

OpenMP at Sun

EWOMP 2000, Edinburgh

September 14-15, 2000

Larry Meadows

Sun Microsystems



Outline

- Sun and Parallelism
- Implementation
 - Compiler
 - Runtime
- Performance Analyzer
 - Collection of data
 - Data analysis
- Performance and case studies
 - MM5: A personal odyssey
 - Selected EPCC microbenchmarks
 - NAS Parallel benchmarks in OpenMP and MPI

Sun and Parallelism

Sun leads the way in parallelism

- On the Desktop
 - SPARCstation 10 in 1992 had 4 cpus
 - UltraSPARC-based desktops starting in 1995
- And in the Data center
 - SPARCServer 600MP in 1991
 - SPARCCenter 2000 in 1992(20 cpus)
 - Sunfire in 1996 (64 cpus)
 - E10000 (Starfire) in 1997 (64 Ultra-II cpus)

Parallel Compilers

- Automatic parallelism for F77, F95, C, C++
- Sun directive-based parallelism for F77, F95, C
- OpenMP for F95 (ABI Version 1.1)
- Common back end
 - Front ends process directives
 - Back end implements parallelism (automatic and directive based)
- Common runtime
 - Microtasking library controls parallel execution
 - Mixed-language, automatic, and explicit parallelism can co-exist

Parallel Tools

- **Debuggers**
 - dbx: thread aware single-process debugger
 - prism: thread aware multi-process debugger
- **Performance analysis**
 - collector: instrument multi-threaded programs without recompilation
 - analyzer: view results of collector to understand performance
- **Compiler commentary**
 - Messages on parallelization
 - Commentary exported to image file, available through analyzer

Compiling for OpenMP/Autopar

- OpenMP available in Forte™ Developer 6 Fortran 90/95
- Parallelize only OpenMP directives:
`f90 -openmp file.f`
- Automatically parallelize loops:
`f90 -autopar file.f`
- Parallelize OpenMP directives and automatically parallelize loops:
`f90 -autopar -openmp file.f`

OpenMP Implementation

- **Compiler Front end**
 - Recognize directives
 - Check syntax, language-specific processing
 - Pass information to optimizer
- **Optimizer (`iropt`)**
 - Receive directive information
 - Transform program for parallel execution
 - Insert calls to microtasking library
- **Microtasking library (`libmtask.a`)**
 - Manage threads
 - Control distribution and scheduling of work

Transformations

Parallel regions are transformed as follows:

- Determine properties of variables (`SHARED`, `PRIVATE`, `FIRST/LASTPRIVATE`, `REDUCTION`, etc). Note: This is done automatically for Sun directives.
- Extract the body of the parallel region and place it in a separate subroutine (the *outlined* routine).
- Replace the parallel region with a call to `__mt_MasterFunction`, passing the address of the outlined routine.

Outlining Example: Before

```
!$OMP PARALLEL PRIVATE(ID) SHARED(A, B, C)
    CALL Work (ID)

!$OMP DO PRIVATE(I)
    DO I = 1, 1000
        C(I) = A(I) + B(I) * ID
    END DO
!$OMP END DO

. . .
!$OMP END PARALLEL
```

Outlining Example: After (1)

```
TASK_INFO_T TaskInfo
TaskInfo = ...
call __mt_MasterFunction_ (TaskInfo, PARALLEL_001,
    A, B, C, ...)
END
SUBROUTINE PARALLEL_001 (A, B, C)
INTEGER A(1000), B(1000), C(1000)      ! SHARED
INTEGER ID                             ! PRIVATE
TASK_INFO_T TaskInfo
CALL Work (ID)
TaskInfo = ...
call __mt_MasterFunction_ (TaskInfo, DOALL_001,
    A, B, C, ID...)
...
END SUBROUTINE PARALLEL_001
```

Outlining Example: After (2)

```
SUBROUTINE DOALL_001 (TaskInfo, A, B, C, ID)
TASK_INFO_T TaskInfo
INTEGER A(1000), B(1000), C(1000)      ! SHARED
INTEGER ID
! PRIVATE IN REGION, PASSED AS ARGUMENT TO DOALL
INTEGER I                                ! PRIVATE

DO I = TaskInfo%FROM, TaskInfo%TO, TaskInfo%STEP
    C(I) = A(I) + B(I)
END DO
END SUBROUTINE DOALL_001
```

Microtasking Library

- Based on Solaris threads
- Threads initialized once on first call (increased if necessary on subsequent calls)
- Idle threads busy-wait by default
- Environment variables (Sun-specific):
 - `MT_BIND_LWP = {FALSE, *TRUE}` – bind thread to LWP
 - `MT_BIND_PROCESSOR = {*FALSE, TRUE}` – bind LWP to processor
 - `MT_SPIN = {*FALSE, TRUE}` – allow sleep while waiting for work (spin time tunable with `SUNW_MP_THR_IDLE`)
 - `STACKSIZE = N` – Per-thread stacksize is $N * 1024$ bytes

Performance Analyzer

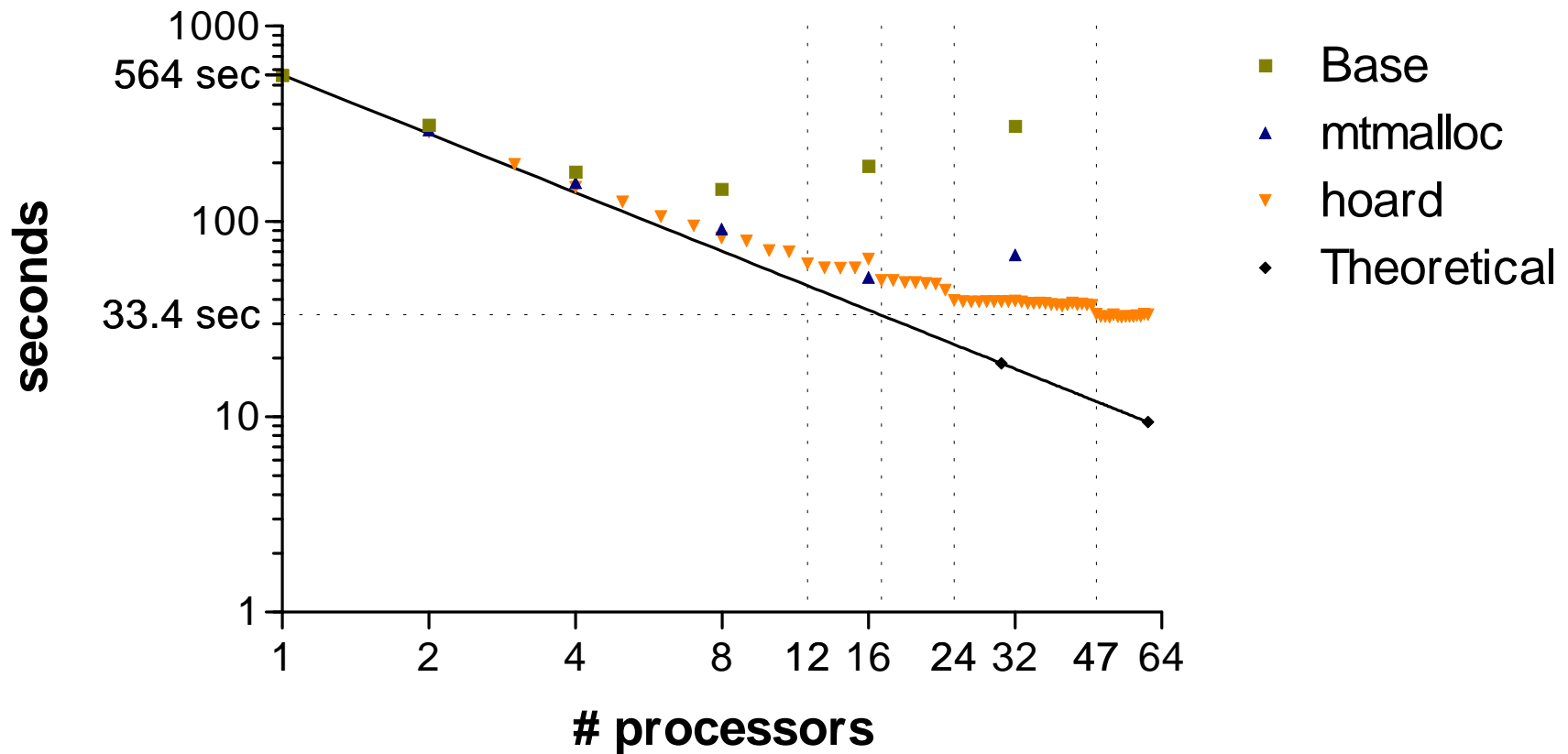
- User Model
 - Compile target as for production run
 - No recompilation
 - Full optimization: -g does not alter generated code
 - Collect the data (`collect` command or `dbx`)
 - Analyze the data (GUI or command-line)
- Event-based Performance Metrics
 - Event occurs at timer-tick, synchronization (mutex), or HW counter overflow
 - Each event has data and full callstack
 - Metrics computed against program using event data and callstack

