BLUE WATERS

SUSTAINED PETASCALE COMPUTING

May 21, 2013

Blue Waters Workshop, May 22-23, 2013
Application Scaling Case Study

Mark Straka, NCSA


GREAT LAKES CONSORTIUM
FOR PETASCALE COMPUTATION

Weather Research and Forecasting (WRF)

A Case Study – issues and guidelines for performance

Contents:

1. Similar in performance to several PRAC applications (e.g. CM1) for climate and severe storm simulation; offers many common insights to scaling
2. Huge code – many different types of physics can be activated
3. Issues to watch for – tips from what we've learned
4. Results and Limits on our scaling – what's under our control, and what is not…
5. General guidelines – "best practices" (or at least good)
6. When to approach us for help – using resources efficiently

Three Conceptual Realms for Application Analysis

- Source Code
  - Profiling, applying compiler options, recoding
  - ARCH-specific configuration flags
- Runtime Input Parameters
  - I/O formatting, hybrid layouts, grid structure
  - namelist.input file
- Operating System
  - Topology, core/node placement (e.g. ***aprun*** command), network protocols, MPICH env, etc.

Source Code Layer

Consider generalizing these to your own code:

- Modified code for output diagnostic volume
  - Avoid multi-task writing of redundant information
- Modified WRF's internal profiling macros
  - Useful for a code to self-profile as a sanity check
- Compiler directives and precompiler macros
  - Used to avoid known compiler bugs or weaknesses
  - Invoke customized code for a given compiler/arch combo
- Loop restructuring for vectorization
  - Good luck… Cray compiler is fairly aggressive
- Analysis of microphysics routines for poor computational rate and load imbalance
  - Lots of IF conditionals inhibit vector performance

Input Parameters and Runtime Configuration

These are the result of a long development history!

- Hybrid Code
  - Empirically test MPI vs. OpenMP balance
  - Choice of grid layout at runtime (MPI)
  - Choice of sub-tiling size at runtime (OMP)
- Choice of formats for parallel I/O
  - PNetCDF, multi-file, quilted I/O servers, etc.
- Choice of output fields for volume control
  - Limit output to “interesting” data (e.g. rain, snow)

Operating System Layer

- MPICH rank reordering with grid_order (module load perftools)
- ***aprun*** options
  - Be familiar with concept of sockets, nodes, NUMA regions, int cores, etc.
- Lustre striping
  - WRF heuristic, e.g. number of OSTs ~ ½(sqrt(numprocs)) in multiples of 2
- Balanced Injection – did not help WRF
  - More useful for ALL_TO_ALL collective comm patterns
  - https://bluewaters.ncsa.illinois.edu/balanced-injection
- Core Specialization – did not help WRF
  - Dedicating a core on a node to handle OS interrupts; less jitter
- More advanced node placement schemes
  - Topology awareness tools being studied in depth by Cray, others

Incremental Approach to Scaling

- Used 5 “weakly scaled” problem sizes on node counts 1, 9, 81, 729, 6561
  - Number of grid points increased 9x correspondingly
  - 81km resolution up to 1km res.
- Use of CrayPAT and Cray profiler library and WRF’s internal BENCH
- Seemingly minor I/O quickly became an impediment to rapid testing – each MPI rank was writing redundant diagnostic information; had to make minor source code mods. Very problematic to “ls” or “rm” 100,000 files…
- Hybrid MPI/OpenMP code – optimal balance of comm and cache – experimented with MPI task layout and number of OMP tiles
- Additional compiler options do not help WRF substantially - Cray compiler has fairly aggressive defaults; -lfast_mv (module load libfast) only a few %
- Nearest neighbor 2-D communication pattern – benefits from ***grid_order*** utility (setenv **MPICH_RANK_REORDER_METHOD** 3) by up to 20%

Incremental Approach to Scaling, continued

- Lustre striping heuristic – e.g. *lfs setstripe* command
- setenv MPICH_MPIIO_HINTS_DISPLAY 1 – will show if file has been opened using MPI-IO
- setenv MPICH_MPIIO_HINTS "wrfout*:striping_factor=64"
  - All of our "wrfout*" files will be created with 64 OST striping; 64 MPI-IO concentrators will be utilized
- WRF also has several internal I/O schemes (PNetCDF, HDF5, Fortran unformatted, multi-file, etc.) – alternatively, we found it useful to process our initial input files using PNetCDF since they were supplied as single huge files; then we immediately rewrote these out as multi-restart files in raw (Fortran unformatted), one file per MPI rank.

- Multi-file I/O is the fastest, but limits us always to that number of tasks

More about hybrid functionality

- There are 4 NUMA regions on the Cray XE node, each with 8 integer cores
  - Even though full node can physically be addressed by OMP, extending threading beyond a NUMA region is not likely to be better than 8 or fewer.
  - Empirically, we determined that using 16 MPI tasks per node, each with 2 OMP threads was optimal
  - It is also known that WRF responds better to a more rectangular decomposition (i.e. X << Y) which leads to longer inner loops for better vector and register reuse, better cache blocking, and more efficient halo exchange communication pattern.
  - Lesson: If your code allows flexibility for domain decomposition, experiment toward the above goals

But nothing is perfect…

- There may be some negative side effects of rank reordering w.r.t. impact on I/O collections (optimizing task layout for the 1st may not be best for the 2nd)
- But overall improvement in the pure integration (non-I/O) time steps is an overwhelming win.

