

LA-UR-24-28965

Approved for public release; distribution is unlimited.

Title: MCNP6 Parallel Performance Analysis: How to Efficiently Run MCNP6 in Parallel

Author(s): Bull, Jeffrey S.

Intended for: 2024 MCNP User Symposium, 2024-08-19/2024-08-14 (Los Alamos, New Mexico, United States)

Issued: 2024-08-29 (rev.1)



Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Triad National Security, LLC for the National Nuclear Security Administration of U.S. Department of Energy under contract 89233218CNA000001. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.



MCNP6 Parallel Performance Analysis: How to Efficiently Run MCNP6 in Parallel

Jeffrey Bull
XCP-3, MCNP Development Team

2024 MCNP® User Symposium

August 21, 2024

LA-UR-24-28965

MCNP[®]
MONTE CARLO N-PARTICLE

Introduction

To take advantage of multi-core computer architecture, MCNP6 provides two independent methods to run problems in parallel: task-based threading using OpenMP and distributed processing supported by the Message Passing Interface (MPI).

How to setup and run MCNP6 in parallel depends on several factors, including the computing hardware as well as the problem to be run. This presentation will discuss how MCNP6 runs in parallel, present the results of several sample problems with the goal of providing some insight on how to run MCNP in parallel effectively.

Comparison of Threading and MPI in MCNP6

	Threading	MPI
Ease of use	Integrated into MCNP	Requires additional software
Versatility	Limited to neutron, photon, and electron particles only -- No model physics	Works with all MCNP features
CPU utilization	All processes transport particles	Manager process only sends and collects data; it does no particle transport
Memory usage	Geometry and cross section data memory is shared	No memory shared
CPU overhead	Requires thread locks so that only one thread at a time can run certain sections of the code	Each process runs independently
Inter-process data transfer and collection	None – done internally	Manager broadcasts data to individual processes and collects the results when they are finished.
Number of parallel processes	Limited to one core	None -- can run across nodes of a cluster or computers on a network

Except for UM preprocessing, only particle transport is run in parallel.

Sample Problem: Godiva

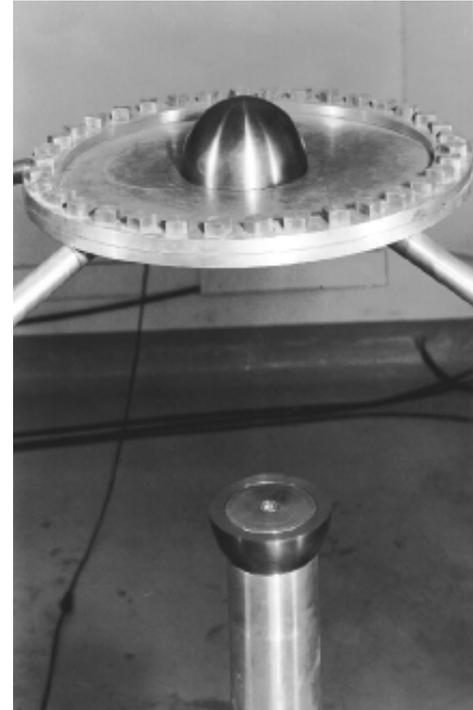
```
Godiva Solid Bare HEU sphere HEU-MET-FAST-001
1      1      4.7984e-02      -1      imp:n=1
2      0      1      1      imp:n=0

1      so      8.7407

prdmp j -1e6 j 1 1e9
sdef  cel=1      erg=d1      rad=d2      pos=0.0 0.0 0.0
sp1   -3
si2   0.0      8.7407
sp2   -21      2
totnu

c ----- ENDF/B-VII -----
m1    92234.70c  4.9184e-04
      92235.70c  4.4994e-02
      92238.70c  2.4984e-03

c -----
print
kcode 10000 1.0 80 800
```



Run godiva

```
time mcnp6 i= godiva.inp
```

```
.....
```

```
source distribution written to file srctq
```

```
cycle= 800
```

```
run terminated when 800 kcode cycles were  
done.
```

```
=====> 409.14 M histories/hr (based on wall-  
clock time in mcrun)
```

```
comment.
```

```
comment. Average fission-source entropy for the last  
half of cycles:
```

```
comment. H= 7.42E-01 with population std.dev.=  
1.40E-03
```

```
comment.
```

```
comment.
```

```
comment. Cycle 17 is the first cycle having fission-  
source
```

```
comment. entropy within 1 std.dev. of the average  
comment. entropy for the last half of cycles.  
comment. At least this many cycles should be  
discarded.  
comment. Source entropy convergence check passed.  
comment.
```

```
final k(col/abs/trk len) = 0.999574 std dev =  
0.000217
```

```
ctm = 1.17 nrn = 411408359  
dump 2 on file runtpf.h5 nps = 8001005 coll =  
29312333  
mcrun is done
```

```
real 1m12.282s  
user 1m5.402s  
sys 0m6.060s
```

