



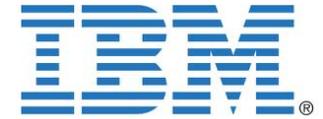
Understanding Application and System Performance Issues On WCOSS Systems

***James Taft
IBM Contractor
jtaft@siennasoftware.net***

January 15, 2013



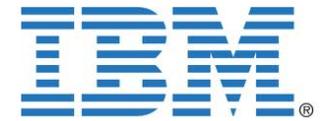
Agenda



- **Kinds of Optimizations**
- **GSI Experience**
- **Conclusions**
- **Suggestions**



Kinds of Optimizations



There are several items to consider when you embark upon an optimization effort on any machine. Each can have significant impact on the total job performance. The major areas of concern are:

- Run time environment
- Compiler/Library Issues
- MPI Issues
- OpenMP issues
- I/O Issues
- Algorithmic Issues
- Coding efficiency



Run Time Environment



The user typically will use a “runit” script to execute his job. This script consists of directions to the system to set up the best system configuration for executing the particular job at hand. Aside from specifying the number of nodes/cpus to run on, the user must also declare the memory requirements, and the types of MPI/OpenMP settings to insure the best communication performance.

- Set node and memory needs
- Set communication protocols
- Set system buffers for I/O and communication
- Set OpenMP parameters
- Set MPI parameters
- Provide instructions for job initialization, run, and post processing cleanup



Sample Runit Script IBMPE - MPICH2



```
#!/bin/bash
#BSUB -J GSI
#BSUB -a poe
#BSUB -W 01:00
#BSUB -o out
#BSUB -e err
#BSUB -n 256
#BSUB -q "hpc_ibm"
#BSUB -R span[ptile=4]
#BSUB -x
```

```
ulimit -s unlimited
```

```
export LANG=en_US
```

```
export MP_EAGER_LIMIT=65536
export MP_EUIDEVELOP=min
export MP_EUIDEVICE=sn_all
export MP_EUILIB=us
export MP_MPILIB=mpich2
```

```
export MP_USE_BULK_XFER=yes
export MPICH_ALLTOALL_THROTTLE=0
export MP_COLLECTIVE_OFFLOAD=yes
```

```
export KMP_STACKSIZE=1024m
export MP_TASK_AFFINITY=core:1
export OMP_NUM_THREADS=2
(time mpirun.lsf ./gsi.x ) > stdout
```

This script is used to execute IBM MPICH2 codes on WCOSS. It is very simple and very few settings are needed by the user. The most common user tunable is the eager setting. Start with the default (64K) and move up/down in multiples of 2 until best results are obtained.



Compilers and Their Impact on Performance



The world is moving to just a handful of compilers. This is good. It means that codes are much more portable. Both Zeus and WCOSS support the Intel ifort, and icc compilers.

Compiler Options - General

There are a myriad of compiler options. Many sound very interesting and would appear to offer enhanced performance if invoked. This is generally not the case. Don't be led into using a series of exotic options to get 1 or 2% better performance. Your code will be more subject to compiler issues downstream as compiler upgrades come along. Use the fewest possible.

Compiler Options - Recommended

-O3 that's it. Use -O2 if this breaks

Compiler Options – If needed

byte_recl, r8, i8, extended_source, fp_model=strict



Parallel Processing Paradigms



Users are continually assaulted with calls for increased scaling for almost all codes in existence on HPC systems. There are two major ways to achieve this. These are:

OpenMP Threading

This is a simple compiler directive based form of parallelism that will allow the user to use simple compiler directives to instruct the compiler to generate parallel threads to execute subsections of the code at hand. This is often called a fine-grained parallel approach. It works only within a node.

MPI

This is a message passing based approach to parallelism that allows the user to explicitly sub-divide the computational burden across a given number of IB connected nodes on the system. This is often called a coarse grained approach.

HYBRID MPI/OpenMP

A combination of both. Sometimes used to attempt a better load balance of the work, by adjusting the thread counts per process to speed up slower sub-blocks of the computation. Running with a constant thread count for each process is becoming less useful today due to HW issues limiting its scaling.



What kind of parallelism should you use



Architectural Issues

Modern HPC architectures are evolving to a common design across almost all vendors. This architecture is a system of IB coupled modest core count SMP nodes running linux, talking to a global file system that is either Lustre or GPFS. This common architecture has been in place for about a decade and will be in effect for the next many years.

Historical Approaches

In the past users were typically using a single level of parallelism (MPI) to run in parallel. More aggressive users added an underlying 2nd level (OpenMP) to handle some load balance or exotic failure to scale issues.