- Following 3 slides, work of Robert Fiedler, Cray Applications Analyst

Virtual Topologies and Task Placement

- **Many applications define Cartesian grid virtual topologies**
  - MPI_CartCreate
  - Roll your own (i, j, …) virtual coordinates for each rank
- **Craypat rank placement**
  - Automatic generation of rank order based on detected grid topology
- **grid_order tool**
  - User specifies virtual topology to obtain rank order file
  - Node list by default is in whatever order ALPS/MOAB provide
- **These tools can be very helpful in reducing off-node communication, but they do not explicitly place neighboring groups of partitions in virtual topology onto neighboring nodes in torus**

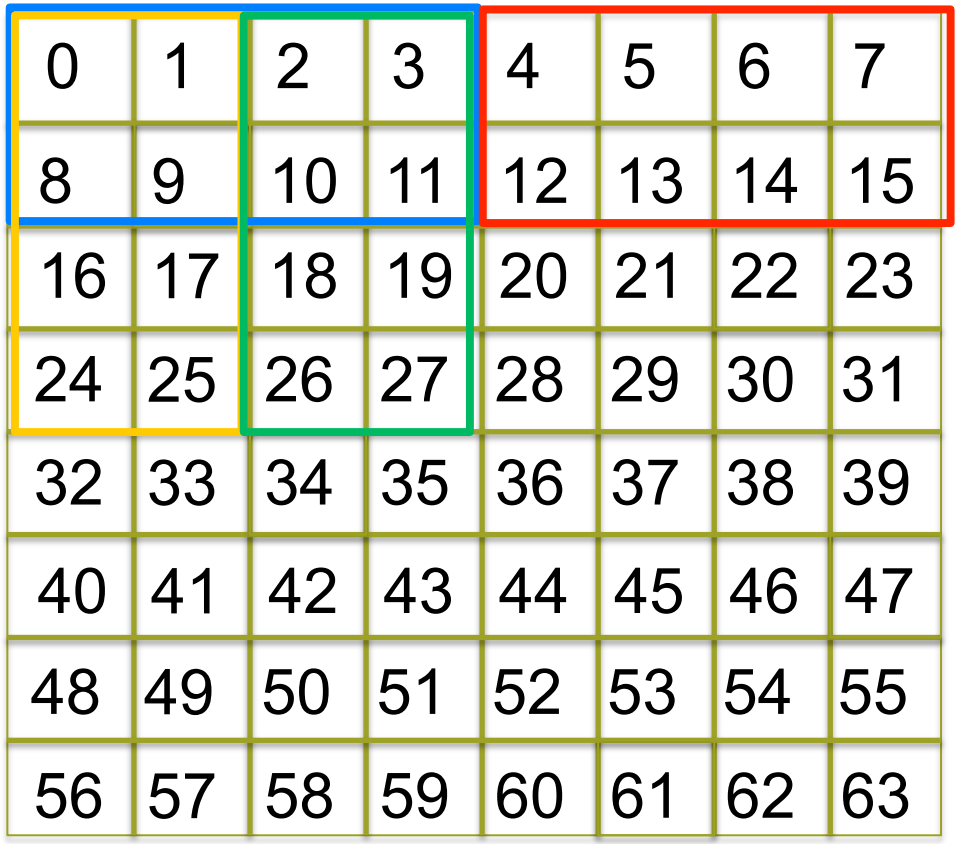
Examples: 2D Virtual topology

grid_order –C –c 4,2 –g 8,8

- **Ranks ordered with 1st dim changing fastest (column major, like Fortran)**
- **Nodes get 4x2 partitions**
- **Rank order is**
  - 0,1,2,3,8,9,10,11 on 1st node
  - 4,5,6,7,12,13,14,15 on 2nd
  - Node pair is 8x2

grid_order –R –c 4,2 –g 8,8

- **Ranks ordered with 2nd dim changing fastest**
- **Rank order is**
  - 0,1,8,9,16,17,24,25 on 1st node
  - 2,3,10,11,18,19,26,27 on 2nd
  - Node pair is 4x4



0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Examples: 2D Virtual Topology

WRF

- 2D mesh, 6075x6075 cells
- 4560 nodes, 16 tasks per node, 72960 tasks
- 2 OpenMP threads
- Found best performance with grid_order -C -c 2,8 -g 190,384
  - Node pair is 4x8
  - ~18% speedup over SMP ordering

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
etc															

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
etc															

Grid Order and Virtual Topology

First, "module load perftools"

Usage: grid_order -C|-R [-P|-Z|-H] -g N1,N2,...
-c n1,n2,... [-o d1,d2,...]
[-m max] [-n ranks_per_line] [-T] [i1 i2 ...]

Used to generate a rank order list for an MPI application with nearest-neighbors communication.
Grid is a 'virtual' topology in the application's logic, not the physical topology of the machine.
Assumed that ranks in the list will be packed onto machine nodes in the order given.

You must specify either -C or -R for column- or row-major numbering.
For example, if the application uses a 2 or 3 dimensional grid, then
use -C if it assigns MPI rank 1 to position (1,0) or (1,0,0), but
use -R if it assigns MPI rank 1 to position (0,1) or (0,0,1).
To see the difference, compare the output from:

- grid_order -C -g 4,6
- grid_order -R -g 4,6

Example:

```
# grid_order -C -Z -c 1,6 -g 4,6
# Region 0: 0,0 (0..23)
0,4,8,12,16,20
1,5,9,13,17,21
2,6,10,14,18,22
3,7,11,15,19,23

# grid_order -R -Z -c 1,6 -g 4,6
# Region 0: 0,0 (0..23)
0,1,2,3,4,5
6,7,8,9,10,11
12,13,14,15,16,17
18,19,20,21,22,23
```

Notice that the “stride” will be the X grid dimension (e.g. 4) if using Column Major ordering (like Fortran – 1st dimension changes fastest);

It will be the Y grid dimension (e.g. 6) when invoking Row Major ordering (2nd dimension changes fastest).

Virtual Toplogies[3]

What Are They?

- In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric "shape". [3]
- The two main types of topologies supported by MPI are Cartesian (grid) and Graph.
- MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology.
- Virtual topologies are built upon MPI communicators and groups.
- Must be "programmed" by the application developer.

Why Use Them?

- Convenience
  - Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure.
  - For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data. (e.g WRF and CM1 in this talk)
- Communication Efficiency
  - Some hardware architectures may impose penalties for communications between successively distant "nodes".
  - A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.
  - The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored.