Run godiva with 2 threads

```
time mcnp6 i= godiva.inp tasks 2
. . . . .
source distribution written to file srctq      cycle=
800
run terminated when      800 kcode cycles were done.

=====>      815.62 M histories/hr      (based on wall-clock
time in mcrun)

comment.
comment. Average fission-source entropy for the last
half of cycles:
comment.      H= 7.42E-01 with population std.dev.=
1.40E-03
comment.
comment.
comment. Cycle 17 is the first cycle having fission-
source
```

```
comment. entropy within 1 std.dev. of the average
comment. entropy for the last half of cycles.
comment. At least this many cycles should be
discarded.
comment.
comment. Source entropy convergence check passed.
comment.
```

```
final k(col/abs/trk len) = 0.999574      std dev =
0.000217
```

```
ctm =      1.15      nrn =      411408359
dump 2 on file runtpg.h5      nps =      8001005      coll =
29312333
mcrun is done
```

```
real      0m36.655s
user      1m8.906s
sys       0m3.052s
```

Godiva Timing Results for Different Numbers of Tasks

Number of Tasks

1	=====>	409.14 M histories/hr
2	=====>	815.62 M histories/hr
4	=====>	1353.13 M histories/hr
8	=====>	2029.36 M histories/hr

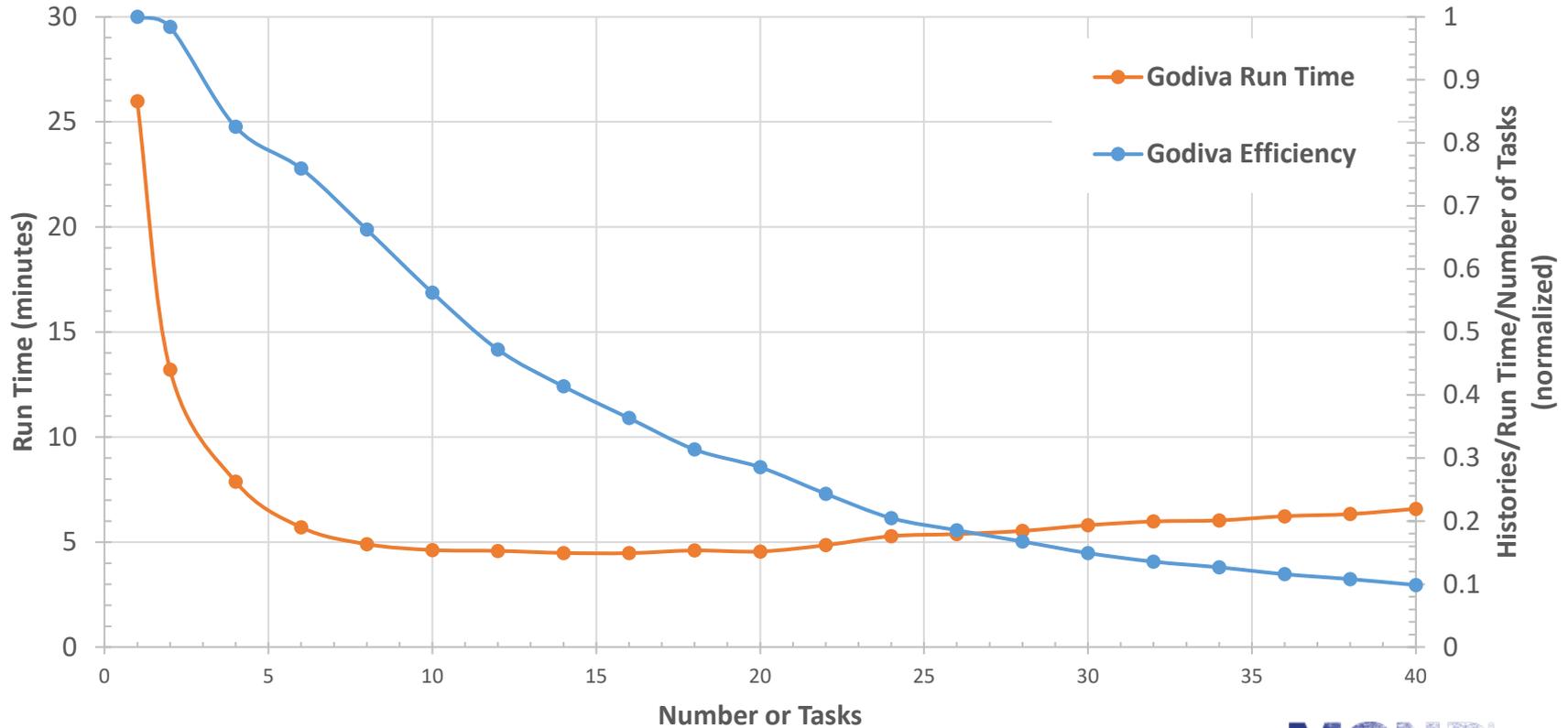
Number of Tasks

1	real	1m12.282s
2	real	0m36.655s
4	real	0m22.637s
8	real	0m15.561s

Number of Tasks

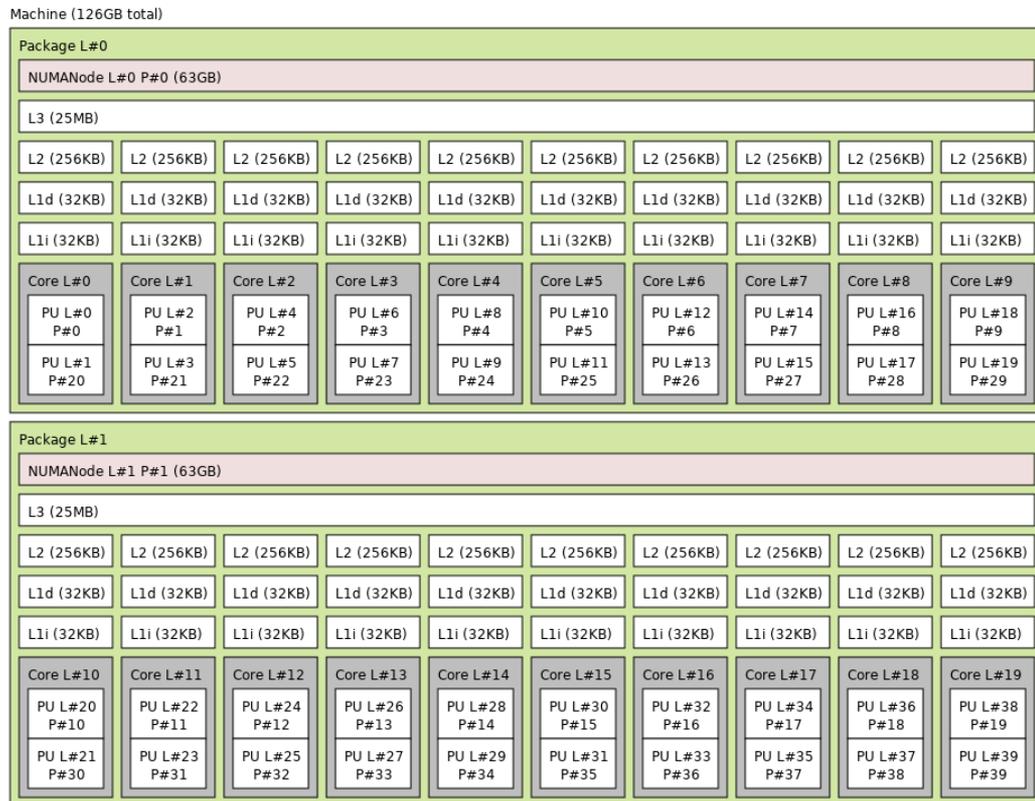
1	ctm =	1.17	nrn =	411408359
2	ctm =	1.15	nrn =	411408359
4	ctm =	1.36	nrn =	411408359
8	ctm =	1.79	nrn =	411408359

Godiva Threading Performance



Computer Hardware Used For This Analysis

- Linux RedHat 8.10
- Intel Xeon(R) CPU E5-2600 v3, Haswell
 - 2 sockets
 - 10 cores/socket
 - Hyperthreading on
 - 20 logical CPUs/socket
 - 2 NUMA nodes, one per socket



Intel VTune Profiler

Elapsed Time: 29.819s

Hotspots Insights
 If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee or the Flame Graph views to track critical paths for these hotspots.