My Suggestion

After many years of writing code across many scientific disciplines, it is my opinion that one should adopt MPI first and use OpenMP only sparingly when absolutely needed. The GSI example later in this talk gives some credence to this thinking.



MPI Communication Issues



MPI simply dominates the modeling community when it comes to building parallel code. Many codes are coarse grained parallelized using domain decomposition. That is, the total simulation is broken into many smaller sub-domains each running as a separate process and each responsible for a small fraction of the total problem. The user invokes the MPI communication library to exchange data and boundary conditions as needed between them. Synchronizations are also done to insure the total solution moves forward in a lock step manner.

The use of MPI is a complex topic and requires much discussion. The bottlenecks typically experienced with MPI have evolved over the years. In the past network bandwidth and latency issues were extensive. Current systems are much more robust and scaling has improved dramatically. Some of the issues to watch are:

- Overuse of MPI_ALLTOALLV
- Excessive barriers (explicit or implied)
- Short message exchanges in tight loops



OpenMP Issues



There are very few pure OpenMP parallel codes. In my experience I have seen 1 in production and it was built from scratch to be OpenMP. Almost all other codes that use OpenMP are “hybrid” codes in that they do a coarse grained parallel decomposition using MPI, and a fine grained “loop” level parallelization under this using OpenMP compiler directives. This can be a terrific combination, providing the science supports it, and the user is judicious its application. Some of the issues to worry about are:

- Overuse – Some loops run better serial to avoid cache thrash
- OpenMP applied at the wrong point – higher is better
- Roundoff issues – some constructs don’t guarantee repeatability
- Scaling Issues due to hardware – simple memory system fails to scale
- Sometimes single level parallel with MPI is better – better decomposition



The Issues with I/O

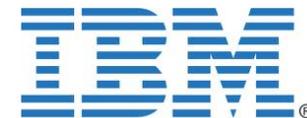


In the weather and climate community it is often the case that I/O can consume significant fractions of the total run time of a job. It is not uncommon to see I/O consume 20-50% of the total run time. Often, there are a number of ways to improve on this, and it is common to see factors of 4x or more when additional optimization efforts are made. This is particularly true when users are moving to systems with new I/O architectures. Some of the major issues with I/O are:

- Overuse of user and system buffer space
 - Do careful with buffers. Codes often use extensive buffer space
- Too many readers/writers
 - Many modern I/O architectures do not respond well to a flood of I/O requests
 - Often one reader/writer with MPI broadcast is much better
- Large numbers of short records
- Big/Little Endian conversions
- Overuse of Direct Access I/O when Binary will do



Decomposition/Algorithmic Issues



Users have developed a number of popular approaches to solving the equations of motion and physics sub-models in the climate and weather communities. It is important to match these solution techniques to the features of the architectures at hand.

As hardware moves to supporting larger and larger multi-core chips, and faster lower latency interconnects, the user must re-evaluate what makes sense moving forward.

Historically we have seen differing styles of grid decomposition, spectral vs spatial approaches, etc. Current systems tend to favor pure real space solution approaches, but there are many spectral models that remain in use.

Icosahedron based meshes are gaining in popularity and map well on the new systems. They also have some advantages in handling the poles in global simulations, in BC exchanges, and can be readily coded using MPI/OpenMP hybrid approaches.



Coding Efficiency



Codes develop over many years and are the product of a diverse group of developers. This allows the community to develop highly sophisticated simulation models in the shortest time, but also almost guarantees that significant sections of the codes are written in less than optimal fashion. Historically, most codes can expect to gain a factor of 2x or more with additional optimization work.

When making a second pass for optimization after initial development, one should look at the following:

- Extravagant use of memory
- Too many short loops and extraneous temporaries that thrash cache
- Poor choices of library routines or using home grown alternatives
- Poor loop ordering



Suggestions



This discussion touches on a number of areas that are complex and require substantial discussion to understand fully. I suggest that we have a continuing education program (perhaps a 1/2 day session every month) that addresses each of the topics below. I propose that it be a “hands on” session with the class watching a live walk through of an assessment using real codes provided by NCEP. We would cover the following topics at a minimum:

- Timing codes for optimization in performance
- Discovering memory requirements
- I/O Optimization strategies
- Identifying scaling issues in MPI and OpenMP



Finding your Memory Occupancy



There is a straightforward way to determine the ever changing memory requirement of your code as it executes. It basically involves calling the routine “getrusage” at various stages in the execution of the code and printing out the result. You can then determine exactly which routine is taking all the memory and evaluate whether it can be fixed or not. An example of a simple calling sequence is:

```
call getmemuse(mbytes)
write(1000+mype,222) mbytes
222 format('GETRUSAGE resident memory is:', i6,' GB')
call flush(1000+mype)
```

This will produce prints to a series of files tagged with the process number + 1000. The user can then grep through the files looking for the process or processes that take all the memory. Often times it is the master that is the culprit as it often has extra large buffers for I/O operations. Some codes have these buffers for all processes, though they don't need them.

