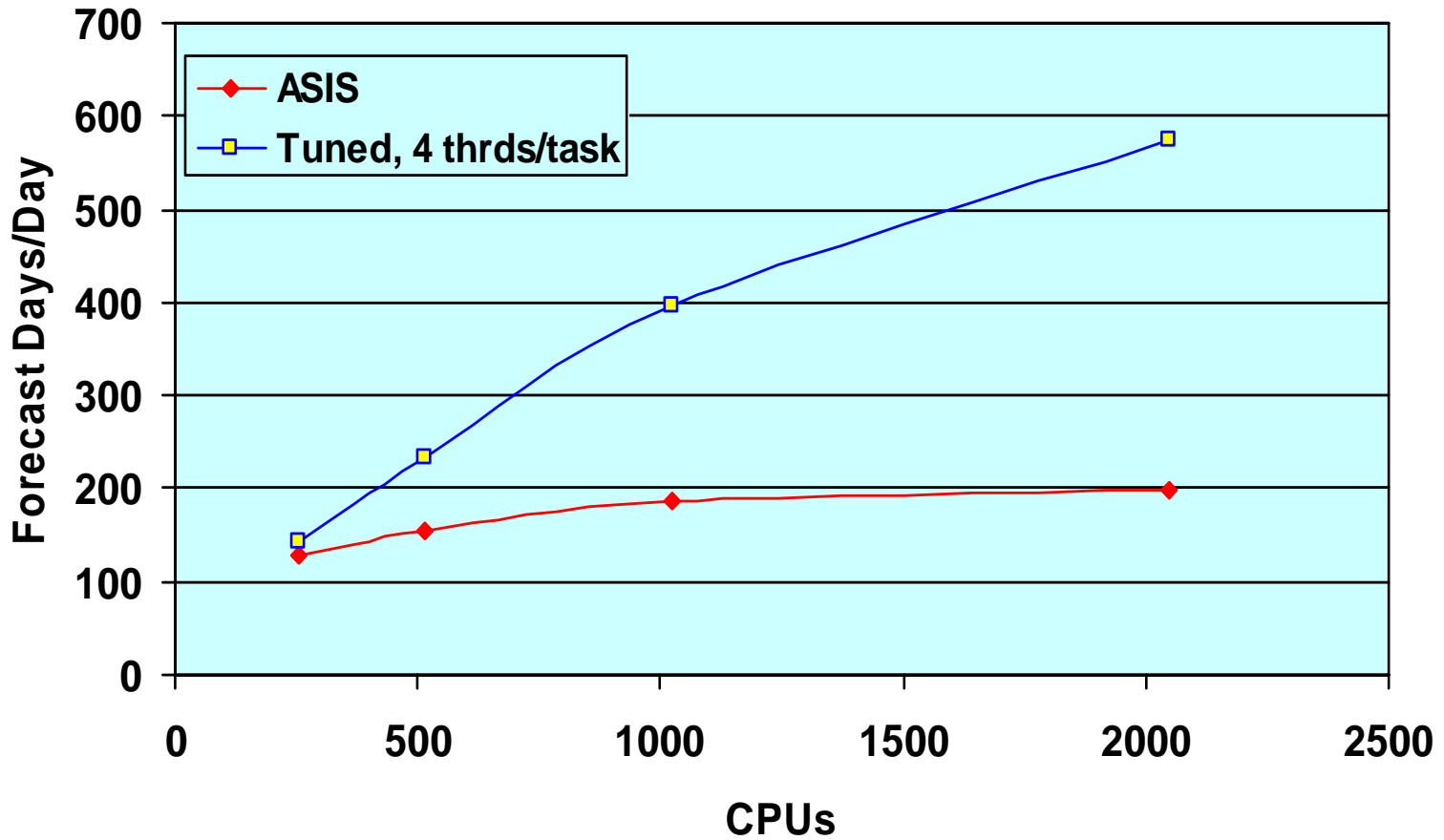
- Don't overlap with application CPU
 - intuitively good thing...
 - ...but communication takes CPU
 - so application CPU slows down communication and everyone suffers
 - effect increases with processors (factor of 2 with 2000 processors)
 - replace buffered sends with blocking or non-blocking sends
- Use MPI global communication
 - e.g. broadcast, reduce, allreduce, alltoall