Explore Additional Insights
 Parallelism: 13.5%
 Use % Threading to explore more opportunities to increase parallelism in your application.
 Microarchitecture Usage: 0.0%
 Use % Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.

Top Hotspots
 This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
__kmpc_set_lock	libomp5.so	41.679s	18.1%
accolt	mcnp6	24.590s	10.7%
accolt	mcnp6	15.350s	6.7%
colidd	mcnp6	12.830s	5.6%
colidd	mcnp6	11.500s	5.0%
[Others]		123.750s	53.9%

*NA is applied to non-summable metrics

Effective CPU Utilization Histogram
 This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the idle CPU utilization value.

Collection and Platform Info
 This section provides information about this collection, including result set size and collection platform data.

Application Command Line: iwashomejsbuilg_mcp6mcnp6\$sourcebuilg_profilemcp6 i-godiva inp n-godiva_hst1 tasks 8
 Operating System: 4.18.0-553.8.1.el8_10.x86_64 Red Hat Enterprise Linux release 8.10 (otop)
 Computer Name: cn132
 Result Size: 15.3 MB
 Collection start time: 16:50:29 07/08/2024 UTC
 Collection stop time: 16:50:57 07/08/2024 UTC
 Collector Type: Event-based counting driver/User-mode sampling and tracing
 Finalization mode: Fast. If the number of collected samples exceeds the threshold, this mode limits the number of processed samples to speed up post-processing.

CPU
 Name: Intel(R) Xeon(R) E5/E7 v3 Processor code named Haswell
 Frequency: 2.6 GHz
 Logical CPU Count: 40
 L1L2 size: 26.2 MB

Cache Allocation Technology
 Level 2 capability: not detected
 Level 3 capability: not detected

Flame Graph
 A visualization showing the call stack of the application. The x-axis represents time, and the y-axis represents the call stack. The graph shows various functions and their interactions, with colors representing different modules.