This process is excellent in not only reducing memory needs, but also can boost performance as the resulting smaller executables often run better.



Code Died – You don't Know Why



Unfortunately, this happens a lot more than anyone would like on many machines. For some reason the system simply doesn't clean up after a trap, and you as the user have no clue as to where the code died, and for what reason. The following will help a lot. The process is to insert the prints below into the code in various locations. Usually, a dozen or so is adequate.

```
call getmemuse(mbytes)
write(1000+mype,222) mbytes
222 format('GETRUSAGE resident memory is:', i6,' GB I got to 1')
call flush(1000+mype)
```

What you get is a series of files that prints out how far each process got and its memory needs at that point. You can quickly see which process didn't make it to the next print and how much memory is being used just before it died (out of memory is often the reason for many codes dying).



Three Things about the Memory System

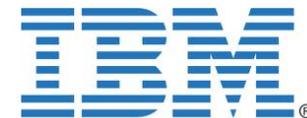


Modern microprocessors differ in the fine details, but overall they are extremely similar in their architectural features. The most important features for all microprocessors that impact user code performance are given below:

- Caches – there are a lot of them – usually 3 now
 - Caches are getting smaller – there is more reliance on memory access
 - Contention effects are getting worse
 - Can be 30% hit easily
- It's NUMA Memory – all systems have NUMA effects
 - Sometimes true NUMA – sometimes bad pinning
 - Can be 30% hit easily
- TLB Issues
 - You can only “see” memory pages in the TLB
 - If it isn't in the TLB you take huge hit to update TLB
 - Random walks through memory are bad here
 - Can be 30% hit easily



Process/Thread Placement – Performance



Ever since SGI built the first NUMA memory system in its Origin series almost 20 years ago, the user community has had to worry about process and/or thread placement. In particular, such systems are highly sensitive to how far the memory is from the CPU actually running the thread or process. In true NUMA systems this can be a number of “hops” away. Each hop adds latency and perhaps bandwidth degradation in the access of the cache line from memory.

In newer systems the nodes are small and the memory is at most 1 or 2 hops away. Even so, this is significant and can reduce effective memory speed by 2x in some cases.

Part of the problem that greatly exacerbates the problem is that the system may be stupid and inadvertently place the process far from its memory. This can be overcome by intelligent pinning of the processes close to the memories that they will use. Currently, batch schedulers, and/or MPI launchers do this function. It is done well by most systems today. Though hybrid codes may be problematical.



Computing with 80 Bit Precision



Since the world is almost exclusively using x86 hardware at this point, users are beginning to gain some advantage in the precision of their computations. This is because the x86 architecture does its register to register floating point work in 80 bit precision., a carry over from the 287 math coprocessor days of 20+ years ago.

This can have significant stability advantages in large problems where roundoff is always an issue. For NCEP codes the users first notice the effect when they see the results on x86 systems differ from that of CCS. The first thought is that the new systems are giving wrong answers. The actual fact is that they are giving better answers and the roundoff differences are the result of increased mathematical precision. Users frequently turn this off with the use of the “fp_model strict” flag that brings the precision back to IEEE 64 bit standards.



Using Libraries - MKL



MKL is the default math and science library provided with Intel compiler suite. It is significantly optimized and should be used whenever possible. Generally, it is highly reliable, and has the benefit of literally millions of users providing input to Intel whenever an accuracy or performance issue is discovered.

Users should abandon any custom routines that provide the same functions. There is no need to maintain such source and often times it is substantially less efficient.

Also, MKL is an OpenMP threaded library that allows the user to use it liberally in OpenMP sections without having to modify custom routines for that purpose.



MPI Programming Strategies



The philosophy behind MPI programming can be summed up in a few simple statements. They are:

- Communicate a little as possible
- Avoid extraneous explicit barriers
- Avoid extraneous implicit barriers in the global operations too
- In particular, avoid `MPI_ALLTOALLV` calls if at all possible
- Beware of `MPIIO` – It is generally not supported well on systems with weak I/O
- Keep it SIMPLE – Virtually every code is overly complex in MPI
- Use `MPI_ANY_SOURCE` whenever possible
- Don't bother with buffered or asynchronous send/receives – it very rarely helps



FORTRAN 9x – Issues and Helpful Hints



FORTRAN 9x has been around for decades at this point. Every year more object oriented and C like features are added to its syntax. I would suggest that you avoid most of them for best performance, code readability, and code stability. In particular:

- Avoid the use of pointers
- Avoid the use of interface definitions
- Avoid the user of optional arguments
- Use array syntax sparingly
- Use Modules sparingly – Do not use the CONTAINS feature
- Build source with a single extension type such as .F or .f



The TLB Problem



There is a Translation Look-aside Buffer (TLB) in every modern microprocessor. This buffer is a fundamental part of the hardware, and consists of 10s to 1000s of entries, depending on the manufacturer and the age of the processor. Simply put, The size of the TLB determines the number of active pages that can be addressed by a user program at any given time. If the user accesses data with an address not in the TLB active list, a serious hit is taken (1000s of clocks) as the TLB is updated with the new page.

Usually, users don't care about the TLB, but codes that are doing a lot of random lookup can experience a significant reduction in performance because of it.

Because the system page size under Red Hat Linux is by default only 4K bytes, TLB misses can occur in surprising places, such as running through 3D loops in the wrong order.

Many systems boot with larger page sizes to reduce the impact of TLB misses (larger page sizes allow the TLB to cover more user address space – thus fewer chances to encounter a miss). This is an experiment we should do here.

FYI, It was observed at NASA that a default page size of 64K bytes would often increase code performance by 15-20%.



Memory Fetch Reduction Is all Important



Floating Point is Free

In the “old” days of Cray vector machines, memory fetches were essentially hidden behind the floating point work. Thus, the MFLOP number quoted by Cray users was a true reflection of the runtime of the code.

Today, just the opposite is true. Floating point work is essentially free. One can perform ~1000 floating point operation in the time it takes to fetch a single cache line from memory.

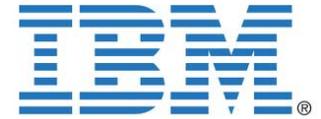
This stunning turnaround is due to the need for commodity microprocessor manufacturers to build simple inexpensive memory systems that can be inserted into inexpensive desktop workstations and laptops.

Unfortunately, simple memory systems mean long latency times for any memory fetch and a guarantee that most real codes will only run a small percentage of the peak performance of the system CPU.

Users must strive to reduce spurious memory fetches to the absolute minimum to get better code performance.



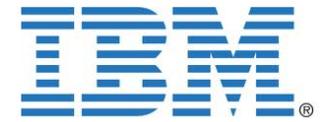
Cache hit rate is 99% - Big Deal



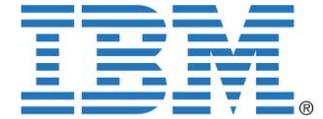
Most hardware system designers like to quote that their platform is designed so well with complex mult level caches that most codes get cache hit rates of 90+ percent.

My response is “big deal”. The problem we have is that you need a hit rate of 99.9% to really get a significant percentage of the peak performance of the machine. A simple idealized example shows why:

- 1) Memory access time: 100 clocks
- 2) 99 out of 100 fetches in cache
- 3) That means it takes 99 clocks for 99 operands and 100 clocks for the next - 199 clocks
- 4) 199 clocks to get 100 operands means that you are running half as fast as all cache run
- 5) Thus, you are running $\frac{1}{2}$ the peak of the machine with only 1 cache miss.



Tide/Zeus GSI Experiences



GSI Optimization Work

A Platform Independent Generic Modification Effort

The GSI work on Zeus/Tide has been crafted as a generic effort that will allow the code to run significantly better on all platforms at hand. The work was focused on just a few areas as the result of extensive timing studies which showed some “low hanging fruit” that could seriously impact performance with a modest effort. Some features of the effort are:

- All optimizations resulted in ZERO differences in all output of any kind
- The work was mainly focused on the GETUV and SMOOTHRF routines and their children
- Some work was focused on MPI performance issues with ALLREDUCE and ALLTOTALLV
- I/O was not addressed at all – it is about 10% burden for the data sets given