Case Study: MM5

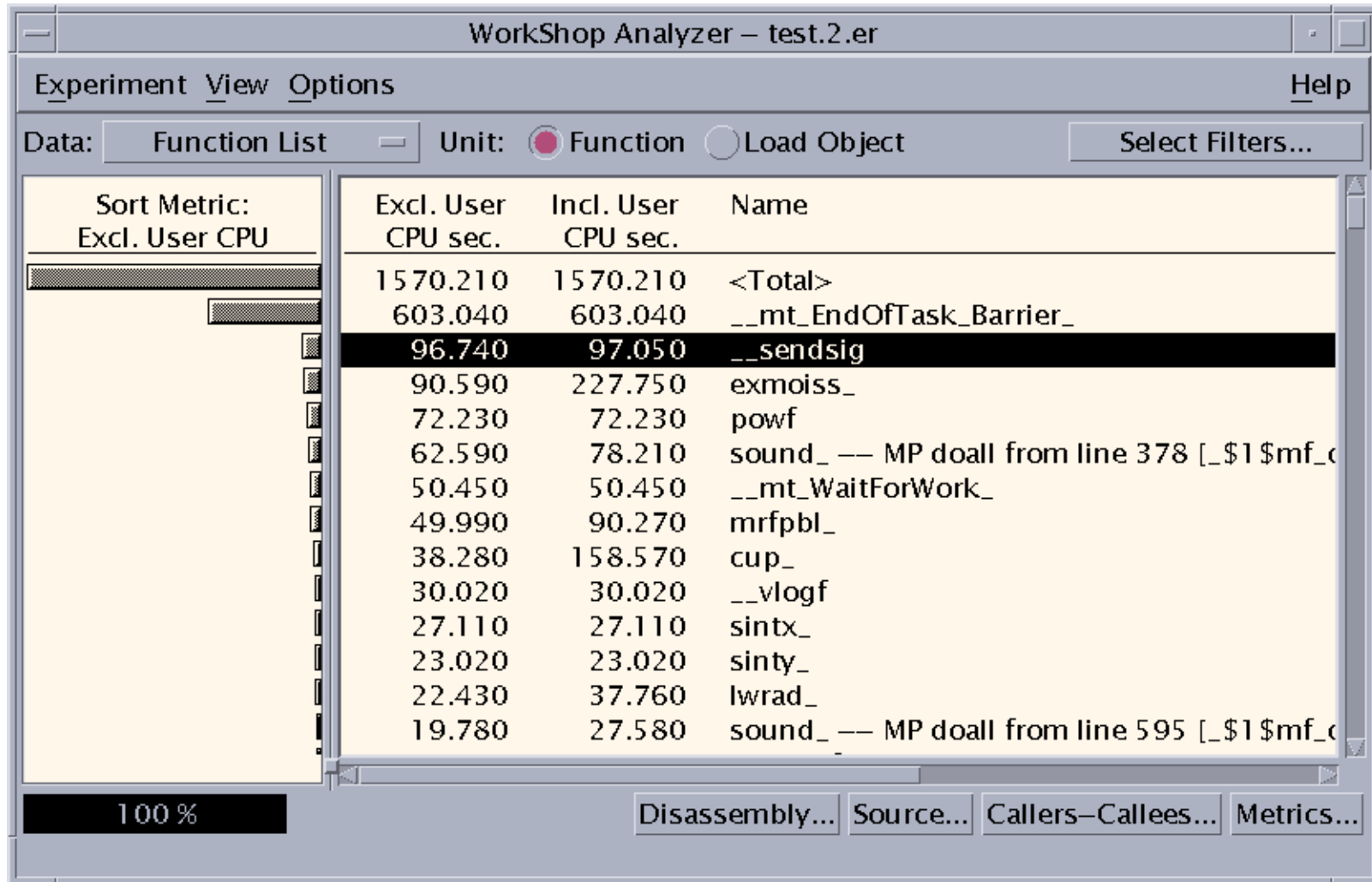
- MM5: mesoscale and regional scale atmospheric circulation model; see <http://www.mmm.ucar.edu/mm5/mm5-home.html>
- Code already parallelized using OpenMP directives
- MM5 V3.2, SOC data set
- 3-D space, iteration count $(I, J, K) = (49, 52, 23)$
- Loop-level parallelization, primarily in K
- Default scheduling on all loops