Use OpenMP with MPI

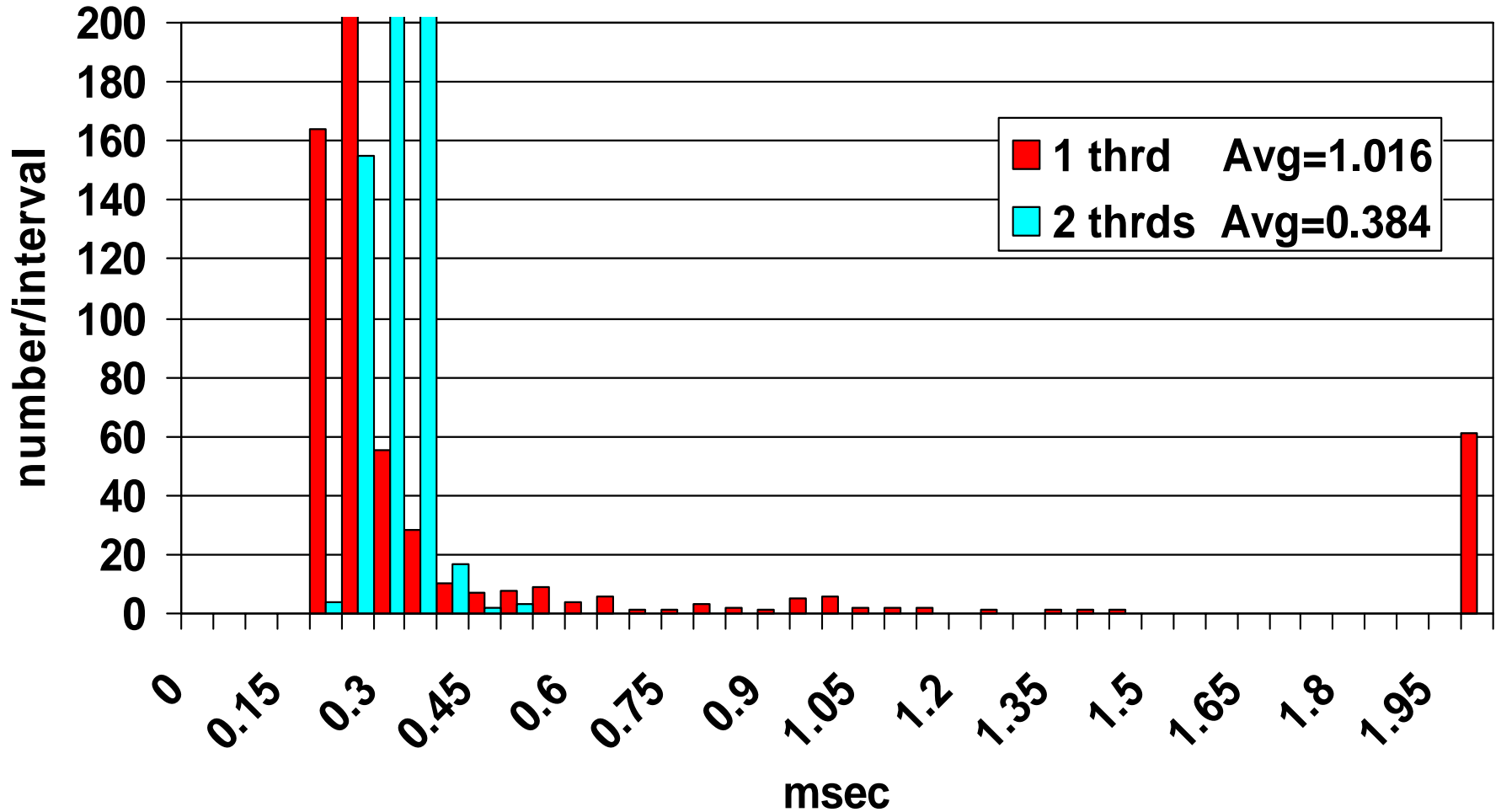
- Reduces MPI collective communication time
 - application cpu time proportional to $1/(\text{number of processors})$
 - global communication time proportional to $\log_2(\text{number of MPI tasks})$
- Reduces “fat halo” % communication time
 - for large number of MPI-tasks, “fat halo” remains constant with number of MPI tasks
- Removes “ASCI” effect
 - daemon interrupts on multiple nodes get serialised during collective communication (e.g. barrier)
 - using OpenMP:
 - master thread handles collective communication
 - other processors available to handle interrupts from daemons