Call Stacks
 18.0% (41.644s of 229.740s)

```

libomp5.so | __kmpc_set_lock - kmp_lock.h
mcnp6 | threading_lock_on+0x6 - smmp.F90.119
mcnp6 | tmspt+0x3d - tmspt.F90.357
mcnp6 | mcrun_somp$parallel@109+0xac - mcrun.F90.110
libomp5.so | [OpenMP dispatcher]@0x132 - kmp_runtime.cpp.7845
libomp5.so | __kmpc_forall@0x7e1 - kmp_runtime.cpp.2300
libomp5.so | [OpenMP fork]@0x1a2 - kmp.h.350
mcnp6 | mcrun+0x32f - mcrun.F90.109
mcnp6 | main+0x1951 - main.F90.329
mcnp6 | main+0xc
libc.so.6 | __libc_start_main+0xe4
mcnp6 | start+0x2e
    
```

Thread
 Running
 CPU Time
 Spin and Overhead
 CPU Sample
 CPU Utilization

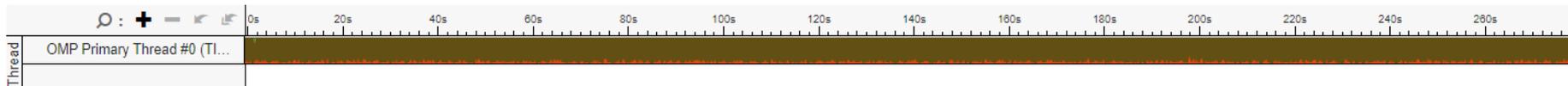
FILTER: 100.0% Any Process Thread Any Thread Module Any Module Any Utilization User functions + 1 Functions only Show inline functions

Godiva CPU Cycle Breakdown

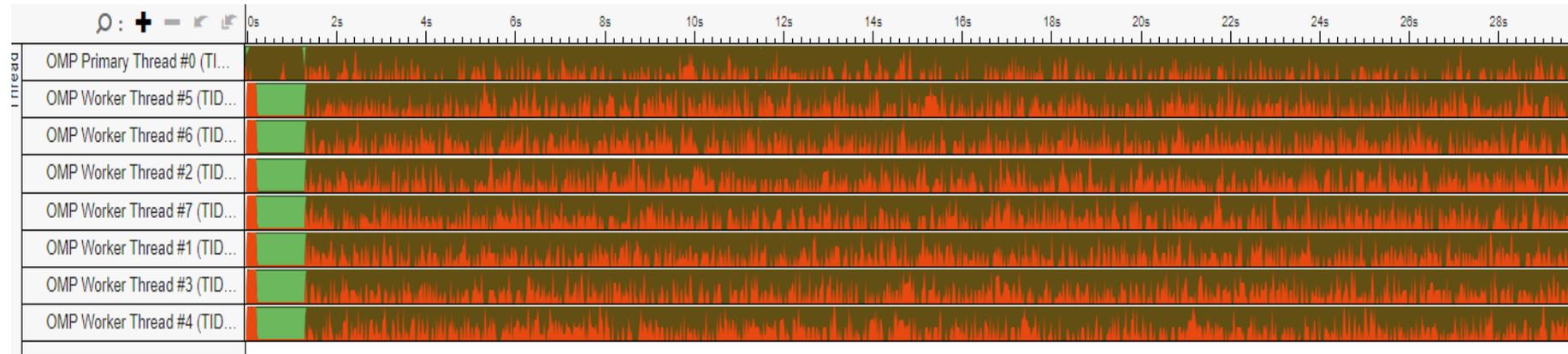
Top-level CPU Time (seconds)	Number of Tasks		
	1	8	40
Total CPU Time	278	230	2,693
Time spent executing MCNP	261	161	268
Time threads are locked	2	24	774
Time spent managing the threads (overhead)	14	44	1,650
Effective CPU utilization	1.0	5.4	3.5
Elapsed Time	280	30	77

How is MCNP Using CPU Cycles

1 task



8 tasks



How is MCNP Using CPU Cycles – 40 Tasks



■ Running MCNP ■ Idle ■ Spin and overhead

What's Causing the Spin and Overhead Time?

1 task

Function	CPU Time (s)	% of CPU Time
acetot	30	10.80%
uname	25	9.10%
acecol	24	8.80%
getrusage	22	7.90%
colidn	17	6.10%
[Others]	159	57.30%