A simplified mapping of processes into a Cartesian virtual topology appears below: [Reference #3]

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Example of MPI rank ID Locality Using Popular Cartesian code:

https://computing.llnl.gov/tutorials/mpi/samples/Fortran/mpi_cartesian.f

```
…
 call MPI_INIT(ierr)
     call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)



       call MPI_CART_CREATE(MPI_COMM_WORLD, 2, dims, periods, reorder,
    &                       cartcomm, ierr)
       call MPI_COMM_RANK(cartcomm, rank, ierr)
       call MPI_CART_COORDS(cartcomm, rank, 2, coords, ierr)
       call MPI_CART_SHIFT(cartcomm, 0, 1, nbrs(UP), nbrs(DOWN), ierr)
       call MPI_CART_SHIFT(cartcomm, 1, 1, nbrs(LEFT), nbrs(RIGHT),
    &                      ierr)
…
```

MPI rank ID Locality

Add following call to return NID number:

call MPI_GET_PROCESSOR_NAME (myname, mylen, ierr)

Also, can use the “xtnodestat” command to see job IDs mapping to the hardware nodes.

aprun -d 2 -N 16 -n 1024 ./a.out (2 OMP threads per MPI task)

32 total tasks per node (2x16)

Alternatively:

1 OMP thread per MPI rank :

grid_order -C -Z -c 2,16 -g 32,32 aprun -d 1 -N 32 -n 1024 ./a.out

Maintain 32 total tasks per node (1x32).

4 OMP threads per MPI rank :

grid_order -C -Z -c 2,4 -g 32,32 aprun -d 4 -N 8 -n 1024 ./a.out

Maintain 32 total tasks per node (4x8).

Run 2 experiments: (1) using default rank ordering; (2) using grid_order utility and MPICH_RANK_ORDER file

MPI rank ID Locality, continued

setenv MPICH_RANK_REORDER_METHOD 1 (default)

nid09140 992
nid09140 993
nid09140 994

... **(Consecutive assignment)**

nid09140 1005
nid09140 1006
nid09140 1007

(16 total entries per node)

```
nid09140 rank=   992 coords=    31    0
nid09140 rank=   993 coords=    31    1
nid09140 rank=   994 coords=    31    2
nid09140 rank=   995 coords=    31    3
nid09140 rank=   996 coords=    31    4
nid09140 rank=   997 coords=    31    5
…
nid09140 rank=  1006 coords=    31   14
nid09140 rank=  1007 coords=    31   15
```

MPI rank ID Locality, continued

- setenv MPICH_RANK_REORDER_METHOD 3

```
# grid_order -C -Z -c 2,8 -g 32,32
# Region 0: 0,0 (0..1023)
0,1,32,33,64,65,96,97,128,129,160,161,192,193,224,225
2,3,34,35,66,67,98,99,130,131,162,163,194,195,226,227
4,5,36,37,68,69,100,101,132,133,164,165,196,197,228,229
...
796,797,828,829,860,861,892,893,924,925,956,957,988,989,1020,1021
798,799,830,831,862,863,894,895,926,927,958,959,990,991,1022,1023
```

- Size of MPICH_RANK_ORDER file is 64 rows by 16 columns wide.
- Note the stride of 2 between ROWS and stride of 32 between pairs in columns.

- Make sure that your chosen core topology (2,8 in this case) matches what you use in your run script (16 MPI ranks per node in this case).

MPI rank ID Locality, continued

nid09140 796
nid09140 797
nid09140 828
nid09140 829
nid09140 860
nid09140 861
...

Note the rank pairings separated by stride of 32.

nid09140 rank= 796 coords= 24 28
nid09140 rank= 797 coords= 24 29
nid09140 rank= 828 coords= 25 28
nid09140 rank= 829 coords= 25 29
…
nid09140 rank= 1020 coords= 31 28
nid09140 rank= 1021 coords= 31 29

Note the difference in the coordinates from the default mapping in previous example.

Sample xtnodestat output

Current Allocation Status at Wed May 22 14:14:52 2013

```
    C0-0          C0-1          C0-2          C0-3
n3 adadaaaaacacacac aoaoaiaiaiaiafaf acacabaaaeaeafaf ararararararaoao
n2 adadaaaaacacacac aoaoaiaiaiaiafaf acacaaaaaeaeafaf ararararararaoao
n1 abadacacacacacac aoaoaiaiaiaiafaf acacaeaeaeaeafaf ararararararaoao
c2n0 abadacacacacacac aoaoaiaiaiaiafaf acacaeaeaeaeafaf ararararararaoao
n3 ababaaaaababaaaa afafafafafafafaf agagagagagXXafaf arararSSararararar
n2 ababaaaaababaaaa afafafafafafafaf agagagagagagafaf arararSSararararar
n1 ababaaaaaaacaaaa afafafafafafafaf agagagagagagafaf arararSSararararar
c1n0 ababaaaaacabaaaa afafafafafafafaf agagagagagagafaf arararSSararararag
n3 SSSSSSSSSSSSaaaa afafafafAAAAasas agagagagaiaiabab akakAAAAAAAAagag
n2 SSSSSSSSSSSSaaaa afafafafAAAAasas agagagagaiaiabab akakAAAAAAAAagag
n1 SSSSSSSSSSSSaaaa afafafafAAAAasas agagagagahaiabab akakAAAAAAAAagag
c0n0 SSSSSSSSSSSSaaaa afafafafafAAasas agagagagAAaiabab AAakAAAAAAAAagag
s0011223344556677 0011223344556677 0011223344556677 0011223344556677
```

Sample xtnodestat output, continued

Legend:
nonexistent node S service node
; free interactive compute node - free batch compute node
A allocated interactive or ccm node ? suspect compute node
W waiting or non-running job X down compute node
Y down or admindown service node Z admindown compute node

Available compute nodes: 0 interactive, 5499 batch

Job ID	User	Size	Age	State	command line
aa	1656263	haox	64	6h34m	run	enzo_LW.exe
ab	1655967	yanxinl	128	11h17m	run	namd2
ac	1656472	yanxinl	128	4h27m	run	namd2