ECMWF's IFS at LLNL

T511 (15min t/s)

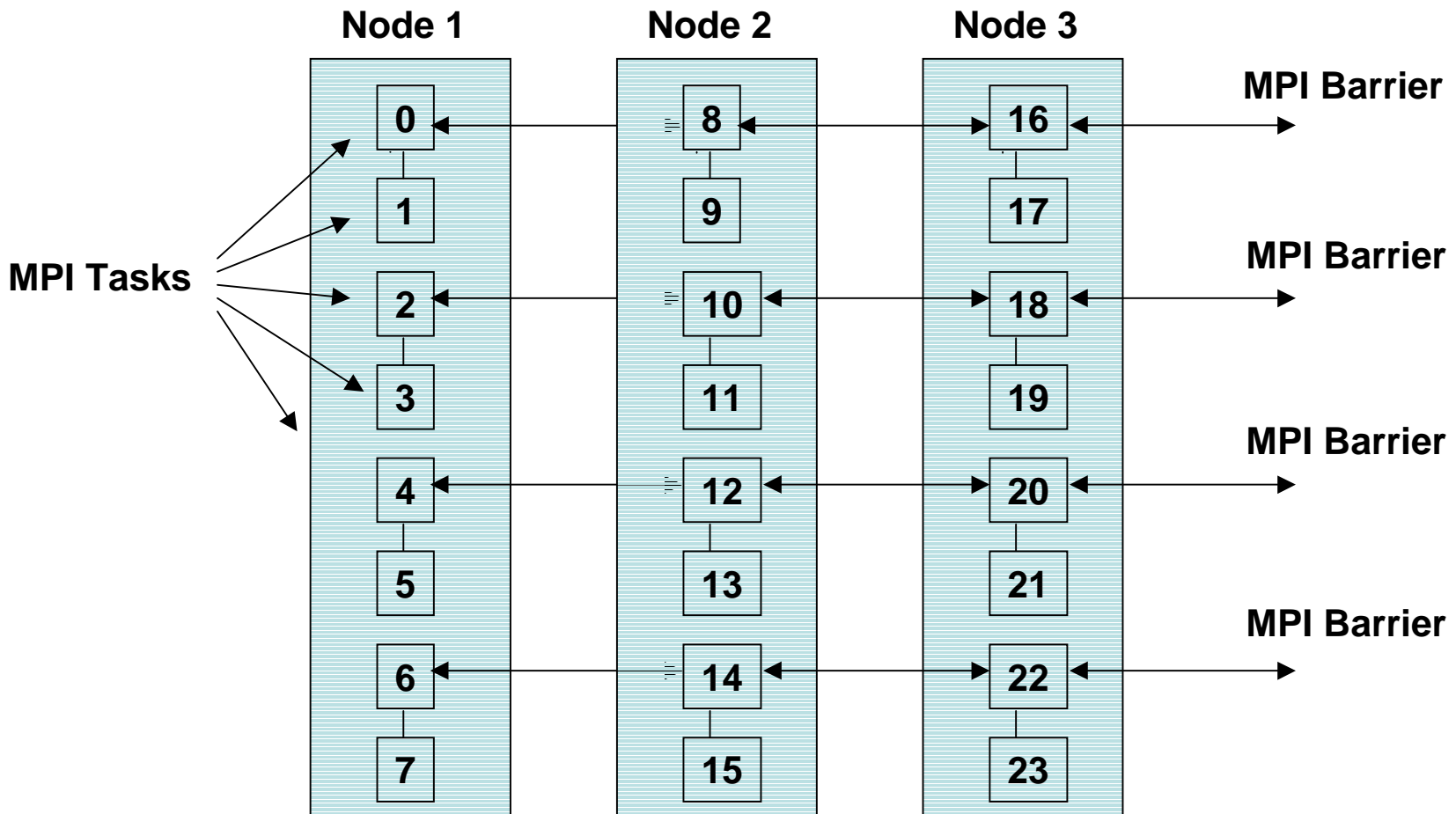