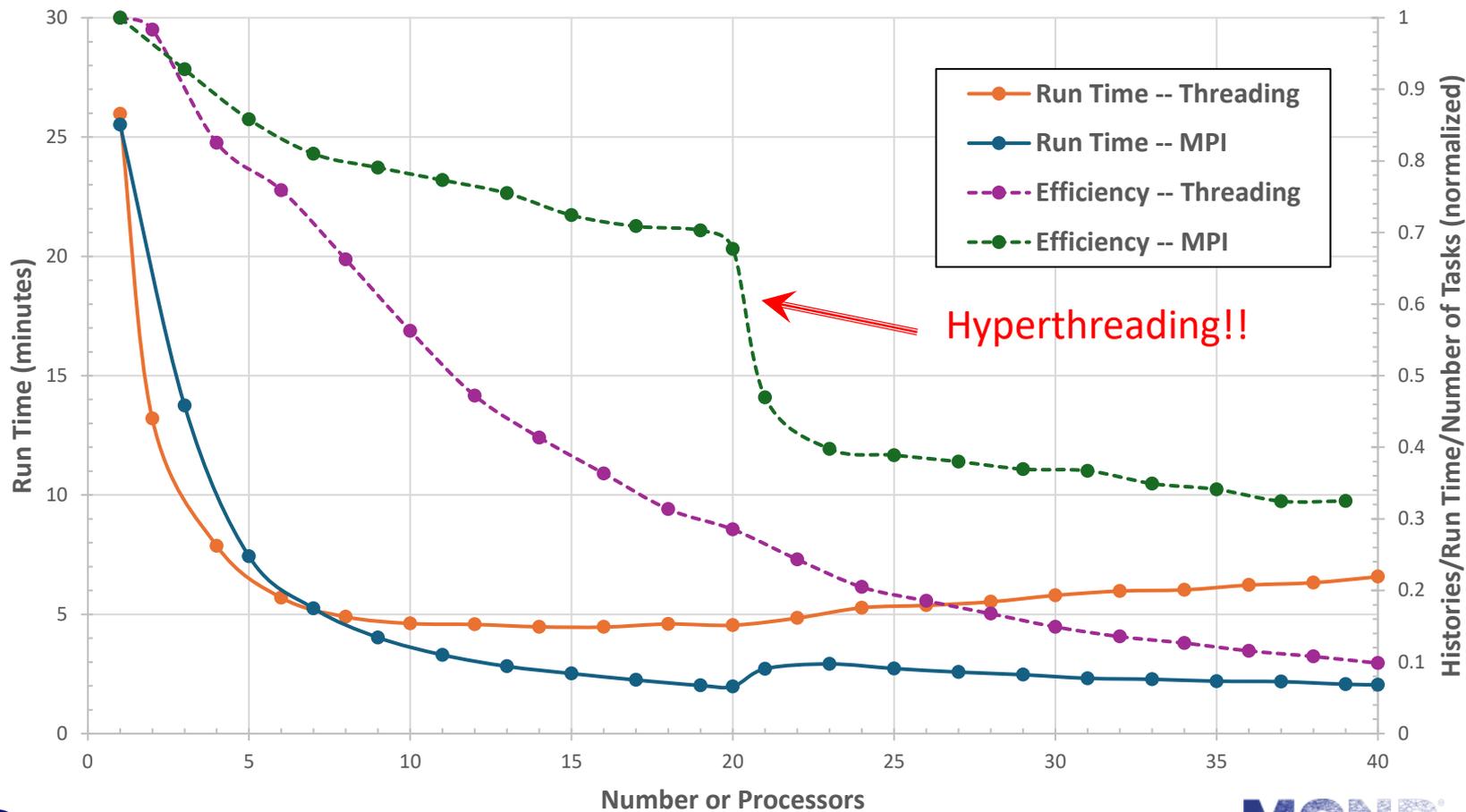
40 tasks

Function	CPU Time (s)	% of CPU Time
__kmpc_set_lock	2,291	85.10%
__kmp_fork_barrier	100	3.70%
acetot	35	1.30%
acecol	27	1.00%
colidn	22	0.80%
[Others]	217	8.10%

```
if( sources_need_locks ) call threading_lock_on(THREADING_LOCK_SOURCE)
if( kbp>0 .or. history_thread&kdb<0 ) then
  exit HISTORY_LOOP
```

For criticality problems, source particles are processed one thread at a time.