PCGSOI Runtime Over the history of Modifications



This chart shows the evolution of the crunch part of the code as the optimizations took place over the course of this project.

```
time.out.v0001.480procs.01threads:PCGSOI      : 1186.0 1186.  
time.out.v0017.480procs.01threads:PCGSOI      : 1103.2 1103.  
time.out.v0018.480procs.01threads:PCGSOI      : 1096.1 1096.  
time.out.v0020.480procs.01threads:PCGSOI      : 1079.4 1079.  
time.out.v0021.480procs.01threads:PCGSOI      : 1050.4 1050.  
time.out.v0025.480procs.01threads:PCGSOI      : 1006.7 1006.  
time.out.v0029.480procs.01threads:PCGSOI      :  977.0  977.  
time.out.v0031.480procs.01threads:PCGSOI      : 1011.1 1011.  
time.out.v0033.480procs.01threads:PCGSOI      :  971.7  971.  
time.out.v0034.480procs.01threads:PCGSOI      : 1026.3 1026.  
time.out.v0042.480procs.01threads:PCGSOI      :  906.8  906.  
time.out.v0044.480procs.01threads:PCGSOI      :  869.4  869.  
time.out.v0050.480procs.01threads:PCGSOI      :  865.4  865.  
time.out.v0057.480procs.01threads:PCGSOI      : 1109.3 1109.  
time.out.v0060.480procs.01threads:PCGSOI      : 1000.5 1000.  
time.out.v0062.480procs.01threads:PCGSOI      :  877.3  877.  
time.out.v0068.480procs.01threads:PCGSOI      :  893.8  893.  
time.out.v0072.480procs.01threads:PCGSOI      :  912.8  912.  
time.out.v0073.480procs.01threads:PCGSOI      :  864.7  864.  
time.out.v0074.480procs.01threads:PCGSOI      :  885.0  885.  
time.out.v0076.480procs.01threads:PCGSOI      :  888.2  888.  
time.out.v0077.480procs.01threads:PCGSOI      :  951.8  951.  
time.out.v0080.480procs.01threads:PCGSOI      :  824.6  824.  
time.out.v0084.480procs.01threads:PCGSOI      :  752.9  752.  
time.out.v0085.480procs.01threads:PCGSOI      :  804.4  804.  
time.out.v0086.480procs.01threads:PCGSOI      :  715.4  715.
```



New MPI_ALLTOALLV - Blocking Send/Receives



This code duplicates the run time performance of MPI_ALLTOALLV on Zeus.

```
subroutine jmpi_alltoallv(sbuf,icnt,ioff,mtypes,rbuf,jcnt,joff,mtyper,mdum,ierror)

use mpimod, only: mype,npe

include "mpif.h"
include "common.inc"

real*8 sbuf(*)
real*8 rbuf(*)

integer ioff(npe),joff(npe),icnt(npe),jcnt(npe)

integer istat( MPI_STATUS_SIZE )

!-----sends
do n=1,npe
isend=mype+(n-1)
if(isend.ge.npe) isend=isend-npe
iblck=isend+1
if(icnt(iblck).gt.0) call mpi_send(sbuf(ioff(iblck)+1),icnt(iblck),mpi_real8,isend,0,mpi_comm_world,ierror)

!-----receives
jrecv=mype-(n-1)
if(jrecv.lt.0) jrecv=jrecv+npe
jblck=jrecv+1
if(jcnt(jblck).gt.0) call mpi_recv(rbuf(joff(jblck)+1),jcnt(jblck),mpi_real8,jrecv,0,mpi_comm_world,istat,ierror)
enddo

return
end
```



Why Stick with MPI Only



I argue that we stick with MPI only as far as possible. The reasons are:

MPI generally scales more efficiently than OpenMP hybrid alternatives.

Modern HW likes coarse grained decompositions of user code. This is particularly true on the new standard for HPC architectures where OpenMP scaling is usually only good on two cores, modestly good on 4 cores, and basically worthless on more.

Occasionally OpenMP Hybridization helps:

Some codes need help in code sections that run on fewer CPUs due to the physics or the shape of the domain decomposition. Some need it because of MPI latency issues that inhibit scaling.

A simple example where it could help

For GSI, some sections only scale to 64 processes due to parallelization across the number of grid levels. Here it would be nice to suddenly ramp up performance with an OpenMP section. However, further examination of the code shows that a rewrite will get us a lot farther.



GSI – Code Decomposition Study



GSI is a good example of a code that can benefit from single MPI parallelism with some possible extension to OpenMP for certain code segments, at least at first blush. The chart below graphically presents the logical flow through the computation and the limits to scaling. Most sections run on 480 processes, but two areas run on 64 and 385 processes respectively.

Workload Over Time





GSI – Code Decomposition Study



1st – GSI benefits not at all from converting any code in the 480 sections to a MPI/OpenMP hybrid mix. The code runs better with pure MPI parallelism. One can divert all focus to the the 64 and 385 way sections. Fortunately, these are relatively small code sections totaling a couple of thousands of lines.

2nd –The 64 way code section will not benefit much from adding OpenMP. The reason is that running on two threads might get you another factor of two in performance, but then you are out of business as 4 threads is much less of a payoff, and more than 4 is a waste of time. Also using OpenMP here requires some fancy tricks not to disturb the remainder of the code. One would like to run this on 480 cores and it looks like a different decomposition will get us there with some creative coding. So the end result is that OpenMP is not needed here at all.