MM5 Timings

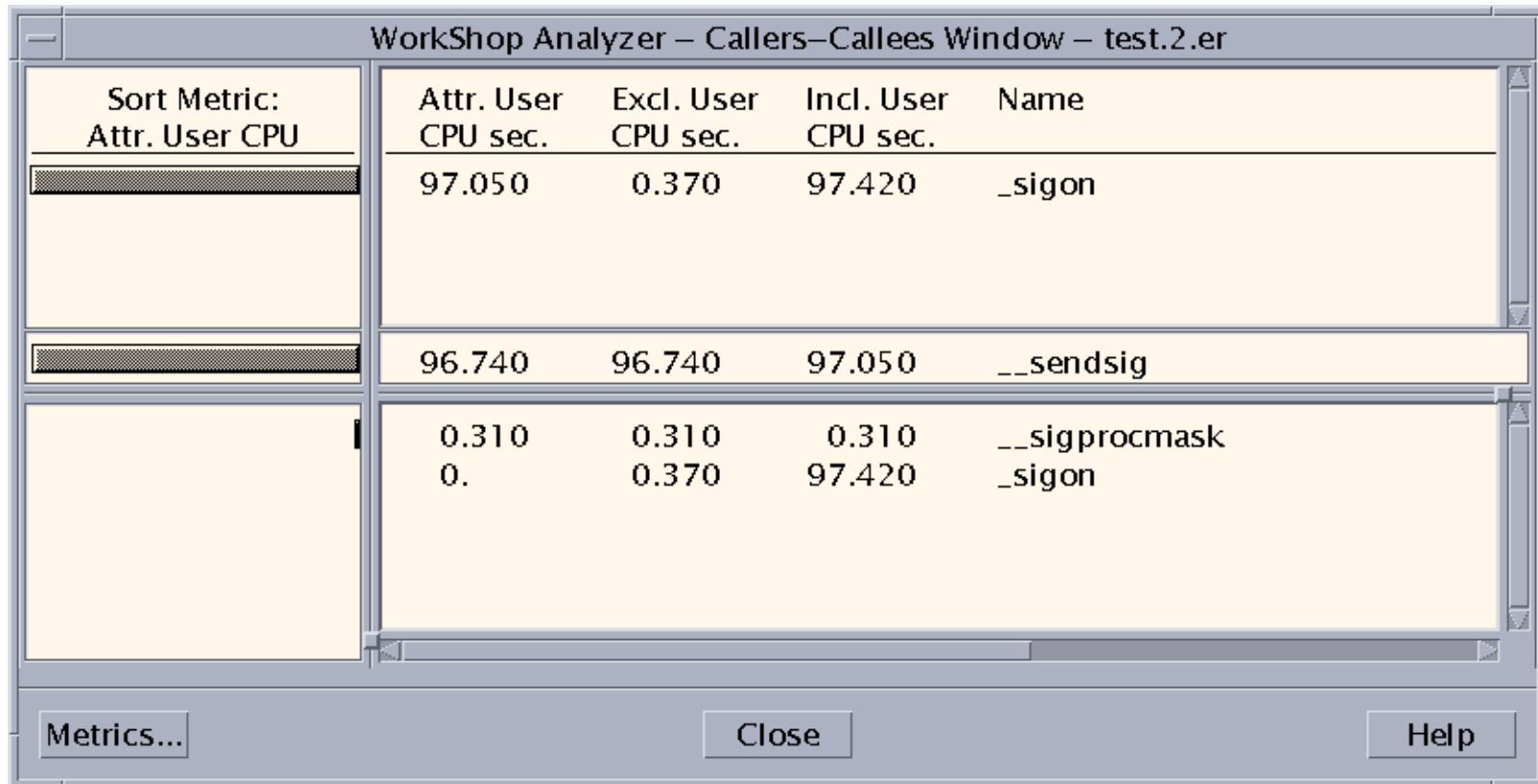
MM5 Scaling



Initial Profile



__sendsig Callstack



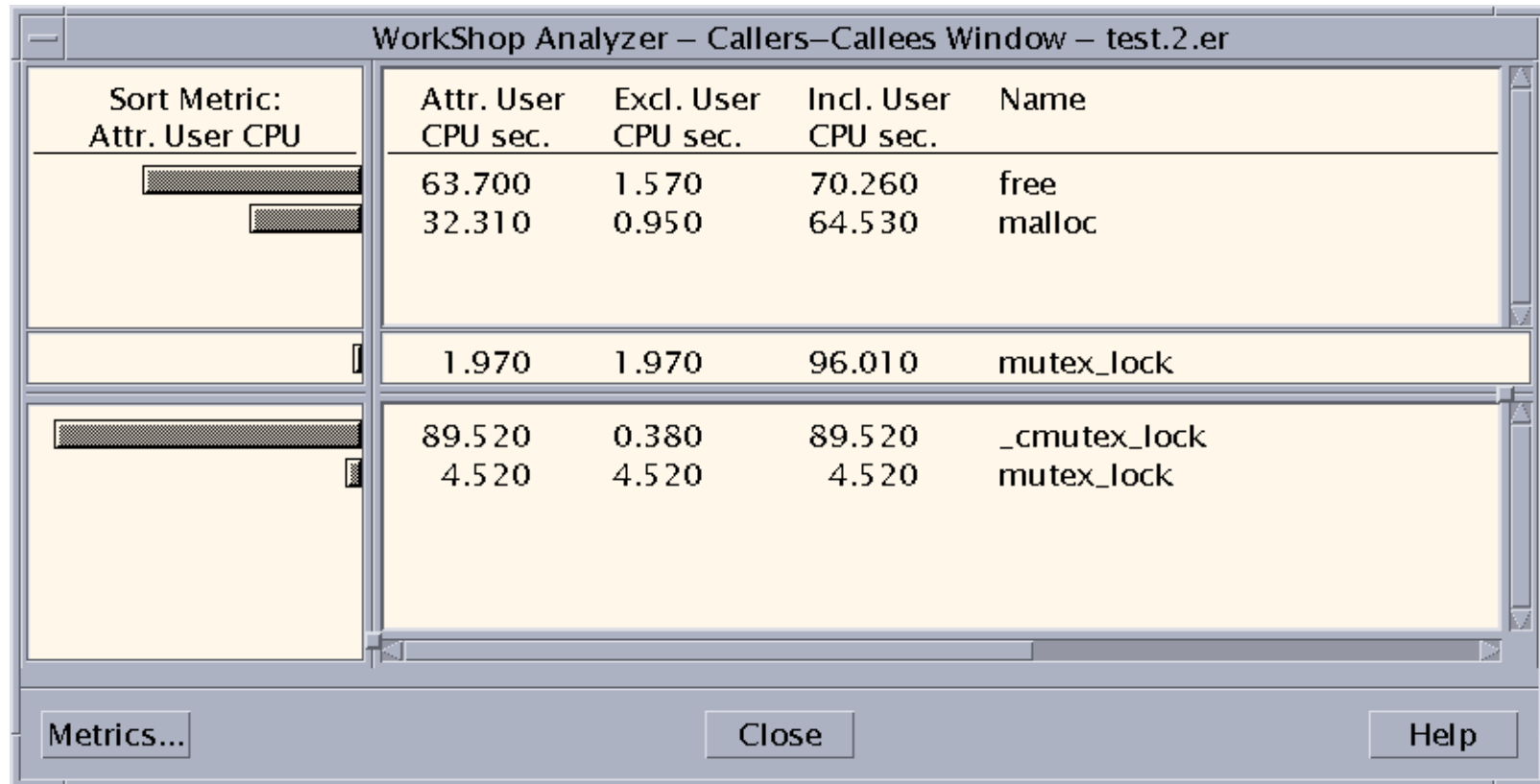
Workshop Analyzer – Callers–Callees Window – test.2.er

Sort Metric: Attr. User CPU	Attr. User CPU sec.	Excl. User CPU sec.	Incl. User CPU sec.	Name
	97.050	0.370	97.420	_signon
	96.740	96.740	97.050	__sendsig
	0.310	0.310	0.310	__sigprocmask
	0.	0.370	97.420	_signon

Metrics... Close Help

What the heck is __sendsig?

__send_sig is from malloc/free!



The screenshot shows a window titled "WorkShop Analyzer - Callers-Callees Window - test.2.er". On the left, there is a "Sort Metric:" section with "Attr. User CPU" selected. The main area is a table with the following data:

Attr. User CPU sec.	Excl. User CPU sec.	Incl. User CPU sec.	Name
63.700	1.570	70.260	free
32.310	0.950	64.530	malloc
1.970	1.970	96.010	mutex_lock
89.520	0.380	89.520	_cmutex_lock
4.520	4.520	4.520	mutex_lock

At the bottom of the window, there are three buttons: "Metrics...", "Close", and "Help".

Ah hah! Tracing up the stack shows `malloc/free`

solve source code

```
Vi - analyzer.source.15586.1;/tmp
File WorkShop Version
0.100      0.100      87.      IPH=1
0.010      0.010      88.      IPHO=1
0.670      0.670      89.      IF(T(I,K).LE.TCRIT)IPH=2
1.160      1.160      90.      IF(TN(I,K).LE.TCRIT)IPHO=2
0.940      0.940      91.      E=EXP(AE(IPH)-BE(IPH)/T(I,K))
→ ## 3.000  3.000  92.      EO=EXP(AE(IPHO)-BE(IPHO)/TN(I,K))
0.900      0.900      93.      QES(I,K)=.622*E/(100.*P(I,K)-E)
0.790      0.790      94.      QESO(I,K)=.622*EO/(100.*PO(I,K)-EO)
1.780      1.780      95.      IF(QES(I,K).LE.1.E-08)QES(I,K)=1.E-08
96.      IF(Q(I,K).GT.QES(I,K))Q(I,K)=QES(I,K)
97.      IF(QESO(I,K).LE.1.E-08)QESO(I,K)=1.E-08
98.      IF(QO(I,K).GT.QESO(I,K))QO(I,K)=QESO(I,K)
0.540      0.540      99.      TV(I,K)=T(I,K)+.608*Q(I,K)*T(I,K)
0.850      2.060      100.     TVO(I,K)=TN(I,K)+.608*QO(I,K)*TN(I,K)
101.      5  CONTINUE
102.
```

Look at source code of malloc/free caller
GUI highlights a call to EXP

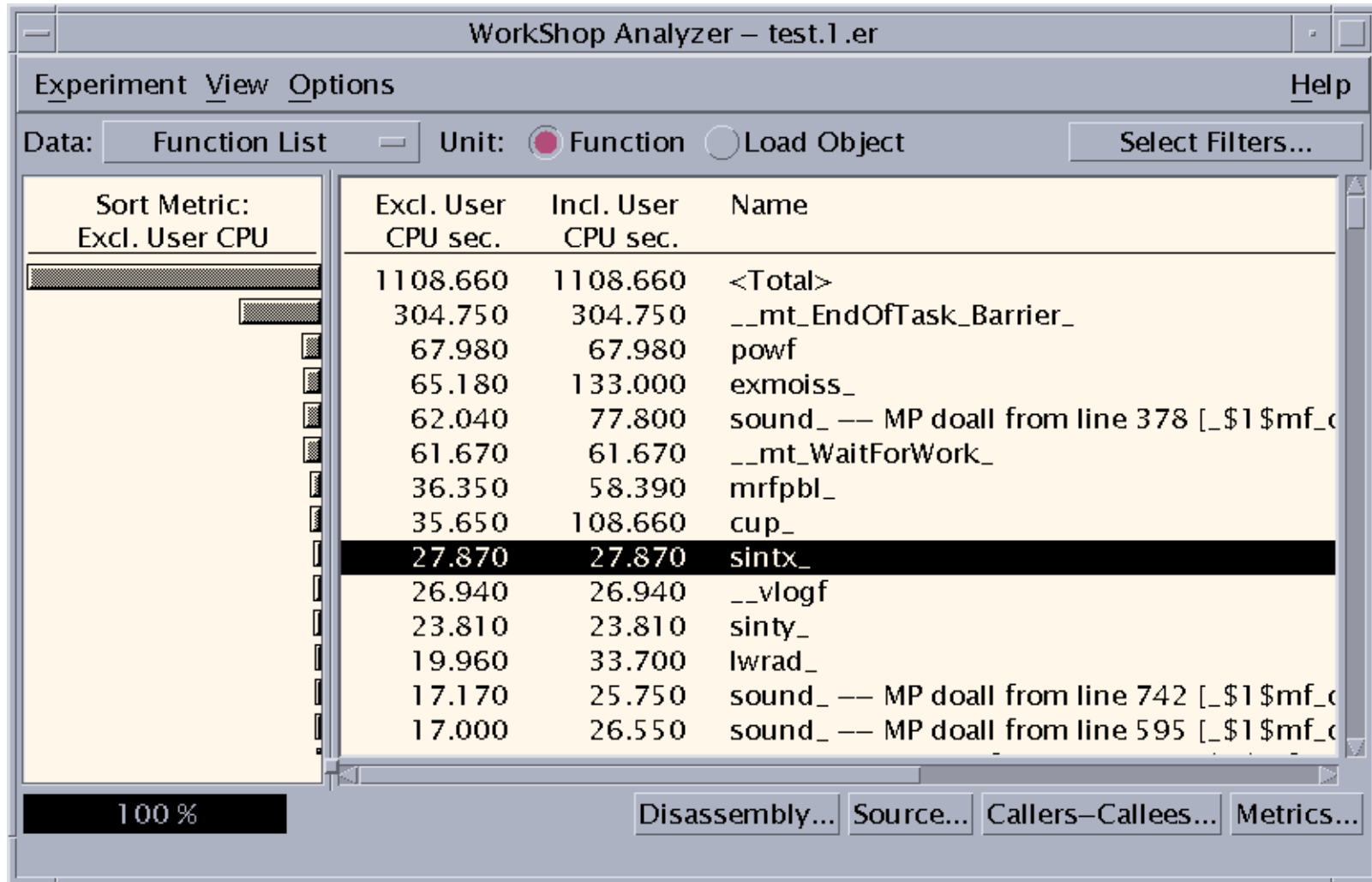
Intuition Time

- The tool made it easy to determine that `malloc/free` were bottlenecks
- Inspection of the code shows no `malloc/free` calls
- Ergo, compiler must be inserting them
- Leap of intuition: vectorization uses `malloc/free` (`EXP()` was the clue)
- Improved compiler annotation could help with this.