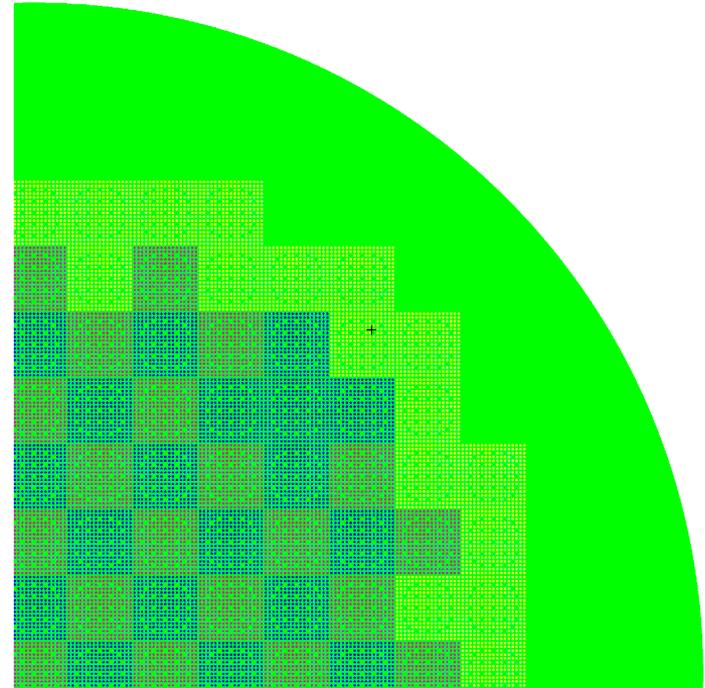
Godiva Threading and MPI Performance



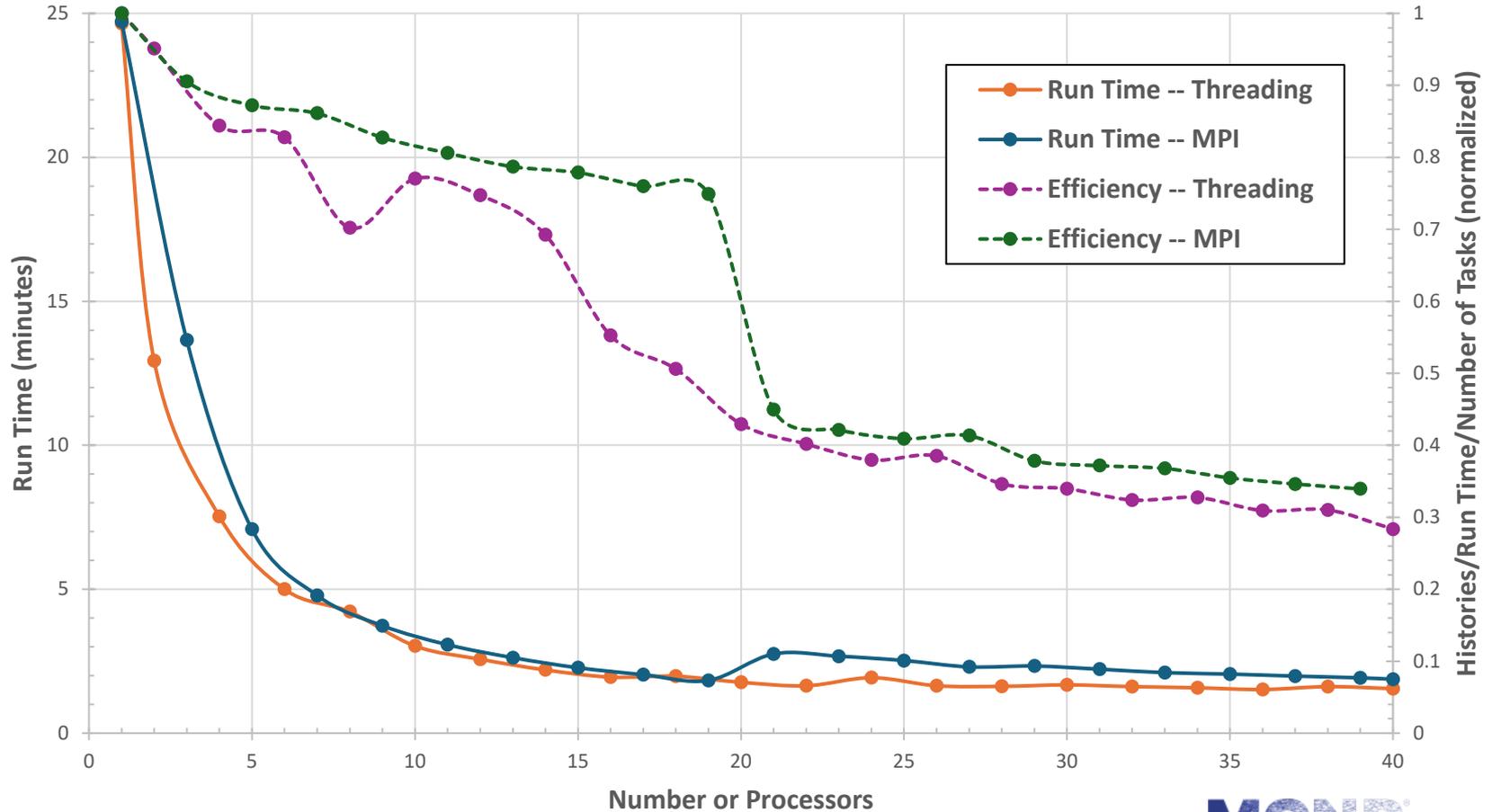
Sample Problem 2: PWR initial core, 2D model, 17x17 bundles

```

pwr2d-whole - PWR initial core, 2D model, 17x17 bundles
c 2.1%, 2.6%, and 3.1% enrichment for assemblies
c Taken from "Whole Core Calculations of Power Reactors
c by Use of Monte Carlo Method" by Nakagawa and Mori,
c J. Nuc. Sci. and Tech., 30(7), 692-701 (1993)
c
1 1 6.60783e-2 -1 u=1 $ UO2 2.1%
2 5 4.310700e-2 1 -2 u=1 $ Zr
3 4 6.622400e-2 2 u=1 $ H2O
c
4 2 6.60798e-2 -1 u=2 $ UO2 2.6%
5 5 4.310700e-2 1 -2 u=2 $ Zr
6 4 6.622400e-2 2 u=2 $ H2O
c
7 3 6.60913e-2 -1 u=3 $ UO2 3.1%
8 5 4.310700e-2 1 -2 u=3 $ Zr
9 4 6.622400e-2 2 u=3 $ H2O
c
10 4 6.622400e-2 -3 u=4 $ H2O
11 5 4.310700e-2 3 -4 u=4 $ Zr
12 4 6.622400e-2 4 u=4 $ H2O
c
c ----- lattice of fuel/water, 2.1% enrichment
13 0 -5 lat=1 u=5 fill= -8:8 -8:8 0:0
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    1 1 1 1 1 4 1 1 4 1 1 4 1 1 1 1
    1 1 1 4 1 1 1 1 1 1 1 1 4 1 1 1
  