3rd – The 385 process section is a bit more subtle. It is already relatively high in core count, and the first order of business would be to improve the single CPU performance to make it less impactful on run time. This has been done and a 2x win was obtained. Further work with OpenMP is not fruitful as one must have a total core count that matches 480 to match up with the rest of the code. Alternatively you could change the core counts, but it is complex how to divide the 385 tasks well (a fixed number for a given data set).



Interesting Behavior When Running Hybrid Code on Zeus/TIDE

One should note that most codes in HPC are memory bound. They are not doing well at reusing cache. Also note that as core counts go up on new products, the trend will be to reduce cache size per core over time. This will make cache reuse even tougher to achieve. I have observed a couple of things on Zeus, which will be true of other HPC architectures. These are:

- Running an N process count memory bound code on N adjoining cores will run in about the same time as a code running N/2 processes running on N cores where we are set up to run on every other one.
- Running an N process code will run in about the same time as an N/2 process hybrid code where we are set up to use 2 threads per MPI process.
- Running on higher thread counts generally throws CPU cycles on the floor. If it scales well on MPI it will not scale nearly as well with a hybrid running 4 or more threads.
- There are known bugs and performance issues with OpenMP – why add stress to your coding efforts if it is not needed – avoid extraneous OpenMP work.



Sample TIME.OUT Output



READEM Module Summary

```

-----
Process Number:      1      5      9      13      17      21      25      29      33      37      41      45      49      53      57
Process Number:      61     65     69     73     77     81     85     89     93     97    101    105    109    113    117
Process Number:     121    125    129    133    137    141    145    149    153    157    161    165    169    173    177
Process Number:     181    185    189    193    197    201    205    209    213    217    221    225    229    233    237
Process Number:     241    245    249    253
-----

```

```

-----
READEM      :  154.1  153.7  154.2  153.7  153.8  153.4  152.9  153.1  153.9  153.3  153.2  153.3  152.9  152.4  152.2
              151.6  154.1  154.3  153.7  153.5  154.3  153.7  153.1  153.6  153.8  153.0  152.6  152.8  154.9  153.2
              153.5  153.0  154.2  152.0  153.0  152.8  154.1  152.2  153.1  152.5  153.7  151.8  152.9  152.2  154.1
              152.7  153.0  153.1  153.4  152.4  152.5  152.7  153.0  152.8  152.1  152.1  153.2  153.1  152.4  151.9
              153.3  152.5  152.2  152.1

READS      :  114.9   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
              0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
              0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
              0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

SENDS      :   43.9   6.0   5.8   5.9   6.2   6.4   6.7   6.5   6.3   6.4   6.0   6.0   7.0   6.9   7.0
              7.0   6.4   6.3   6.3   6.2   6.1   6.2   6.3   6.3   6.2   6.3   6.3   6.4   5.2   5.7
              5.8   5.9   5.7   6.0   5.7   5.6   5.7   5.7   5.6   6.0   5.9   5.9   5.7   6.1   5.5
              5.3   5.9   5.4   6.0   5.9   6.3   6.0   6.5   6.3   6.7   6.4   5.7   5.5   5.6   5.7
              6.0   6.3   6.0   5.9

RECEIVES   :    0.0  153.7  154.1  153.6  153.8  153.3  152.9  153.1  153.8  153.3  153.2  153.3  152.8  152.4  152.2
              151.5  154.1  154.3  153.7  153.5  154.3  153.6  153.0  153.5  153.8  153.0  152.6  152.7  154.8  153.2
              153.5  153.0  154.1  152.0  153.0  152.7  154.0  152.2  153.1  152.5  153.7  151.8  152.9  152.1  154.1
              152.7  153.0  153.1  153.3  152.4  152.5  152.7  152.9  152.8  152.1  152.1  153.2  153.1  152.4  151.9
              153.3  152.5  152.1  152.1
-----