…
…

Following 5 slides courtesy of Pete Johnsen, Cray

WRF and MPI-IO

- Typically WRF performs best when MPI rank ordering is used to place halo exchange nearest neighbors on the same node
- Also, I/O times can be reduced if WRF uses parallel netcdf (MPI-IO)
- But, in many cases, using MPI rank ordering slows down MPI-IO (bug 784718)
- Worked with David Knaak to isolate and fix MPI-IO
  - Problem was multiple MPI-IO concentrators were assigned to the same node when MPI rank ordering is used

WRF MPI rank ordering

- Example WRF run with 16 MPI ranks per XE6 Interlagos node, 2 OpenMP threads per rank
- MPICH_RANK_REORDER_METHOD=1
- At most, 2 neighbors on same node

16 MPI ranks on a node (black cells)

Halo exchange partners for rank 17 (orange cells)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
etc															

WRF MPI rank ordering

- Example WRF run with 16 MPI ranks per XE6 Interlagos node, 2 OpenMP threads per rank
- MPICH_RANK_REORDER_METHOD=3
- 3 neighbors now on same node
- Runs about 8% faster, NCSA large WRF case runs 25% faster

16 MPI ranks on a node (black cells)

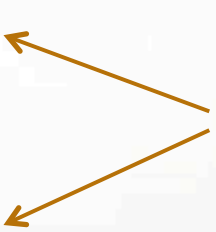
Halo exchange partners for rank 17 (orange cells)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
etc															

WRF MPI-IO

- David K. added diagnostic message to MPICH library to show what MPI ranks and nodes are allocated for MPI-IO concentrators
- New MPICH environment variable turns this on:
  - MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY=1
- For this test case and MPI rank reordering = 3 (MPI-IO cb_nodes=16) :

AGG	Rank	nid
0	0	nid00576
1	2	nid00577
2	4	nid00606
3	6	nid00607
4	8	nid00576
5	10	nid00577
6	12	nid00606
7	14	nid00607
8	16	nid00576
9	18	nid00577
10	20	nid00606
11	22	nid00607
12	24	nid00576
13	26	nid00577
14	28	nid00606
15	30	nid00607



Multiple MPI-IO concentrators on 1 node

WRF MPI-IO

- David made changes to MPI-IO rank selection algorithm based on actual rank ordering so MPI-IO concentrators are back on unique nodes
- Default behavior now starting with MPICH version 5.5.5

WRF I/O performance improved:

18 GBytes input = 11.2 seconds old library

= 8.1 seconds new lib

4.7 GBytes output = 3.2 seconds old lib

= 1.7 seconds new lib

```
B  AGG     Rank         nid
B ----   ------   --------
B     0        0   nid00576
B     1        2   nid00577
B     2       64   nid00578
B     3       66   nid00579
B     4      192   nid00580
B     5      194   nid00581
B     6      132   nid00582
B     7      134   nid00583
B     8      128   nid00600
B     9      130   nid00601
B    10      196   nid00602
B    11      198   nid00603
B    12       68   nid00604
B    13       70   nid00605
B    14        4   nid00606
B    15        6   nid00607
```

Best and Worst Grid Orderings for WRF

- Dataset: 2025 x 2025 x 28 (3km resolution) simulation on 729 nodes, 11664 MPI tasks (81x144)
- Experimented with WRF's internal mesh reordering capability
- Default timing:

```
mean:     0.064262 , min:     0.05126 , max:     0.19463

mean:     0.057271  (best)  # grid_order -R -Z -c 8,2 -g 81,144 , REORDER_MESH = T
mean:     0.200316  (worst) # grid_order -R -Z -c 8,2 -g 81,144 , REORDER_MESH = F

min:      0.049030   (best)   # grid_order -R -Z -c 2,8 -g 81,144 , REORDER_MESH = T
min:      0.127730    (worst) # grid_order -R -Z -c 8,2 -g 81,144 , REORDER_MESH = F

max:      0.097240    (best) # grid_order          -c 2,8 -g 81,144 , REORDER_MESH = F
max:      0.841100    (worst) # grid_order -R -Z -c 2,8 -g 81,144 , REORDER_MESH = F
```

- Factor of 3-4 from best to worst configurations
- Mean times improve by 12%; MIN improves by 5%; MAX by 20x !
- MAX times may indicate worst-case network neighbor exchanges at a particular step
- MEAN times are influenced by outside perturbations/jitter such as Lustre ping effect [5]

Focus on MIN times – Best and Worst cases have *IDENTICAL* MPICH_RANK_ORDER Entries!

BEST: (has WRF REORDER_MESH == .TRUE.)

```
# grid_order -R -Z -c 8,2 -g 81,144
# Region 0: 0,0 (0..11519)
0,1,144,145,288,289,432,433,576,577,720,721,864,865,1008,1009
2,3,146,147,290,291,434,435,578,579,722,723,866,867,1010,1011
4,5,148,149,292,293,436,437,580,581,724,725,868,869,1012,1013
6,7,150,151,294,295,438,439,582,583,726,727,870,871,1014,1015
```

WORST: (has WRF REORDER_MESH == .FALSE.)

```
# grid_order -R -Z -c 8,2 -g 81,144
# Region 0: 0,0 (0..11519)
0,1,144,145,288,289,432,433,576,577,720,721,864,865,1008,1009
2,3,146,147,290,291,434,435,578,579,722,723,866,867,1010,1011
4,5,148,149,292,293,436,437,580,581,724,725,868,869,1012,1013
6,7,150,151,294,295,438,439,582,583,726,727,870,871,1014,1015
```

The WRF reordering interacts with the row-major ordering of the grid_order utility (-R option). Compare to:

```
# grid_order -C -Z -c 8,2 -g 81,144
# Region 0: 0,0 (0..11519)
0,1,2,3,4,5,6,7,81,82,83,84,85,86,87,88
8,9,10,11,12,13,14,15,89,90,91,92,93,94,95,96
16,17,18,19,20,21,22,23,97,98,99,100,101,102,103,104
24,25,26,27,28,29,30,31,105,106,107,108,109,110,111,112
```

Using CrayPAT for Rank Reordering and using Profiling Library

CrayPAT's pat_report tool can generate recommended rank order files.
Specify the -Ompi_sm_rank_order flag.
It generates suggested custom MPICH_RANK_ORDER file(s). You can copy this to your local file.
Compare to the predefined ones below.