More Intuition

- Man page for malloc says it is MT-safe.
- Browsing through the call stack shows mutex calls. Ah hah! Must be locking the heap.
- Experiment with Analyzer's synchronization tracing; huge amount of time in mutex.
- Ask around – turns out there's a `libmtmalloc` available in Solaris.
- Run with that; big improvement (see graph).
- More collector/analyzer...

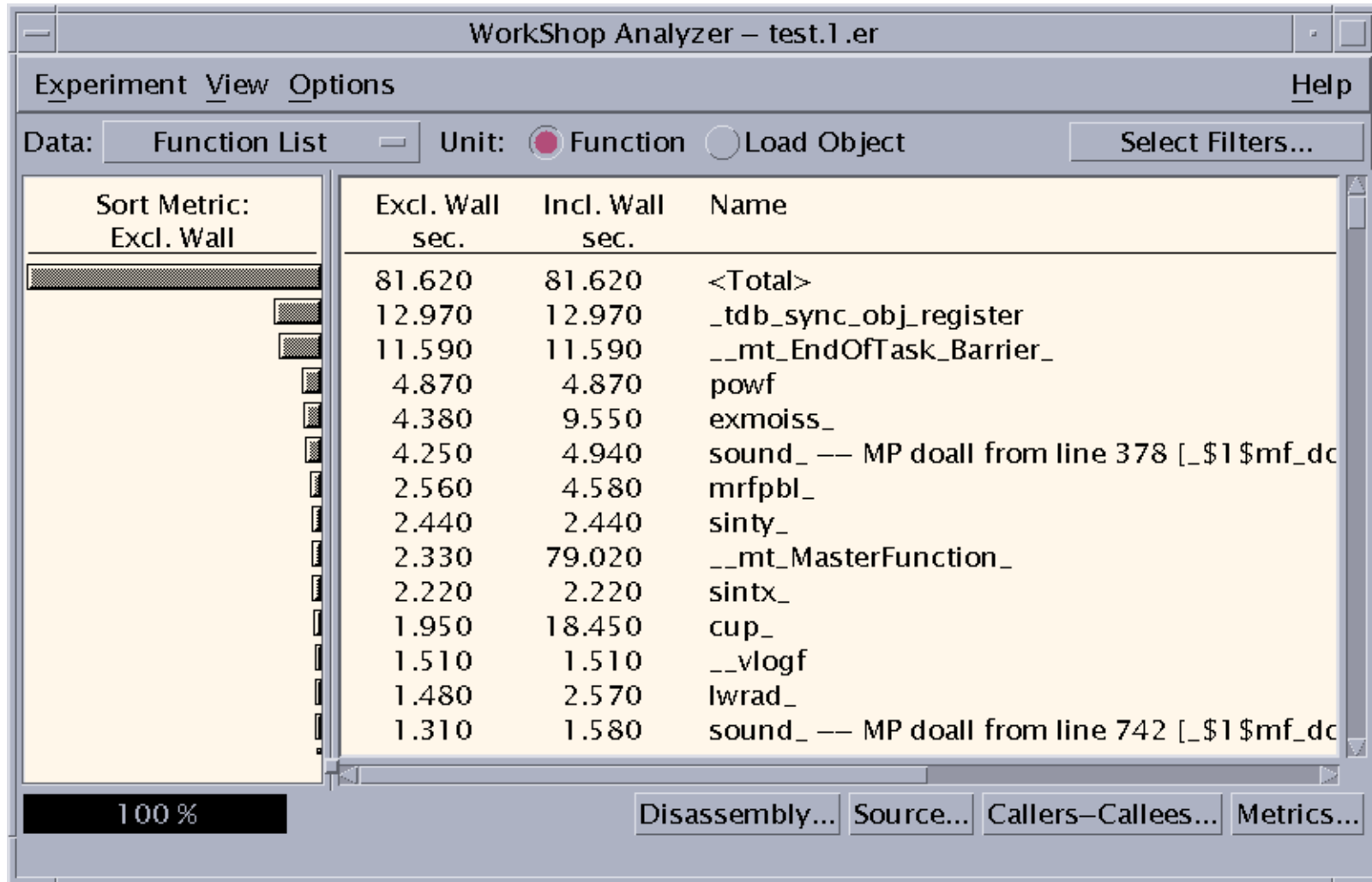
Profile with mtmalloc



But it still doesn't scale...

- The graph shows scaling still drops off at 8 cpus; moreover, it reverses from 16 to 32 cpus.
- However, the CPU profile doesn't show any obvious problems (aside from barrier).
- Hmm, we're thinking about wall-clock time; let's look at the wall clock profile...