```



Sample FORT.301 Output seen in Excel



GSF_ESMF	92.6	92.6	92.6	92.6	92.6	92.6	92.6	92.6	92.6
PREAMBLE	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9
ESMF_VMBarr	0	0	0	0	0	0	0	0	0
GFS_RUN	90.7	90.7	90.7	90.7	90.7	90.7	90.7	90.7	90.7
ESMF_CplCom	0	0	0	0	0	0	0	0	0
ESMF_VMBarr	0	0	0	0	0	0	0	0	0
GFS_Run	0	0	0	0	0	0	0	0	0
GFS_Finalize	0	0	0	0	0	0	0	0	0
ESMF_CplCom	0	0	0	0	0	0	0	0	0
GFS_Run_ESM	90.7	90.7	90.7	90.7	90.7	90.7	90.7	90.7	90.7
ENSEMBLE_W	0	0	0	0	0	0	0	0	0
TLDFI	4.6	4.6	4.6	4.6	4.6	4.6	4.6	4.6	4.6
GET_CD_...	0	0	0	0	0	0	0	0	0
DO_TSTEP	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
GET_CD_...	0	0	0	0	0	0	0	0	0
SYNCHRO	0	0	0	0	0	0	0	0	0
DO_TSTEP	85.5	85.5	85.5	85.5	85.5	85.5	85.5	85.5	85.5
DO_TSTEP	85.5	85.5	85.5	85.5	85.5	85.5	85.5	85.5	85.5
GLOOPA	14.2	14.6	15	13.3	14	14.4	14.9	14	14.2
GCYCLE	0	0	0	0	0	0	0	0	0
ATM_GETSSTI	0	0	0	0	0	0	0	0	0
GLOOPR	30	20.1	20.4	30.6	30.6	20.7	20.5	30.5	30.5
SICDIFE_HYB	0	0	0	0	0	0	0	0	0
SICDIFO_HYB	0	0	0	0	0	0	0	0	0
SICDIFE_HYB	1	0.6	0.5	1	0.9	0.5	0.5	0.9	0.9
SICDIFO_HYB	1	0.5	0.5	0.9	0.9	0.5	0.5	0.8	0.8
SICDIFE_HYB	0	0	0	0	0	0	0	0	0
SICDIFO_HYB	0	0	0	0	0	0	0	0	0
LOOP	0	0	0	0	0	0	0	0	0
BCST	0	10.7	10.5	0.2	0.3	11	11.2	0.5	0.5
DELDIFS	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
FILTR1EO	0.1	0.1	0	0.1	0.1	0.1	0	0.1	0.1
GLOOPB	36.3	35.7	35.6	35.7	35.8	35	34.8	35.7	35.6
DUMP_SPEED	0	0	0	0	0	0	0	0	0
FILTR2EO	0.2	0.1	0.1	0.2	0.2	0.1	0.1	0.2	0.2
WRTOU	7.6	8	7.8	8.3	7.7	8.3	8	7.7	7.7
RESHUFF	0	0	0	0	0	0	0	0	0
ATM_SENDFLL	0	0	0	0	0	0	0	0	0
GLOOPR	30	20.1	20.4	30.6	30.6	20.7	20.5	30.5	30.5
RADINIT	0	0	0	0	0	0	0	0	0
ASTRONOMY	0	0	0	0	0	0	0	0	0
DELNPE	0	0	0	0	0	0	0	0	0
DELNPO	0	0	0	0	0	0	0	0	0
DEZOUV/DOZI	0	0	0	0	0	0	0	0	0
SUMFLNA_R	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3
SUNDERA_R	0	0	0	0	0	0	0	0	0
FOUR2GRID_t	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
LOOPS	29.5	19.7	20	30.1	30.1	20.3	20.1	30.1	30.1
GGRAD	29.3	19.6	19.9	30	29.9	20.1	20	29.9	29.9
sw	4.2	2.4	2.4	4.1	4.1	2.3	2.4	4	4
lw	24.6	16.8	17.1	25.3	25.3	17.4	17.3	25.3	25.3
GLOOPB	36.3	35.7	35.6	35.7	35.8	35	34.8	35.7	35.6