```



PWR Initial Core Threading and MPI Performance



Sample Problem 3: Fixed Source with 7.5 Million Tally Bins

Transport neutrons through void, tallying using F4 tallies
 c Air cells

```

2001 1 -1.29300E-03 -2101      30 -32  imp:n=1  $imp:n= 1
2002 1 -1.29300E-03 -2102  2101 30 -32  imp:n=1  $imp:n= 2
2003 1 -1.29300E-03 -2103  2102 30 -32  imp:n=1  $imp:n= 4
2004 1 -1.29300E-03 -2104  2103 30 -32  imp:n=1  $imp:n= 8
2005 1 -1.29300E-03 -2105  2104 30 -32  imp:n=1  $imp:n= 16
2006 1 -1.29300E-03 -2106  2105 30 -32  imp:n=1  $imp:n= 32
2007 1 -1.29300E-03 -2107  2106 30 -32  imp:n=1  $imp:n= 64
2008 1 -1.29300E-03 -2108  2107 30 -32  imp:n=1  $imp:n= 128
2009 1 -1.29300E-03 -2109  2108 30 -32  imp:n=1  $imp:n= 256
2010 1 -1.29300E-03 -2110  2109 30 -32  imp:n=1  $imp:n= 512
2011 1 -1.29300E-03 -2111  2110 30 -32  imp:n=1  $imp:n= 1024
2012 1 -1.29300E-03 -2112  2111 30 -32  imp:n=1  $imp:n= 2048
2013 1 -1.29300E-03 -2113  2112 30 -32  imp:n=1  $imp:n= 4096
2014 1 -1.29300E-03 -2114  2113 30 -32  imp:n=1  $imp:n= 8192
2015 1 -1.29300E-03 -2115  2114 30 -32  imp:n=1  $imp:n= 16384
  
```

```

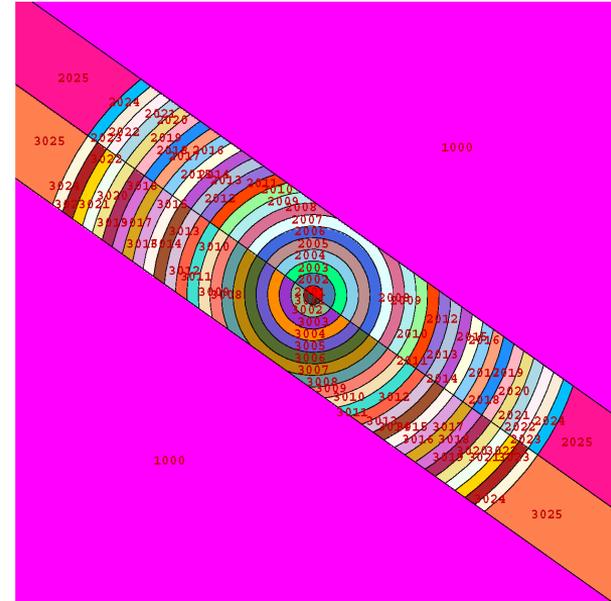
.....
f104:n  2001 2002 2003 2004 2005 2006 2007 2008 2009 2010
        2011 2012 2013 2014 2015 2016 2017 2018 2019 2020
        2021 2022 2023 2024 2025
  
```

```

f114:n  3001 3002 3003 3004 3005 3006 3007 3008 3009 3010
        3011 3012 3013 3014 3015 3016 3017 3018 3019 3020
        3021 3022 3023 3024 3025
  
```

```

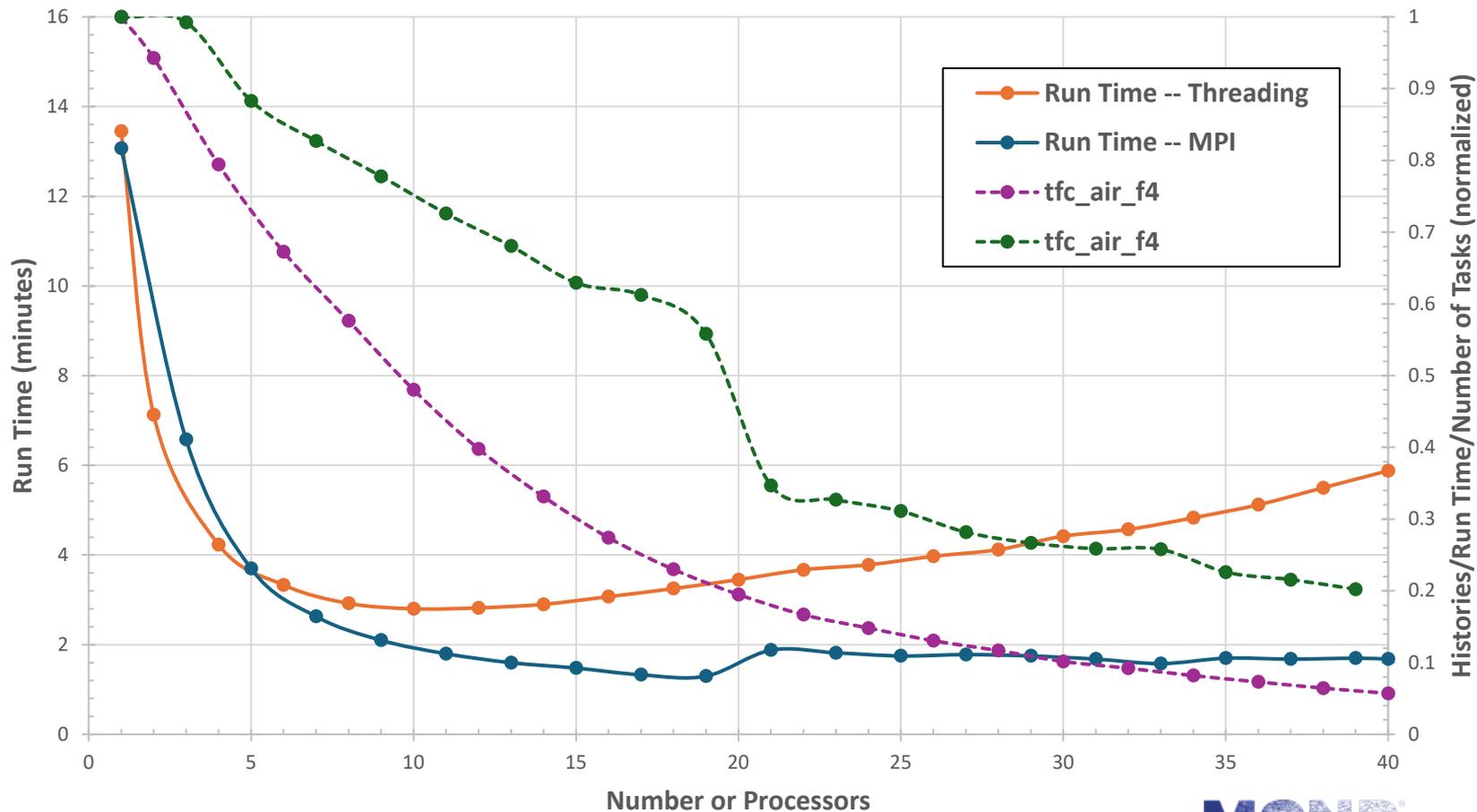
e0 1e-6 999ilog 15
t0 1 98i 1e6
cf104 2010
  