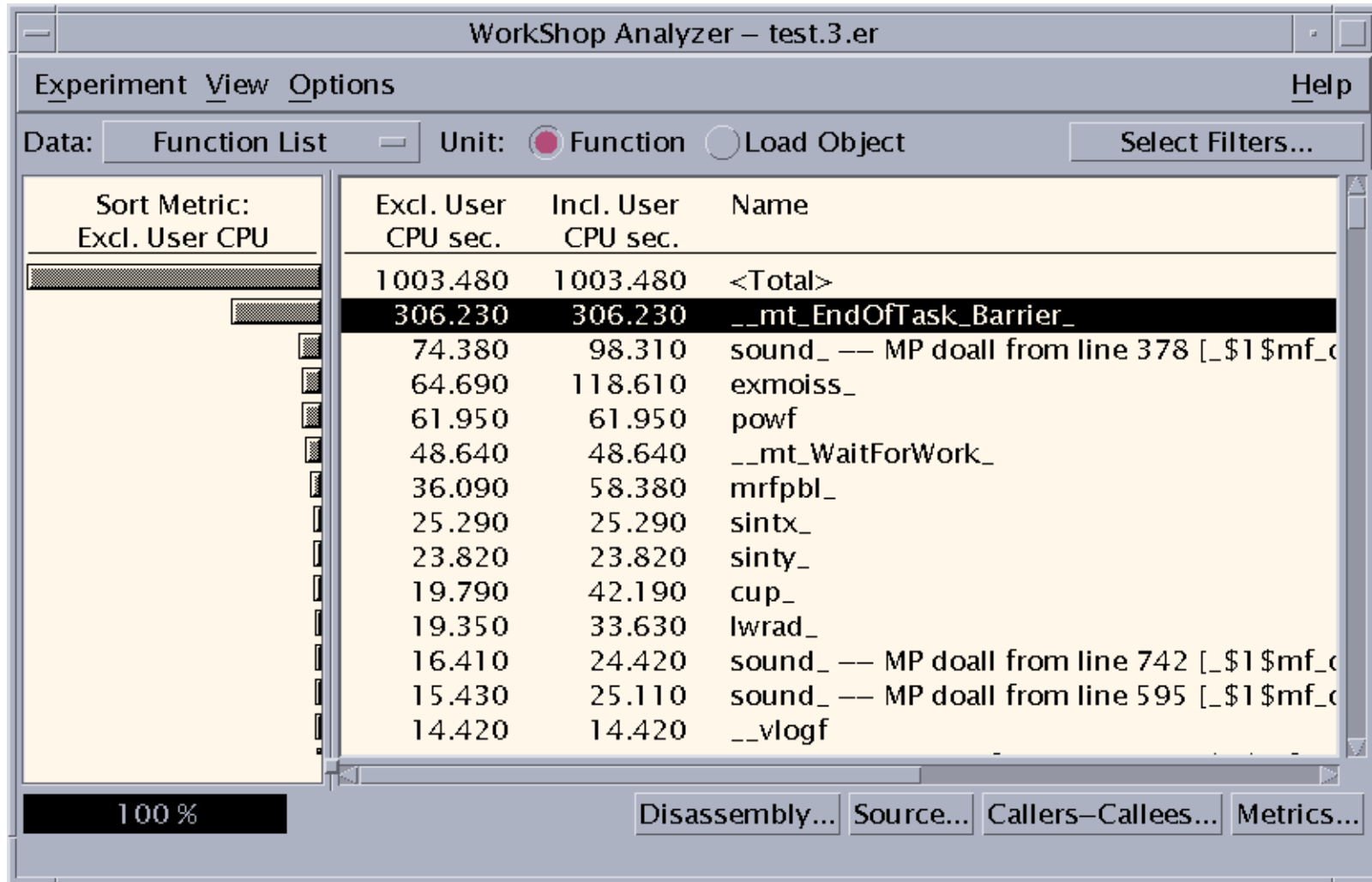
mtmalloc, showing wall time



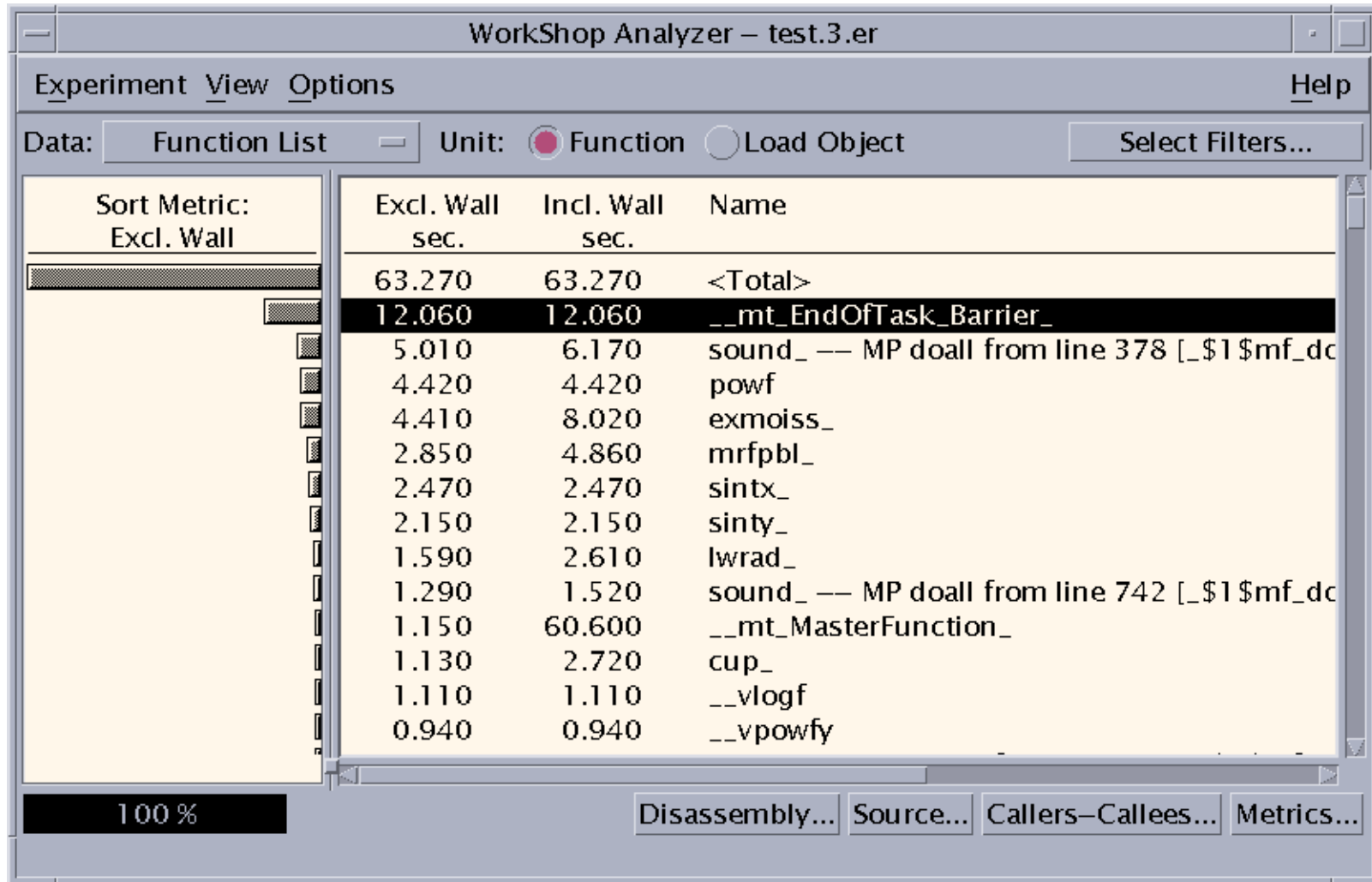
What the heck is that sync thing?

- Whatever it is, it sure takes a lot of time.
- Sparing the screen shots... it turns out that `malloc/free` are again the culprits.
- More asking around... Turns out that there's a Solaris version of `libhoard`, an MT-aware tuned malloc library: <http://www.hoard.org> .
- So, relink with *hoard*, and try again...

CPU time with hoard



Wall time with hoard



WorkShop Analyzer – test.3.er

Experiment View Options Help

Data: Function List Unit: Function Load Object Select Filters...

Sort Metric: Excl. Wall

Excl. Wall sec.	Incl. Wall sec.	Name
63.270	63.270	<Total>
12.060	12.060	__mt_EndOfTask_Barrier_
5.010	6.170	sound_ -- MP doall from line 378 [_\$1 \$mf_dc
4.420	4.420	powf
4.410	8.020	exmoiss_
2.850	4.860	mrfpbl_
2.470	2.470	sintx_
2.150	2.150	sinty_
1.590	2.610	lwrad_
1.290	1.520	sound_ -- MP doall from line 742 [_\$1 \$mf_dc
1.150	60.600	__mt_MasterFunction_
1.130	2.720	cup_
1.110	1.110	__vlogf
0.940	0.940	__vpowfy

100 %

Disassembly... Source... Callers-Callees... Metrics...

Barriers and Analyzer

- Barrier time should be attributed to parallel construct.
- Call stack shows parallel regions called from slave thread – not useful.
- Barrier function is called with address of outlined routine; analyzer constructs artificial callstack.
- Balance can be determined by filtering on thread – threads with more time in barrier did less work in construct.

Hoard call stack

Sort Metric: Attr. User CPU	Attr. User CPU sec.	Excl. User CPU sec.	Incl. User CPU sec.	Name
	23.010	0.010	81.400	solve_ -- MP doall from line 1125
	19.740	74.380	98.310	sound_ -- MP doall from line 378
	18.700	1.260	138.570	solve_ -- MP doall from line 1628
	14.690	10.920	46.880	stotndi_ -- MP doall from line 101
	306.230	306.230	306.230	__mt_EndOfTask_Barrier_

Metrics... Close

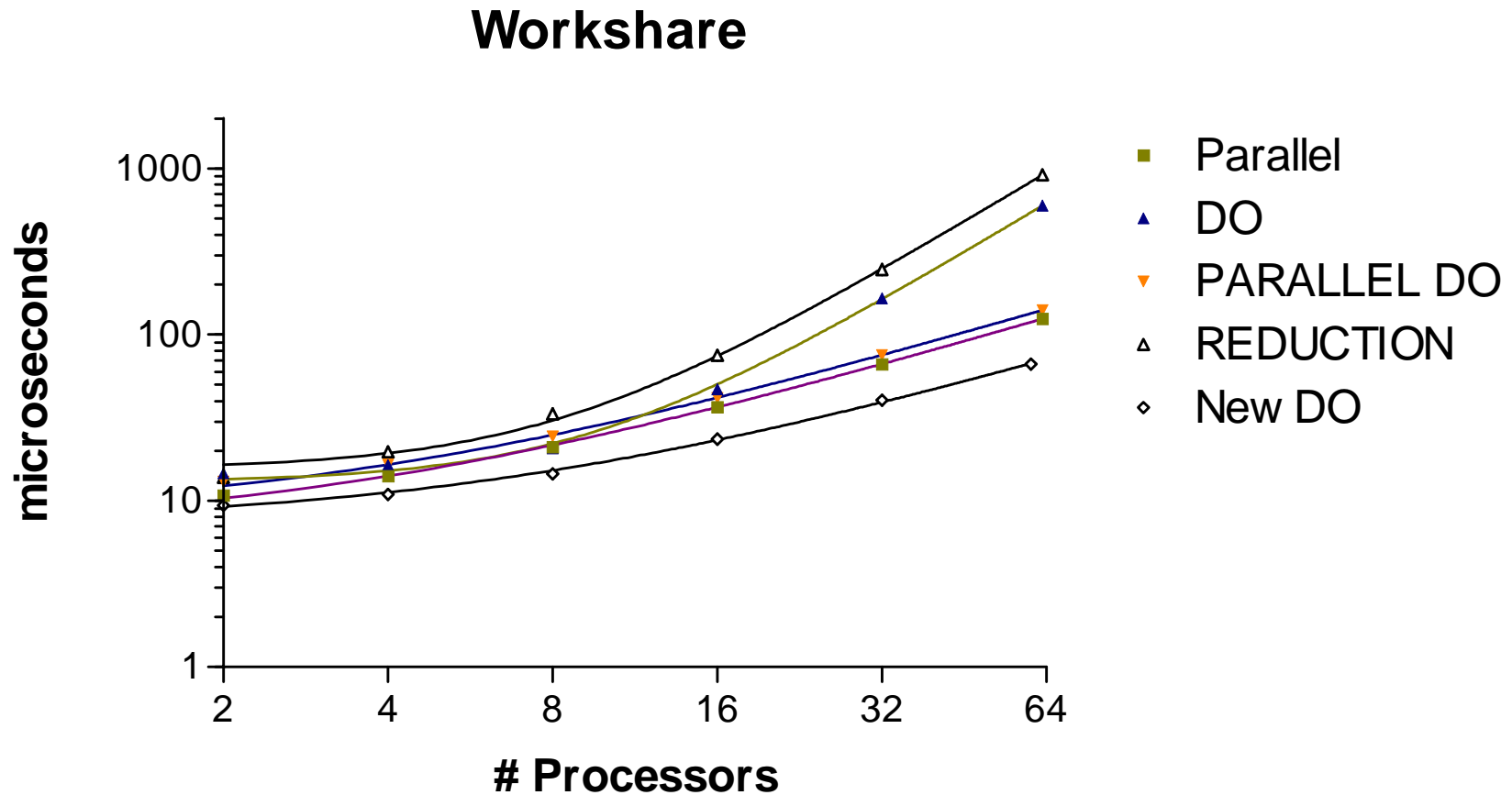
Quantization Effect

- Time spent in barrier (see previous slide) might be worth investigating, but...
- Recall that parallelization is loop-level only, and that the loops are iterated a relatively small number of times.
- Quantization effect: In a statically scheduled loop iterated N times, when speedup is plotted as a function of P (# processors), stair-steps occur when $\text{CEIL}(N/P)$ changes. This is obvious but is often overlooked.