MPI Barrier on 960 Processors

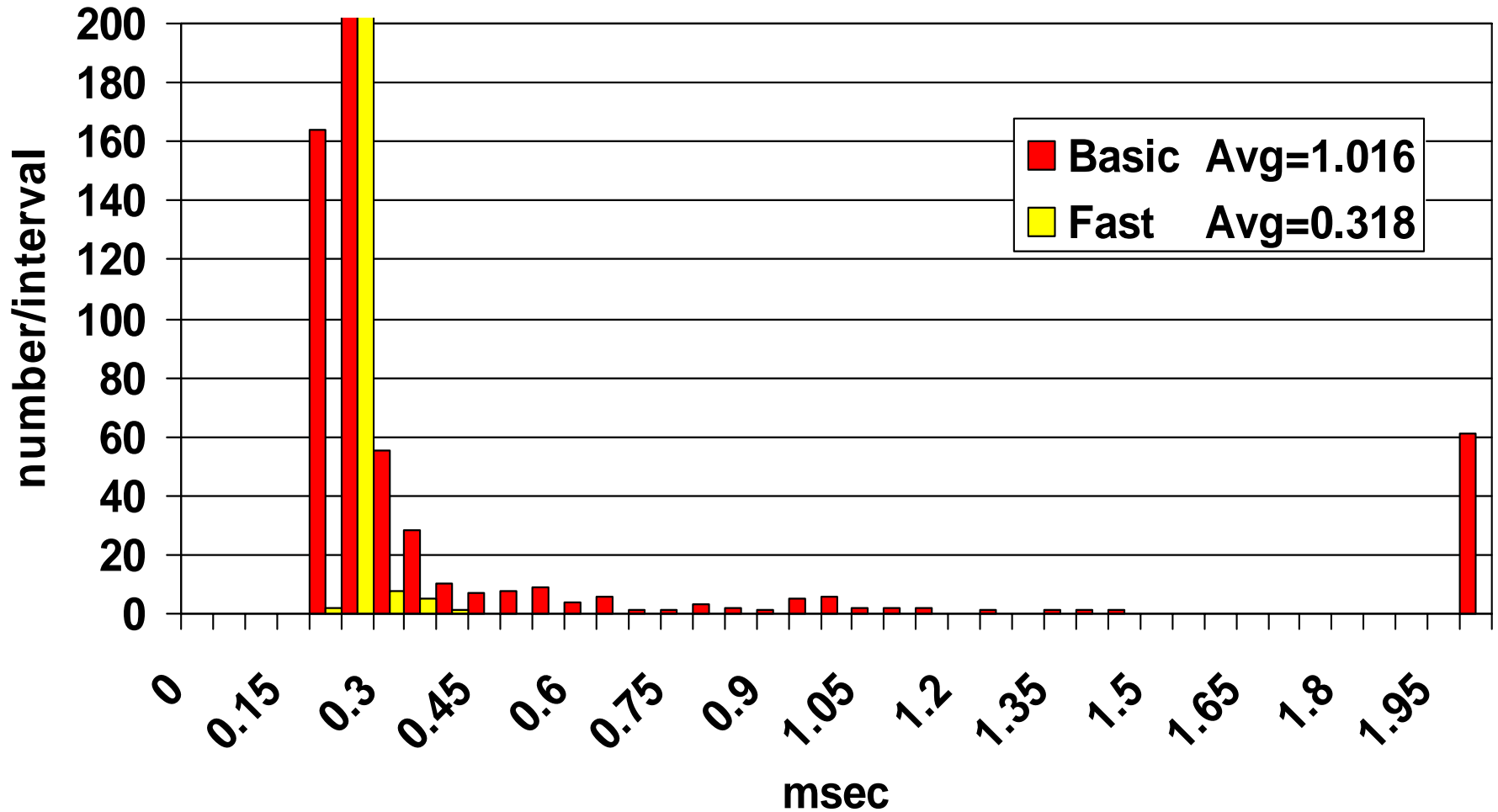


Fast MPI_BARRIER