These are in addition to the 3 predefined mappings:

ROUND ROBIN (0)
One rank per node, wrap around.

SMP STYLE (1)
Fill up one node before going to the next.

FOLDED RANK (2)
One rank per node, wrap back.

For our next experiment, we link with –lprofiler (libprofiler.a)
Run this executable to find a “profile*.txt” file upon successful run completion
Check for MIN/MAX values in key data sections, along with their respective MIN PE and MAX PE process locations

Using Cray profiling library to analyze load imbalance

Sample output:

Profile of wrf.exe
Number of processes 11664
Default page size 4.000K
Huge page size 2048.000K
PROFILER_PAPI RETIRED_SSE_OPS:ALL
MPICH_RANK_REORDER_METHOD 3
MPICH_COLL_OPT_OFF mpi_scatterv
MPICH_MPIIO_HINTS wrfout*:striping_factor=128 auxhist*:striping_factor=128
MALLOC_TRIM_THRESHOLD_ 134217728
MALLOC_MMAP_MAX_ 0
OMP_NUM_THREADS 2

Using Cray profiling library to analyze load imbalance

FAST

System summary Section:

	min	max	avg	minPE	maxPE
Wall clock time	1079.970	1081.210	**1080.194**	10504	10492
User processor time	**529.677**	**1198.579**	**1193.105**	8170	721
System processor time	5.876	64.288	7.652	6086	8170
Maximum memory usage (MB)	414.043	760.348	417.830	11663	8170
Memory usage at exit (MB)	414.039	760.344	417.826	11663	8170
Memory touched (MB)	417.000	777.906	422.524	11520	8170
Heap segment size (MB)	379.250	720.113	382.340	11663	8170
Minor page faults	106752	199144	108166	11520	8170
Major page faults	6	454	63	876	8170
Node memory size (MB)	64512.000	64512.000	64512.000	0	0
User memory available (MB)	64627.230	64627.230	64627.230	0	0
Total huge pages	128	140	130	4	0
Processor clock (GHz)	2.300	2.300			

SLOW

min	max	avg	minPE	maxPE
1246.370	1247.740	**1246.591**	2434	9236
893.880	**1541.284**	**1534.038**	10484	288
6.956	66.660	8.988	5294	10484
266.055	769.680	415.999	11349	10484
266.051	769.676	415.995	11349	10484
273.289	778.781	420.136	11493	10484
232.625	730.312	381.322	11349	10484
69962	199368	107554	11493	10484
5	470	64	10142	10484

~350s (28%) difference in times

Using Cray profiling library to analyze load imbalance

FAST **SLOW**

PAPI Summary Section:

	min	max	avg	minPE	maxPE	min	max	avg	minPE	maxPE
RETIRED_SSE_OPS:ALL	38188.354M	48801.047M	42336.279M	11547	4609	38186.845M	48801.027M	42336.256M	2267	113
per second	**35.548M**	**45.418M**	**39.409M**	11547	4609	**30.776M**	**39.340M**	**34.125M**	2267	113
FPU instructions	23633.175M	30900.586M	26331.532M	7471	4609	23633.668M	30900.632M	26331.622M	10338	113
per second	**22.001M**	**28.758M**	**24.511M**	7471	4609	**19.054M**	**24.910M**	**21.225M**	10338	113
Data cache accesses	448931.390M	2382313.771M	2355969.308M	8170	6843	894653.157M	2773020.406M	2746322.945M	10484	10420
per second	417.805M	2217.962M	2193.096M	8170	118	720.956M	2234.843M	2213.690M	10484	11583
Data cache misses	5992.930M	37296.484M	8149.899M	1256	8746	6665.874M	42116.672M	8851.548M	3566	10629
per second	5.578M	34.711M	7.586M	1256	8746	5.373M	33.940M	7.135M	3566	10629
L1 Dcache refs	2920283.550M	4853565.710M	4826782.471M	8170	6843	3748783.695M	5627009.350M	5599722.946M	10484	10420
User processor cycles	2717.805M	4517.962M	4493.096M	8170	118	3020.956M	4534.843M	4513.690M	10484	11583
Misaligned accesses	4839.964M	1116899.218M	5808.390M	11657	8170	4849.084M	657479.463M	5777.425M	11177	10484
Processor clock (GHz)	4506640.106	1039458556.323	5406824.347	11657	8170	3908867.213	529829561.819	4656915.939	11177	10484