EPCC Synch Benchmarks

- Platform: 64-cpu E10000, 400MHz cpus, 8MB Ecache
- “Relatively” quiescent (shared resource)
- MT_PROCESSOR_BIND set (accounts for lack of 64-cpu numbers)
- 1-cpu numbers discarded (special cased; useful for reality but not for data analysis)
- Only workshare constructs presented (data analysis not complete for others)

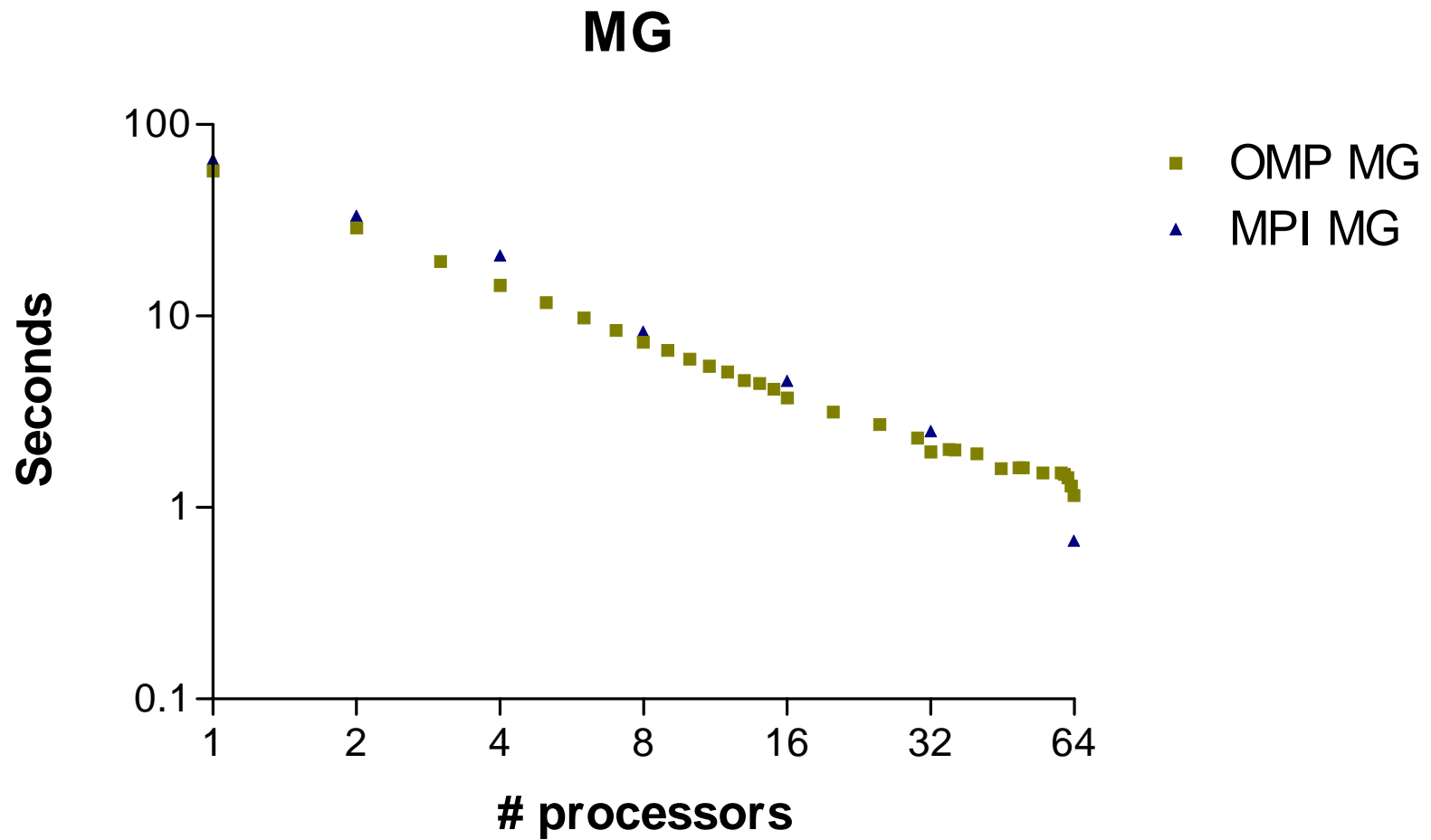
Workshare Constructs



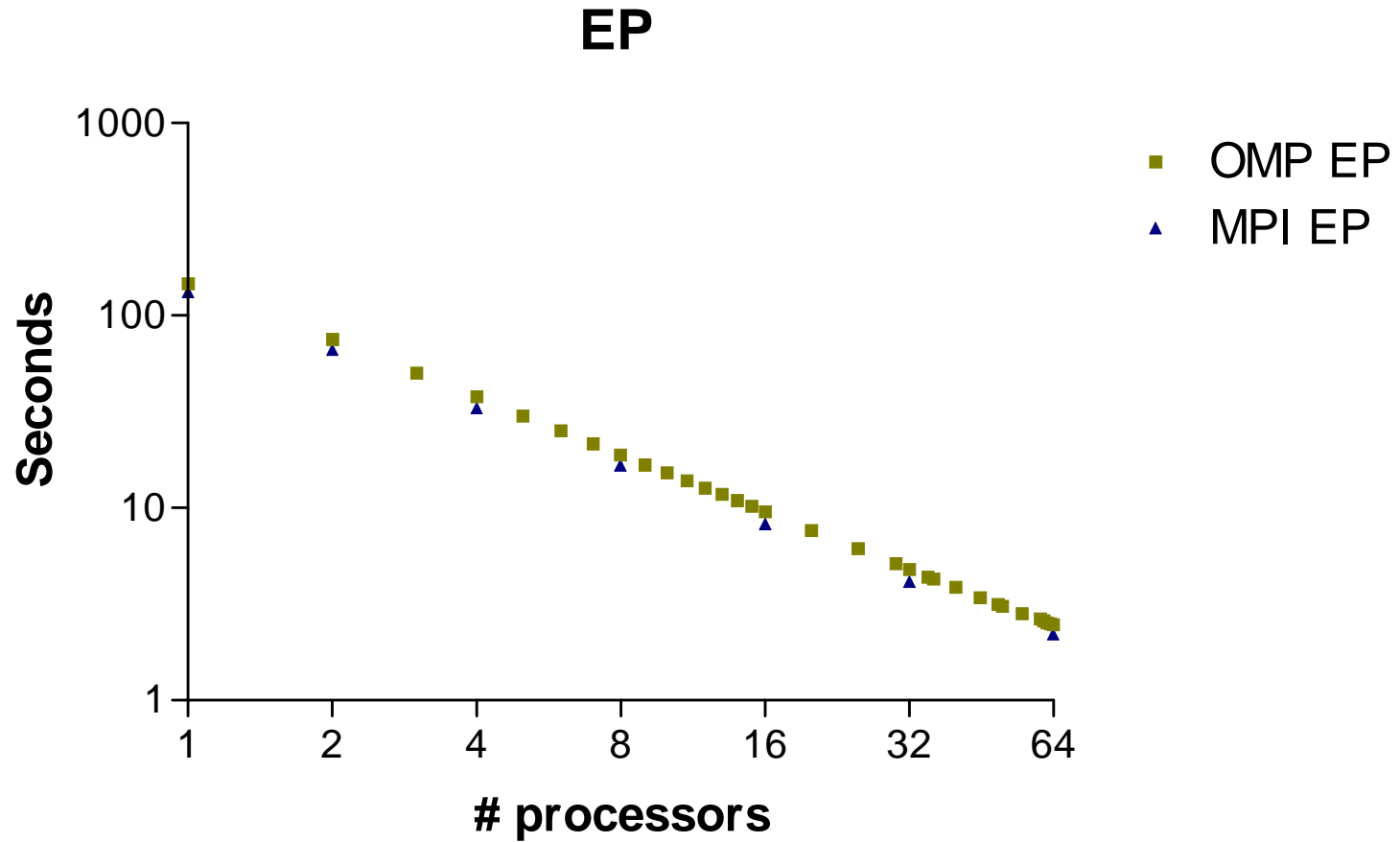
Nas Parallel Benchmarks

- Same platform, run conditions as EPCC benchmarks
- 1-cpu numbers included
- Plotted vs. MPI numbers on same platform
- MPI codes solve same problem but use different algorithm
- Benchmarks: EP, MG, CG, FT
- Pseudo-applications: SP, BT, LU
- IS not included (no Fortran implementation)