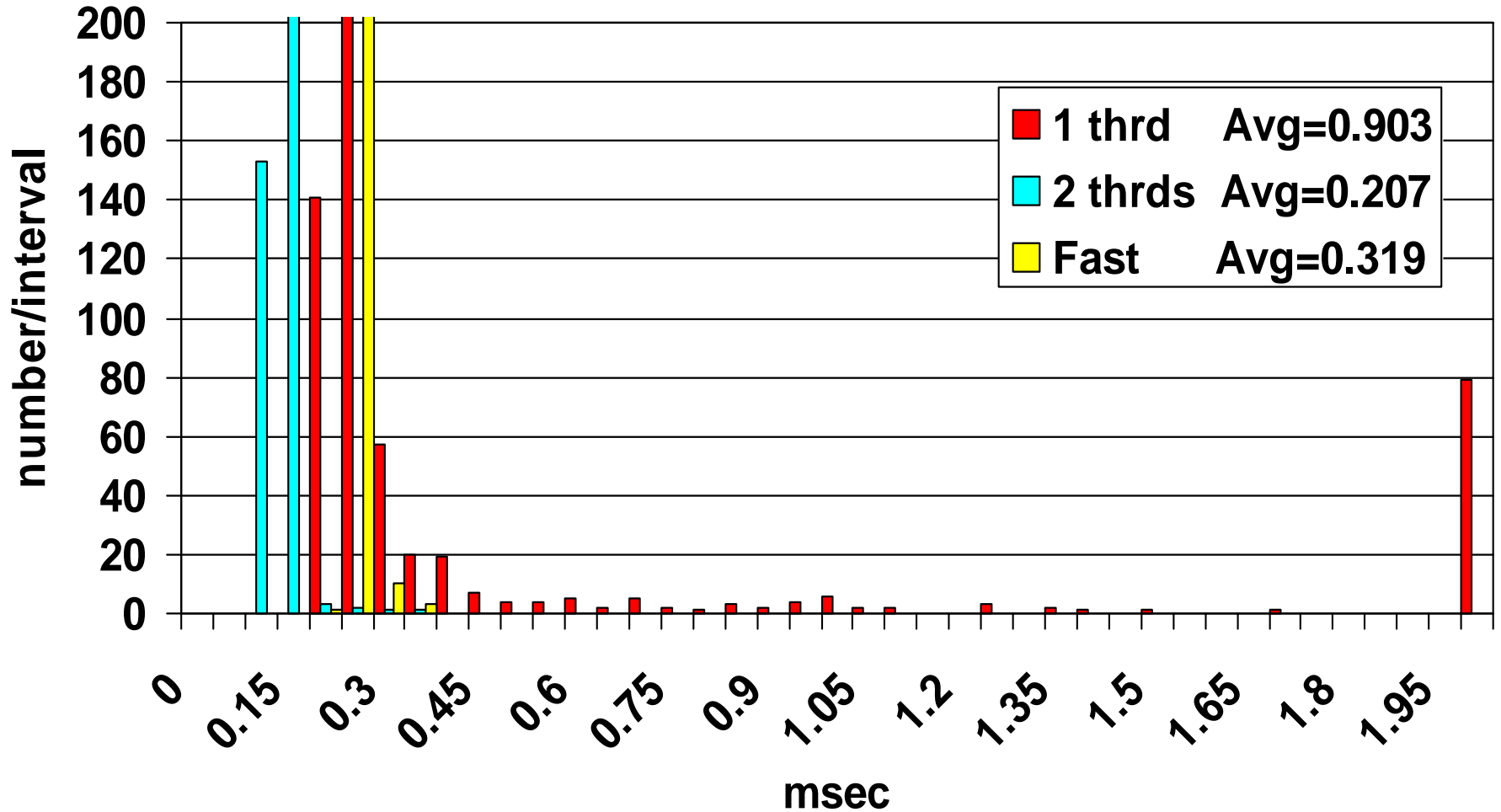
Only even numbered nodes participate in MPI_BARRIER