Using Cray profiling library to analyze load imbalance

FAST

SLOW

MPI Summary Section

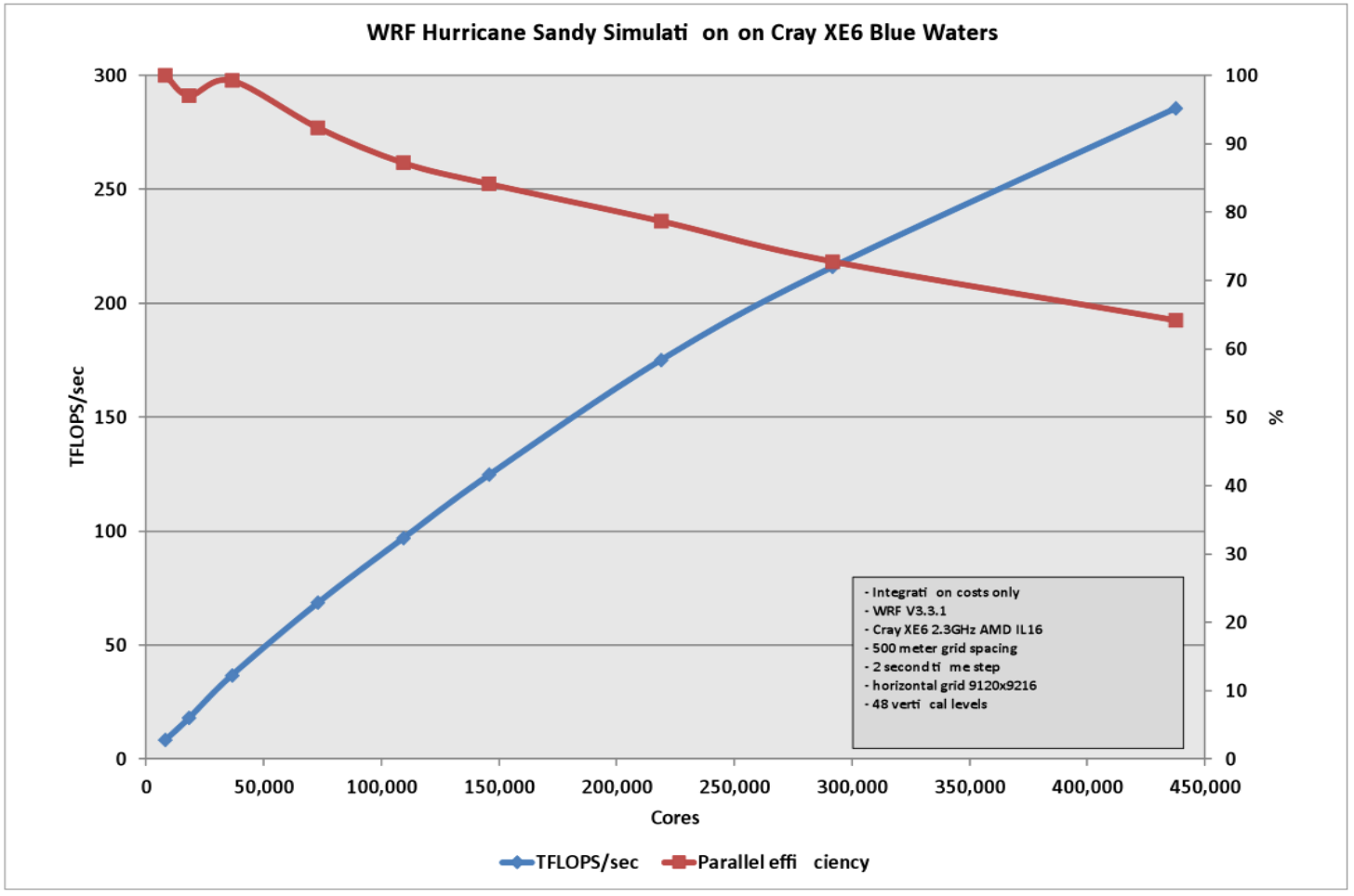
	min	max	avg	minPE	maxPE	min	max	avg	minPE	maxPE
Init-Finalize elapsed time	1073.993	1074.603	1074.312	6042	9013	1240.335	1241.005	1240.622	8003	11492
Total MPI time	**20.831**	**33.834**	**27.809**	4610	87	**185.896**	**198.783**	**191.807**	276	2186
Total communication time	0.692	2.625	0.872	11352	0	0.958	3.115	1.232	8177	0
Total Wait and Probe time	**8.943**	**22.410**	**15.977**	4610	11657	**166.452**	**180.320**	**173.083**	11351	2186
Total collective sync time	**5.099**	**9.722**	**8.243**	0	289	**11.843**	**16.219**	**14.840**	0	289
Isend total time	0.127	0.451	0.373	0	9357	0.250	1.496	0.466	11663	9743
Irecv total time	0.052	0.284	0.158	143	277	0.051	0.699	0.295	11663	11575
Bcast total time	0.129	0.384	0.245	11488	1230	0.126	0.451	0.283	11662	3502

Using Cray profiling library to analyze load imbalance

MPI Summary Section (values common to FAST,SLOW versions)

	min	max	avg	minPE	maxPE
Isend total bytes	2174209584	4816612416	4674004373	0	145
Irecv total bytes	2814767904	5629535808	5474286461	0	145
Bcast total bytes	1276228	1276228	1276228	0	0
Wait total bytes	4988977488	10446148224	10148290835	0	145
Isend average bytes	21503	24829	24264	144	1
Irecv average bytes	27027	30877	28436	144	1
Bcast average bytes	3545	3545	3545	0	0
Wait average bytes	24265	27853	26350	144	1