```



Tally	Bins
102	5,010,000
114	2,505,000
Total	7,515,000

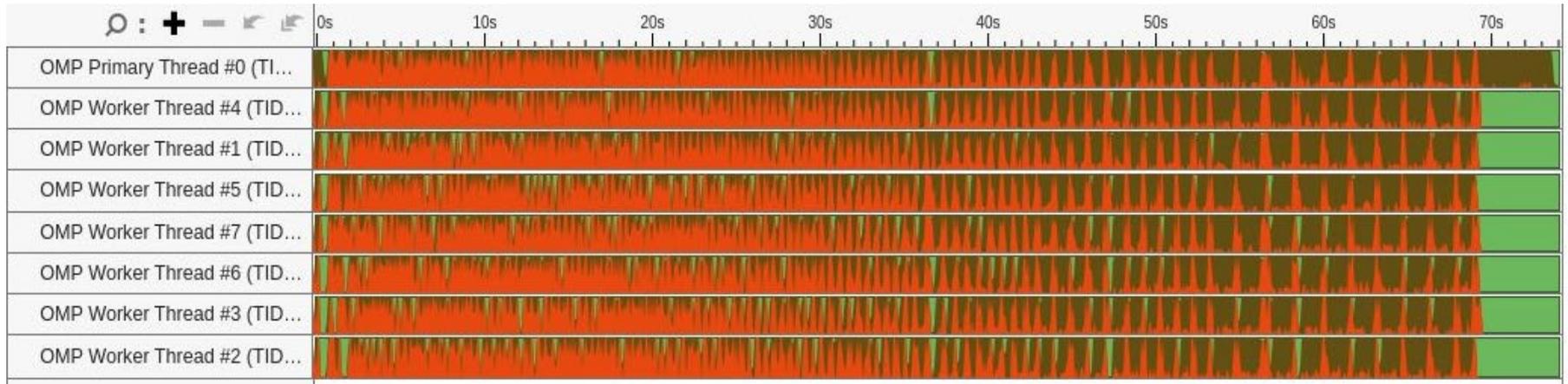
Each task requires an additional 250 MB of memory

Fixed Source With 7.5 Million Tally Bins -- Threading and MPI Performance



Fixed Source w/ 7.5 M Tally Bins -- CPU Cycle Utilization

8 tasks



Note that the frequency of spin and overhead spikes decrease later in the problem

Tally Fluctuation Charts

- To calculate a point for the tally fluctuation chart, MCNP stops transporting particles and collects the tally results.
- MCNP begins by calculating TFC points every 1000 histories. But since the TFC chart is limited to 20 points, after MCNP calculates the 20th point, the TFC chart is trimmed down to 10 points and MCNP doubles the number of histories run between the TFC point calculations.
- The initial value of 1000 histories per TFC point can be changed using the 5th entry of the **PRDMP** card.
 - MPI runs automatically set this value to **NPS/10**.
 - Except for problems with point detectors that use Russian Roulette to limit small contributions **AND** which **k_i** on the **DD** card is negative. This is the default for F5 tallies.

How to find the best way to run a problem? Test Runs!!

Run time (minutes) using all 40 Logical CPUs			
Parallel setup	Godiva	PWR Initial Core	7.5 M tally bins
nmpi 3, tasks 20	22.67	7.97	9.83
nmpi 5, tasks 10	14.40	5.83	5.82
nmpi 6, tasks 8	19.12	7.80	7.88
nmpi 9, tasks 5	10.70	5.82	5.75
nmpi 11, tasks 4	10.42	5.88	5.82
nmpi 21, tasks 2	10.63	6.08	6.28

Summary

- Discussed how MCNP uses threading and MPI to run problems in parallel
- Parallel timing studies for a few MCNP problems were performed. These studies showed that:
 - MCNP does not take advantage of the extra logical CPU cores provided by hyperthreading
 - Source particles are not process in parallel for criticality problems
 - Increasing the number of histories between TFC points can significantly improve performance in thread only problems
- Conduct several test runs to determine which combination of threading and MPI is the most effective for a problem.