MPI Fast Barrier on 960 Processors



MPI Barrier on 240 Processors



Bits and Pieces

- Floating Point Trapping
- Prefetch Data
- Fractional part of a number
- Object Code Listings
- Minimise IF Statements
- @profile Directive
- Inlining
- Directives (CACHE_ZERO, ASSERT, NOUNROLL)
- Compiler Flags
- I/O Tuning
- Towards Max MFLOPS

Floating Point Trapping

- Traditional method is to compile with

`-qfltrap=enable:zerodivide:overflow...`

- But can give high overhead, so at start of program
 - Call `signal_trap` to permit F.P. exception trapping
 - Call `fp_enable` with MASK to specify traps required
 - Call `signal` to specify traceback for specified signals

Prefetch Data

Untuned Copy

```
DO J=1,N
  Y(J)=X(J)
ENDDO
```

Tuned Copy for Power3

```
DO J=1,N
  Y(J)=X(J)+ZERO*Y(J)
ENDDO
```

- # Load of Y(J) activates prefetch
- # Faster if data not in Cache
- # Extra load takes longer if data in cache

Tuned Copy for Power4

```
DO JJ=1,N,16
  !IBM* PREFETCH_FOR_STORE Y(JJ+16)
  !IBM* PREFETCH_FOR_LOAD X(JJ+16)
  DO J=JJ, JJ+15
    Y(J)=X(J)
  ENDDO
ENDDO
```

- # Power4 has lots of store buffers
- # Compiler and Hardware can activate prefetch

Prefetch Streams

- Prefetch stream h/w ramp-up of 128 byte line

			N					
Line ref	1	2	3	4	5	6	7	8(steady state)
L1	-	-	N+1	N+2	N+3	N+4	N+5	N+6
L2	-	-	N+2	N+3->5	N+6->7	N+8	N+9	N+10
L3	-	-	-	-	N+8->11	N+12->15		12 lines ahead every 4 th line

- Not effective for less than 32 D.P. words

Fractional Part of a number

- Usual (23 cycles)

```
f = x - float(int(x))
```

- Tuned (13 cycles)

Must be `-qstrict`

```
parameter(rnd=2d0**52+2d0**51) !D.P.
```

```
t = x-sign(0.5d0,f)
```

```
f = x-((rnd+t)-rnd)
```

```
parameter(rnd=2e0**52+2e0**51) !S.P.
```

Mod Function (real)

- Usual (58 cycles)

```
z=mod(f,c)
```

- Tuned (15 cycles)

Must be `-qstrict`

```
t=f*(1.d0/c)  
x=t-sign(0.5d0,t)  
t=t-((rnd+x)-rnd)  
z=c*t
```

Use object code listing

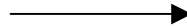
Obtain with `-qsource,list`

```
2 |      REAL*8 S,X,B(*),A(*)
3 |      DO I=1,N
4 |          S=S+B(I)*X/A(I)
5 |      ENDDO
```

```
3 |                                     CL.0:
4 | 000038 fdiv   FC831024 15  DFL      fp4=fp3,fp2,fcr
4 | 00003C lfd   CC460008  0  LFDU     fp2,gr6=a(gr6,8)
4 | 000040 fmul   FC650032  1  MFL     fp3=fp5,fp0,fcr
4 | 000044 lfd   CCA50008  0  LFDU     fp5,gr5=b(gr5,8)
4 | 000048 fadd   FC24082A  3  AFL     fp1=fp4,fp1,fcr
0 | 00004C bc     4200FFEC  0  BCT     ctr=CL.0,
```