Example: WRF strong scaling



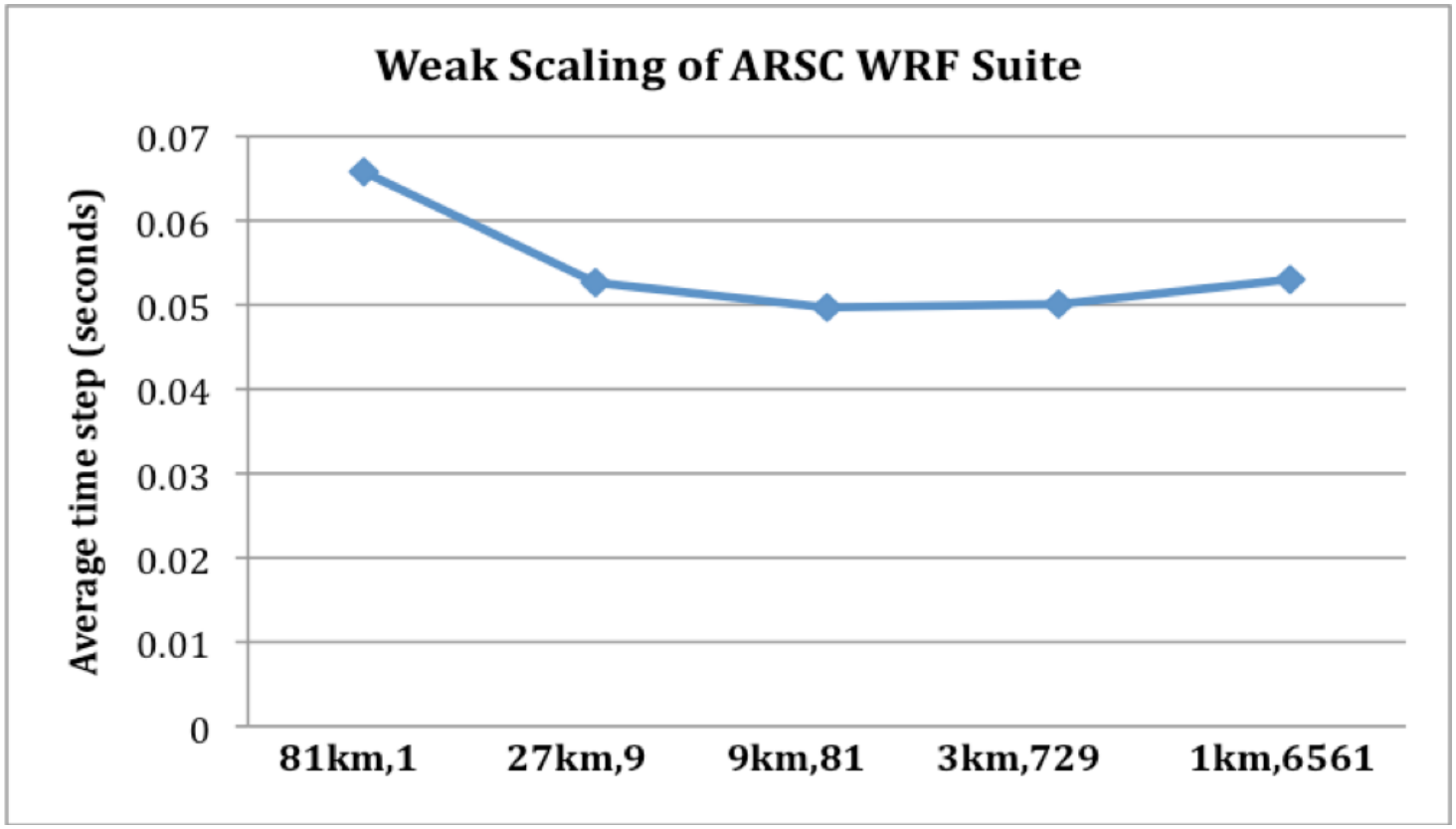
Rank reordering reduces off-node messages; increases on-node

Halo Exchange Messaging Statistics					
Placement Method	Total Messages	Total Bytes Exchanged	On-node Messages	Off-node Messages	Off-node Bytes Exchanged
Default placement	3.6E07	1.5E12	1.8E07	1.8E07	1.1E12
Optimized MPI rank ordering	3.6E07	1.5E12	2.4E07	1.2E07	2.8E11

WRF Strong Scaling for Input of 9120 x 9216 x 48 Points, 16x2 Layout

Scaling Details				
Core Count	XE6 Nodes	Horizontal Decomposition (MPI ranks)	Average Time Step (seconds)	Sustained Performance (Tflops/sec)
8192	256	32x128	3.895	8.3
18240	570	38x240	1.802	18.0
36480	1140	76x240	0.882	36.8
72960	2280	95x384	0.474	68.5
109440	3420	120x456	0.334	97.1
145920	4560	190x384	0.260	124.8
218880	6840	228x480	0.185	175.1
291840	9120	285x512	0.150	216.0
437760	13680	285x768	0.114	285.7

Example: WRF weak scaling



WRF Weak Scaling:

Parallel efficiency normalized to smallest dataset on single node.

km	Core Count	XE6 Nodes	Horizontal Decomp. (MPI ranks)	Patch size	Patch cells	Ave. Time Step (secs)	Parallel eff. (%)
1	209952	6561	144x729	43x9	387	0.053005	124
3	23328	729	81x144	25x15	375	0.050091	131
9	2592	81	16x81	43x9	387	0.049692	132
27	288	9	8x18	28x13	364	0.052637	125
81	32	1	2x8	37x10	370	0.065783	100

Example of Load Imbalance Using Profiling

- CM1 code
  - PRAC application
  - Similar in characteristics and structure to WRF
- 4096 processors, Morrison microphysics
  - Over sample interval only ~200 procs did MP work
  - Work per proc, per step ranged from 1 to ~100
  - Microphysics accounts for ~10% of total step time
- Nature of storm simulation implies small region of more intense work; hence, imbalance

Load Balance, Locality in Compute Grid

MPICH_RANK_ORDER file:

```
# grid_order -C -Z -c 2,8 -g 64,64
# Region 0: 0,0 (0..4095)
0,1,64,65,128,129,192,193,256,257,320,321,384,385,448,449
2,3,66,67,130,131,194,195,258,259,322,323,386,387,450,451
4,5,68,69,132,133,196,197,260,261,324,325,388,389,452,453
…
145 2076,2077,2140,2141,2204,2205,2268,2269,2332,2333,2396,2397,2460,2461,2524,2525
146 2078,2079,2142,2143,2206,2207,2270,2271,2334,2335,2398,2399,2462,2463,2526,2527
147 2080,2081,2144,2145,2208,2209,2272,2273,2336,2337,2400,2401,2464,2465,2528,2529
…
3642,3643,3706,3707,3770,3771,3834,3835,3898,3899,3962,3963,4026,4027,4090,4091
3644,3645,3708,3709,3772,3773,3836,3837,3900,3901,3964,3965,4028,4029,4092,4093
3646,3647,3710,3711,3774,3775,3838,3839,3902,3903,3966,3967,4030,4031,4094,4095
```