3/23/12

James Taft

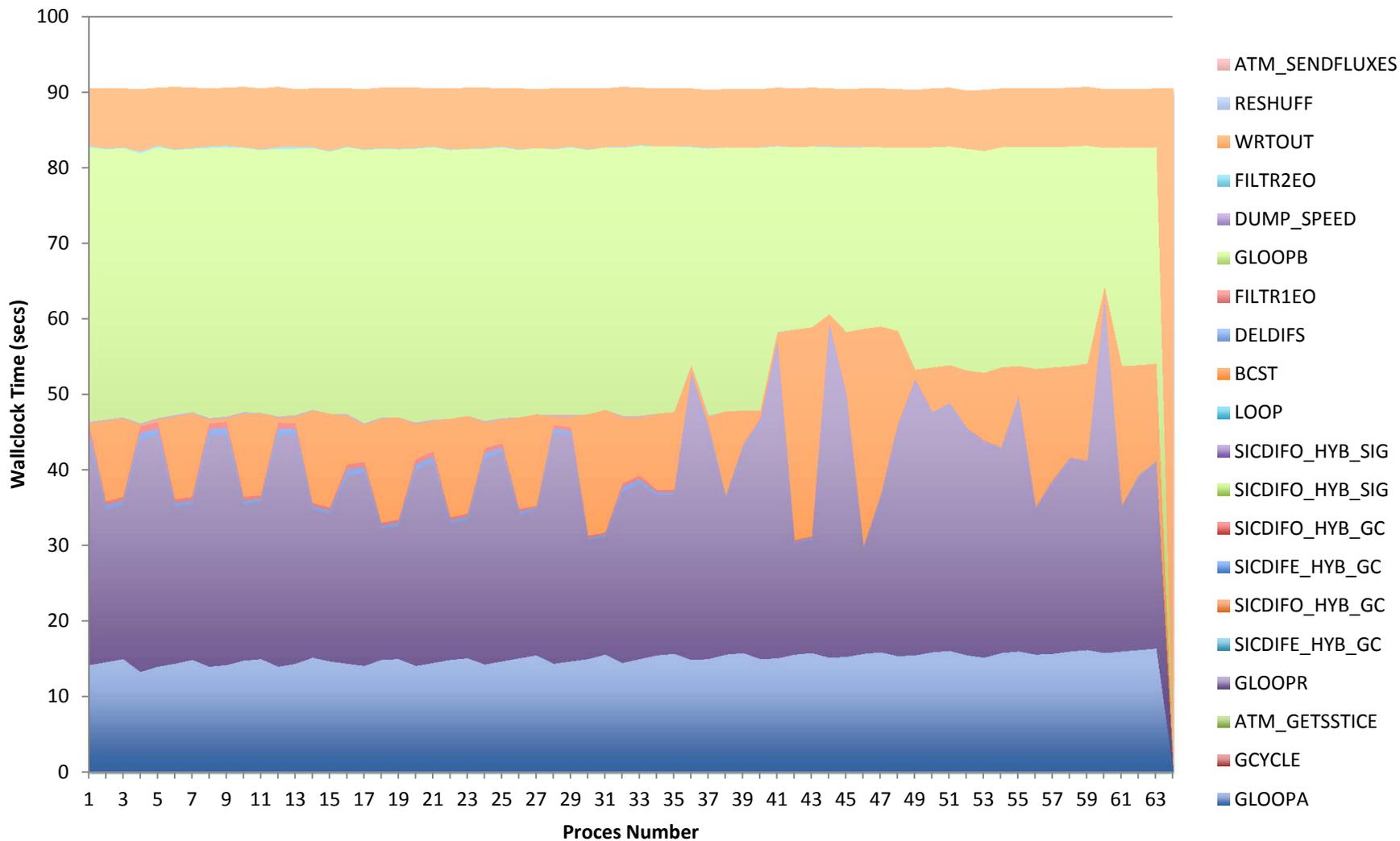


Sample EXCEL Plot from fort.301



DO_TSTEP Timings by Process (secs)

120hrs - 64 procs/4threads



3/23/12

James Taft



Tips That Have Helped My Productivity



I have worked in this business 43 years. I have seen hundreds of HPC applications in a wide range of disciplines. That has provided a unique perspective on what works and what is easiest to maintain. I have developed a number of programming rules that I adhere to that make my job easier. They are:

Building Code:

- o Simplify the Makefile and always build it to run parallel
- o Build standalone Makefiles – no recursion or nesting
- o One Makefile per system philosophy

Writing Code:

- o Put one subroutine in each file. Name the file the same as the routine
- o Never use “CONTAINS” construct.
- o Never ever use the “INTERFACE” construct
- o Minimize the use of modules to the bare essentials
- o Always program to fixed field .f files
- o Avoid wordy documentation. it obscures the code

Running the code:

- o Spend some time on simplifying your runit scripts. They are always too



Issues with NETCDF



During my experiences with NETCDF and the CMAQ code over a year ago, I found that the CMAQ code was using an extraordinary amount of memory per process on CCS.

This was due to the fact that NETCDF routines like XINTERP etc were using HUGE memory buffers to hold planes of global data in each process, while sub-sets were extracted for the process to use in subsequent calculations.

A rewrite of XINTERP was done and the CMAQ code went from 3GB of memory per process to 200MB per process - a 15x reduction in total memory required.

This issue was reported to the NETCDF developers. I am not if the new libraries have been fixed.



Summary and Conclusions



There are many issues that must be considered in order to get the most out of a code on any given system. Run time environment, compilers, MPI communication strategies, OpenMP scaling, I/O approaches, and algorithmic issues can each significantly impact the performance of any code.

The current discussion attempted to point out some of the more important aspects to these issues. Several general strategies were identified that can be used in code development and optimization that should be applicable to emerging general purpose hardware architectures for at least the next 5 years.