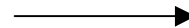
Minimise IF Statements

```
DO J=1,N
  IF(J.EQ.1) THEN
    A(J)=1.0
  ELSE
    A(J)=B(J)
  ENDIF
ENDDO
```



```
A(1)=1.0
DO J=2,N
  A(J)=B(J)
ENDDO
```

```
DO J=1,N
  IF(K(I).EQ.0) X(J)=0.0
  A(J)=X(J)+C*B(J)
ENDDO
```



```
IF(K(I).EQ.0) THEN
  DO J=1,N
    X(J)=0.0
    A(J)=C*B(J)
  ENDDO
ELSE
  DO J=1,N
    A(J)=X(J)+C*B(J)
  ENDDO
ENDIF
```

Minimise IF Statements

- Use Max and Min instead of IF
 - Compiler will use select instruction (with `-qarch=pwr4` and `-qnostrict`)

```
IF(A(I).LT.0.0) A(I)=0.0
```

4 cycles

```
A(I)=MAX(A(I),0.0)
```

1.5 cycles

@process Directive

- Can use to change optimisation. E.g.
 - NOOPT for test if compiler problem suspected
 - NOSTRICT to improve performance
 - NOUNROLL to prevent unrolling
 - E.g.

@process STRICT,NOUNROLL

Inlining

- Select subroutines from xprofiler output
 - Flat profile gives time per call
- Use compiler flags to inline routines with
 - Small amount of code
 - Low time per call
- For routines in same file

```
xlf90 -O -Q+sub1:sub2
```
- For routines in different file

```
xlf90 -O -qipa=inline=sub1:sub2 z1.f z2.f
```
- Effective only if lots of very short routines

Directives

- `!IBM* CACHE_ZERO(a(i))`
 - Zeroes 128byte cache line without accessing memory
 - Best performance if data not in memory
- `!IBM* ASSERT ITERCNT(N)`
 - If low N, compiler will create non-looped code
- `!IBM* NOUNROLL`
 - Will prevent unrolling of particular loop

Compiler Flags

- -O3 -qstrict -qarch=pwr4 -qtune=pwr4 -qsmp=omp
 - My “standard” set
- -qhot
 - Good for array syntax, inverting loops etc
- -qipa
 - Interprocedural analysis (may increase compile time)
- -O4
 - -qhot -qipa -O3 -qarch=auto -qtune=auto -qcache=auto
- -qsmp=omp
 - Permits specific (non-automatic) OpenMP
- -qalias(noaryovrlap)
 - Specifies no array overlap in statements like
`a(j:k)=a(m:n)`

I/O Tuning

- Minimise calls to I/O subroutines
 - Do not use `WRITE(1) (A(I,J),I=1,N),J=1,N)`
 - Use `WRITE(1) A`
- Use Direct Access

```
DO I=1,N
    WRITE(1,IREC=I) A
ENDDO
```
- For Large array, write Forwards, Read Backwards

```
DO I=N,1,-1
    READ(1,IREC=I)
ENDDO
```

Towards Max MFLOP

```
DO I=1,N
  DO J=1,N
    DO K=1,N
      S=S+A(J,K)*B(K,I)
    ENDDO
  ENDDO
ENDDO
```

2 Loads, 1FMAs,
Load/Store bound

```
DO I=1,N,3
  DO J=1,N,2
    DO K=1,N
      S00=S00+A(J ,K)*B(K,I )
      S01=S01+A(J ,K)*B(K,I+1)
      S02=S02+A(J ,K)*B(K,I+2)
      S10=S10+A(J+1,K)*B(K,I )
      S11=S11+A(J+1,K)*B(K,I+1)
      S12=S12+A(J+1,K)*B(K,I+2)
    ENDDO
  ENDDO
ENDDO
```

5 Loads, 6FMAs, FMA bound