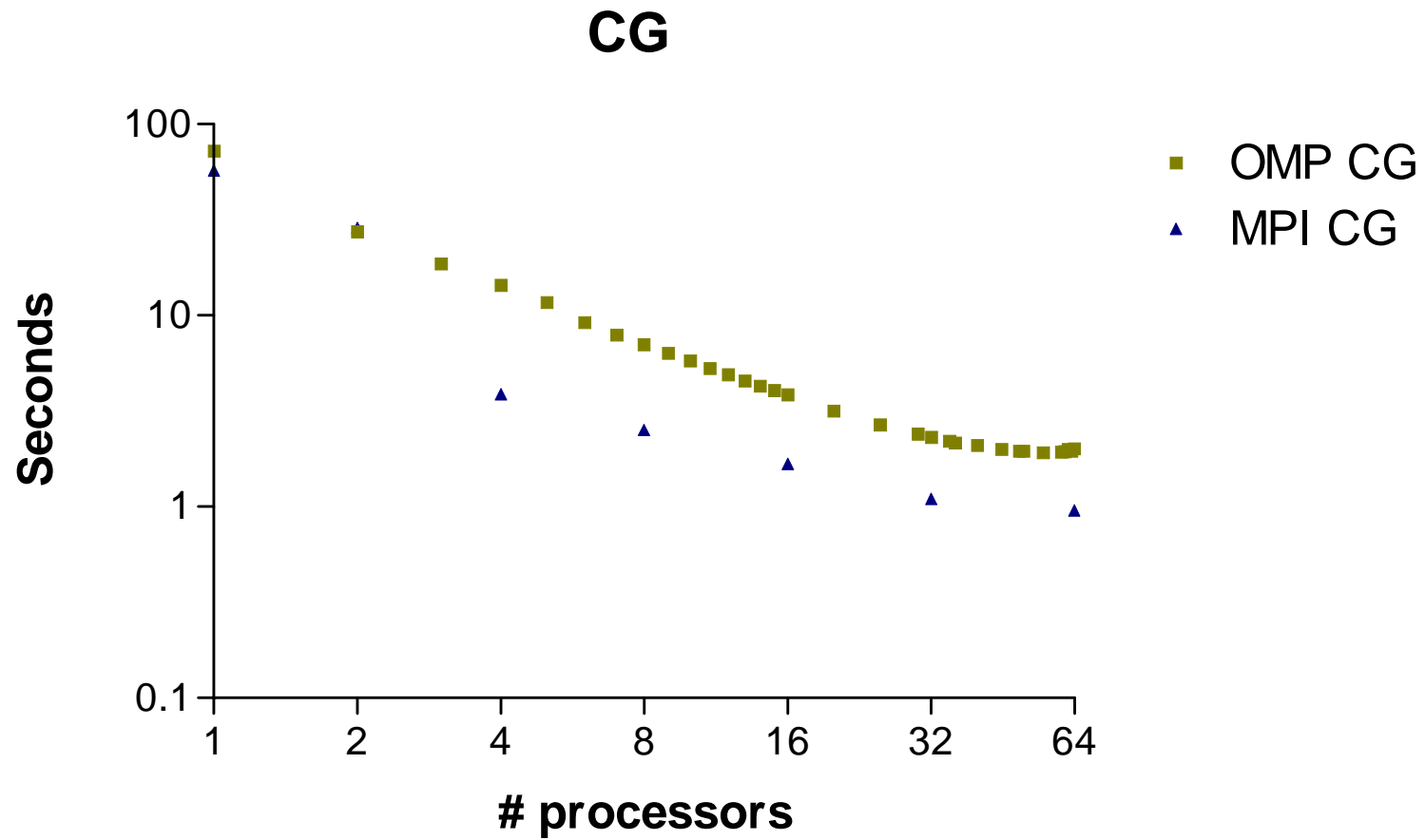
MG



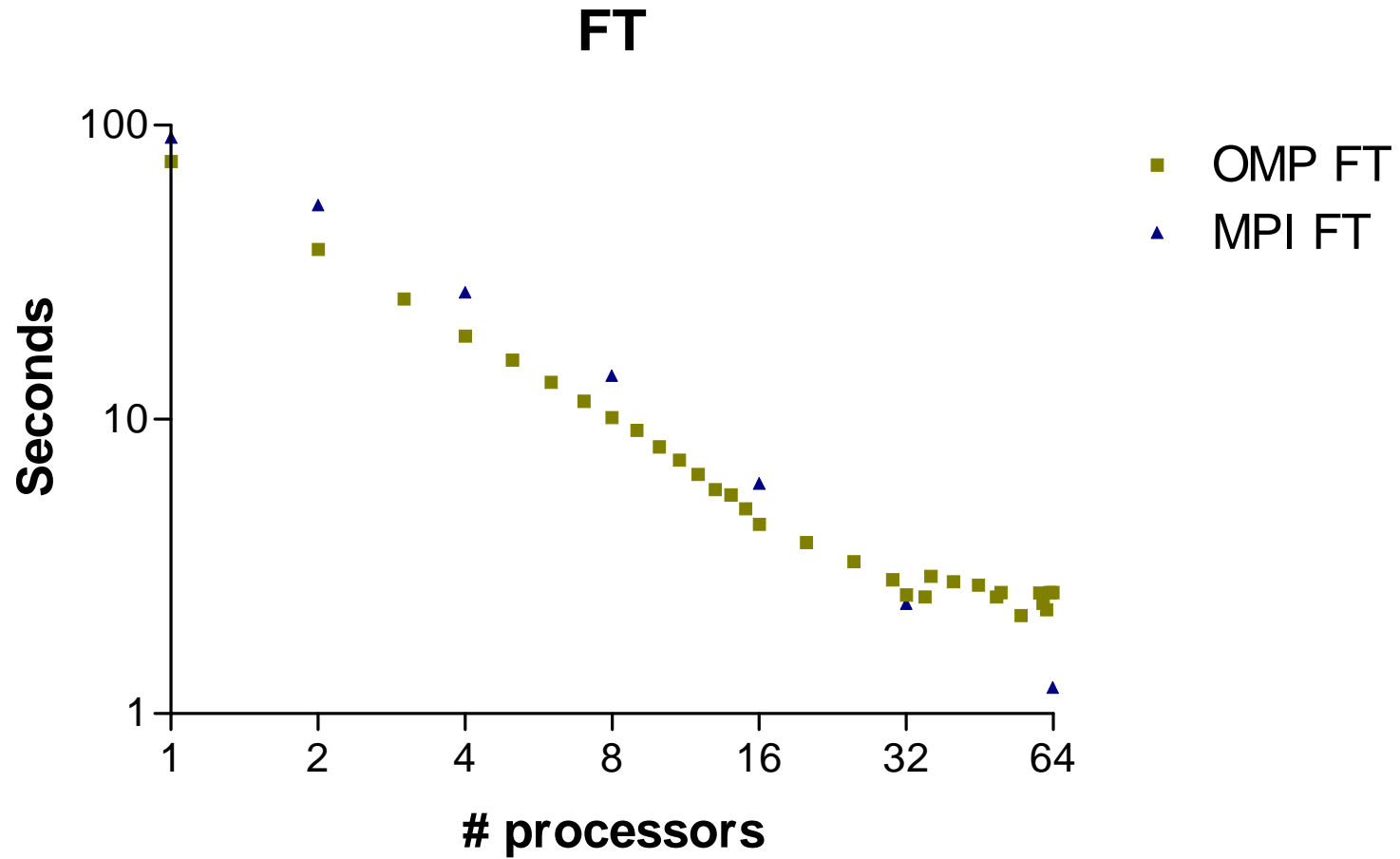
EP



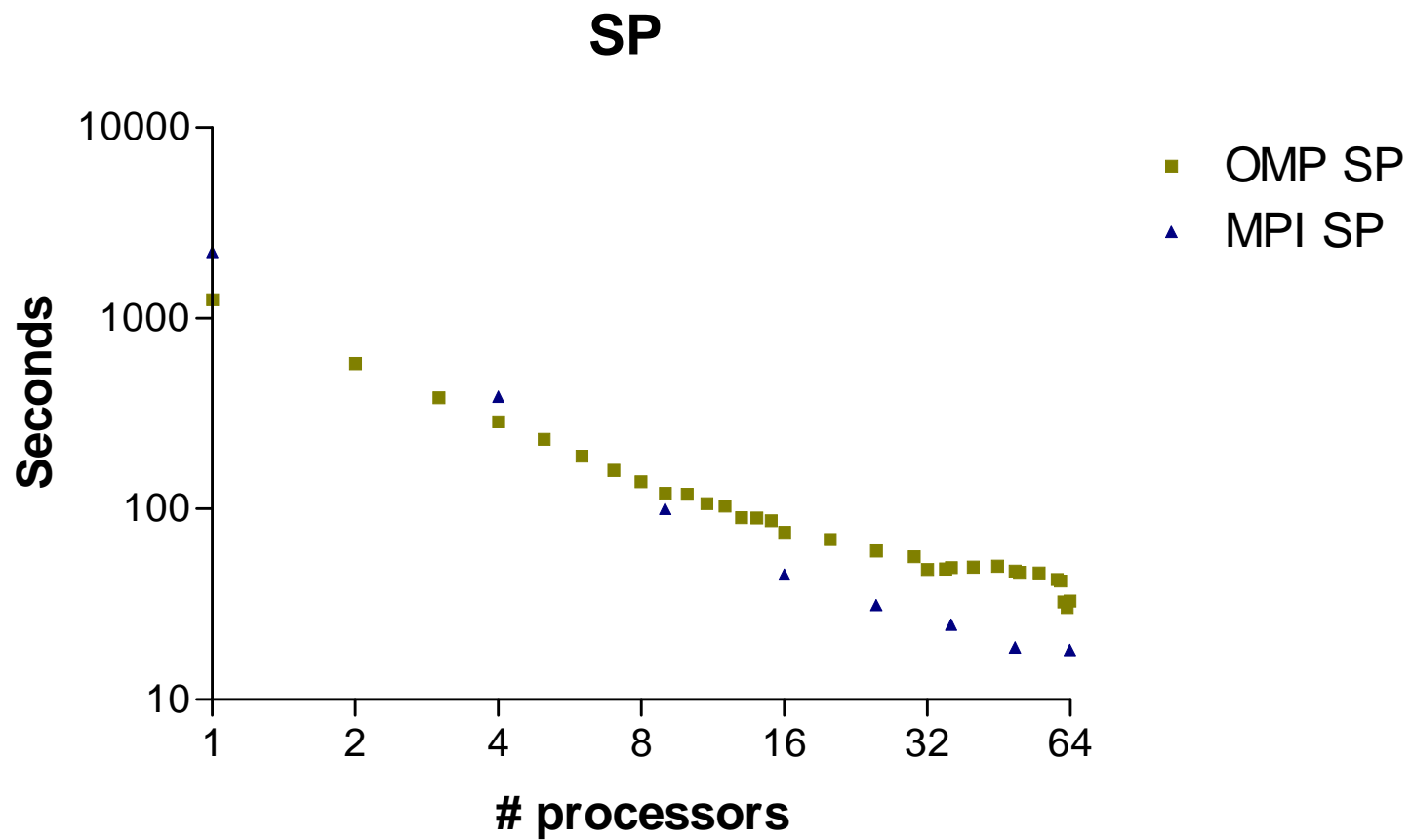
CG



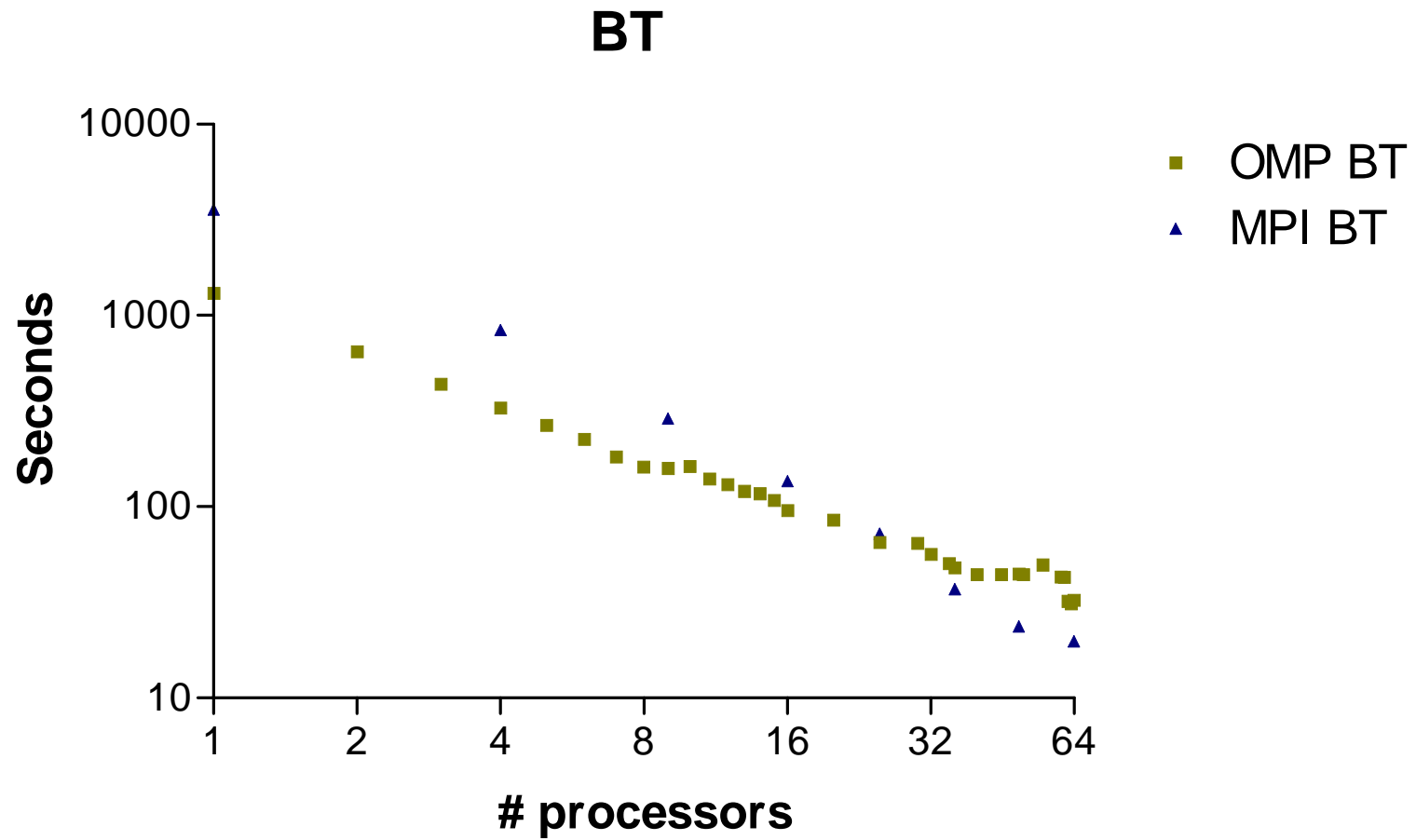
FT



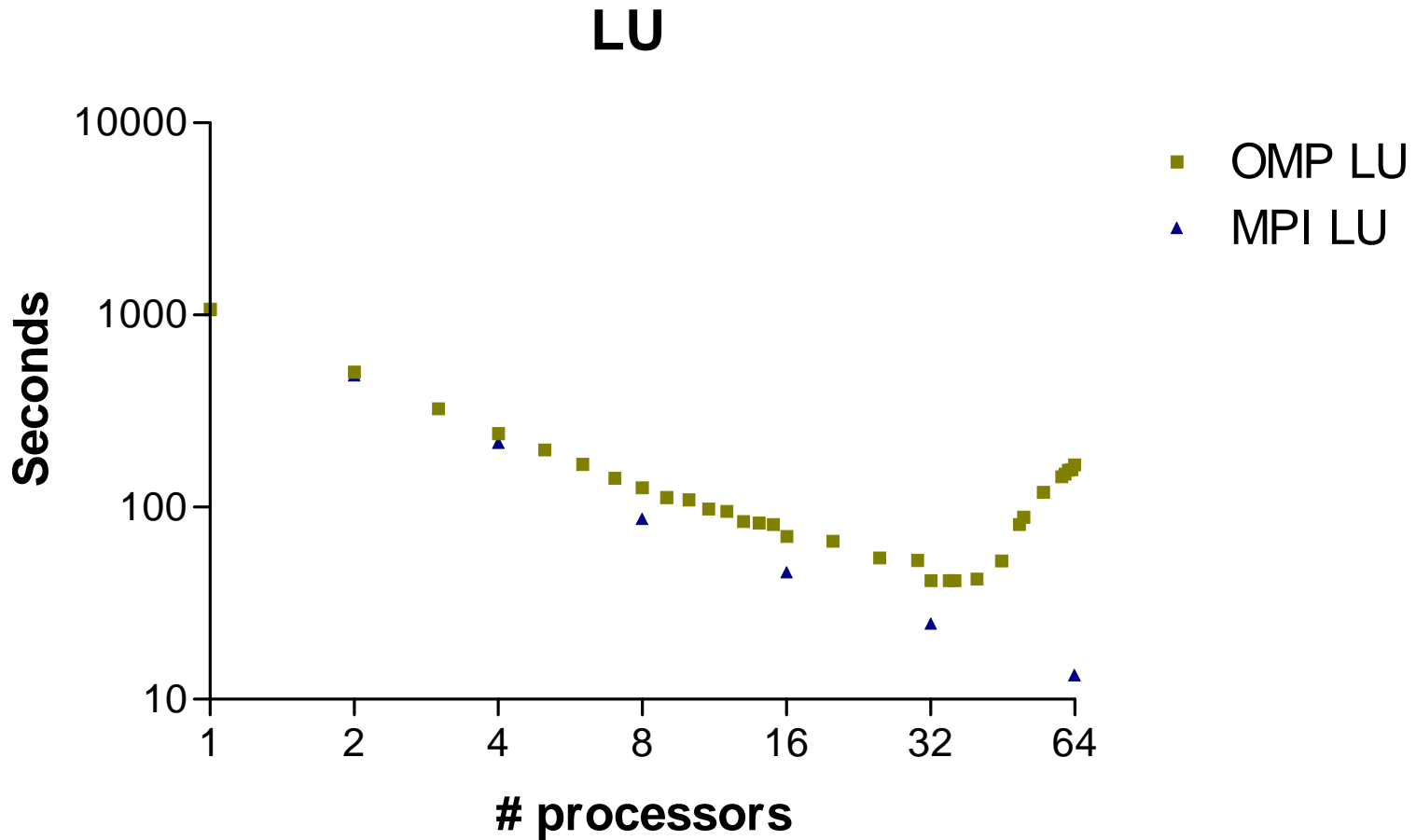
SP



BT



LU (!)



LU scaling

- LU is a pipelined parallel algorithm
- Written like a SPMD code
- Note stair-step effect up to 40 or so cpus
- Possible contention on user locks at >40:

```
iam = omp_get_thread_num()
if (iam .gt. 0 .and. iam .le. mthreadnum) then
    neigh = iam - 1
    do while (isync(neigh) .eq. 0)
!$omp flush(isync)
        end do
        isync(neigh) = 0
!$omp flush(isync)
    endif
```