- **Lowest rank process with MP work : 2076**
- **Highest rank process with MP work: 2273**
- **Lowest MP work process: 2205**
- **Highest MP work process: 2207**

Very little correlation to profiles, other than the minimum-work rank (2205)

RANK REORDERED:

Profile summary	min	max	avg	minPE	maxPE
Wall clock time	59.490	59.670	59.639	3612	4
User processor time	31.054	57.972	56.696	2141	1
Total MPI time	3.540	42.759	38.546	2205	551
Total Wait and Probe time	2.381	38.328	35.493	2205	2266
Waitany total time	1.708	38.145	35.181	2206	551
Data cache accesses	17350.742M	70709.920M	62896.944M	2205	551
L1 Dcache refs	17350.742M	70709.920M	62896.944M	2205	551

DEFAULT:

Profile summary	min	max	avg	minPE	maxPE
Wall clock time	96.060	96.150	96.102	3535	160
User processor time	41.319	94.254	91.807	2205	1905
Total MPI time	13.107	77.723	71.168	2205	3471
Total Wait and Probe time	2.653	63.603	59.466	2205	2232
Waitany total time	1.520	63.285	59.156	2142	1728
Data cache accesses	29262.300M	140634.535M	108415.779M	2205	4057
L1 Dcache refs	29262.300M	140634.535M	108415.779M	2205	4057

Summary

- Be watchful for timings which are not reproducible
  - Check if you are running the same environment
  - Make sure you have compiled the same
  - Lots of sources for “noise” at this scale
    - Lustre ping, other users’ jobs, congestion, bad links
  - Selectively remove “obvious” factors; e.g. I/O
  - Let us know if you see unexplained performance issues – perhaps a configuration has changed
  - System upgrades, new modules, compiler versions
    - This will help us diagnose bugs and report them

References

1. https://bluewaters.ncsa.illinois.edu/monitoring-jobs

2. https://www.olcf.ornl.gov/wp-content/uploads/2013/02/MPI_MPT-HP.pdf

3. https://wiki.ncsa.illinois.edu/download/attachments/24773303/AdvancedFeatures_PRAC_WS_2013-02-27.pdf

4. https://computing.llnl.gov/tutorials/mpi/

5. http://docs.cray.com/books/S-0040-A//S-0040-A.pdf

6. https://cug.org/proceedings/attendee_program_cug2012/includes/files/pap166.pdf

A Quick Primer on Converting MPI to CAF

- Desires:
  - Maintain a single source base
  - Use preprocessing directives as much as possible
  - Minimally invasive – avoid customized coding
  - Do it smartly for performance considerations
  - Use incremental conversion process

Define CAF interfaces to MPI routines:

```
interface
      subroutine mpi_init(ierr)
        integer ierr
      end
end interface

interface
      subroutine MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
        integer MPI_COMM_WORLD, myid, ierr
      end
end interface

interface
      subroutine MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
        integer MPI_COMM_WORLD, numprocs, ierr
      end
end interface
```

Overload interface subroutine names:

```
interface MPI_REDUCE
      subroutine MPI_IREDUCE(time_mpb     ,sum, size, dum1 , dum2 , root, comm ,ierr)
        integer size, dum1 , dum2 , root, comm ,ierr
        integer time_mpb BRACKETS     ,sum
      end
      subroutine MPI_RREDUCE(time_mpb     ,sum, size, dum1 , dum2 , root, comm ,ierr)
        integer size, dum1 , dum2 , root, comm ,ierr
        real time_mpb BRACKETS     ,sum
      end
      subroutine MPI_DPREDUCE(time_mpb     ,sum, size, dum1 , dum2 , root, comm ,ierr)
        integer size, dum1 , dum2 , root, comm ,ierr
        double precision time_mpb BRACKETS     ,sum
      end
end interface
```

Note that the typing of arguments will be handled automatically – no need to change calling interfaces in the source code – only the interface routines… but you still need to write a custom routine for each type of data. This is only half-magic.

Customizing the interface routines:

```
subroutine mpi_init(ierr)
  integer ierr
end

subroutine MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
  integer MPI_COMM_WORLD, myid, ierr
  myid = this_image()-1
  print*,'myid is ',myid
end

subroutine MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
  integer MPI_COMM_WORLD, numprocs, ierr
  numprocs=num_images()
  print*,'num images is ',numprocs
end
```

Main difference between CAF and MPI is 1-based rank indexing vs. 0-based… a great opportunity for errors. Can define co-arrays with different rank ; e.g [0:*]

Custom interfaces, cont.

```
subroutine MPI_IREDUCE(time_mpb     ,sum, size, dum1 , dum2 , root, comm ,ierr)
  integer size, dum1 , dum2 , root, comm ,ierr
  integer time_mpb BRACKETS    ,sum
  print*,'inside of mpi_ireduce'
end
subroutine MPI_RREDUCE(time_mpb     ,sum, size, dum1 , dum2 , root, comm ,ierr)
  integer size, dum1 , dum2 , root, comm ,ierr
  real time_mpb BRACKETS    ,sum
  print*,'inside of mpi_rreduce'
end
subroutine MPI_DPREDUCE(time_mpb     ,sum, size, dum1 , dum2 , root, comm ,ierr)
  integer size, dum1 , dum2 , root, comm ,ierr
  double precision time_mpb BRACKETS    ,sum
  print*,'inside of mpi_dpreduce'
end
```

- Note the different ***actual*** names of the routines, but they will be known as the generic name MPI_REDUCE, as desired.

Errors and Fixes

- PE 187 (1536@nid25126): Failed to register 4877600 bytes of memory starting at 0x49f8f40.

and/or :

- PE 190 (1539@nid25126): DMAPP call failed with error 4 at file craylibs/libpgas/dmapp_interface.c line 376

- setenv HUGETLB_MORECORE yes
- setenv HUGETLB_DEFAULT_PAGE_SIZE 2M
- setenv HUGETLB_ELFMAP W
- setenv HUGETLB_FORCE_ELFMAP yes+