Threading Tradeoffs in Domain Decomposition

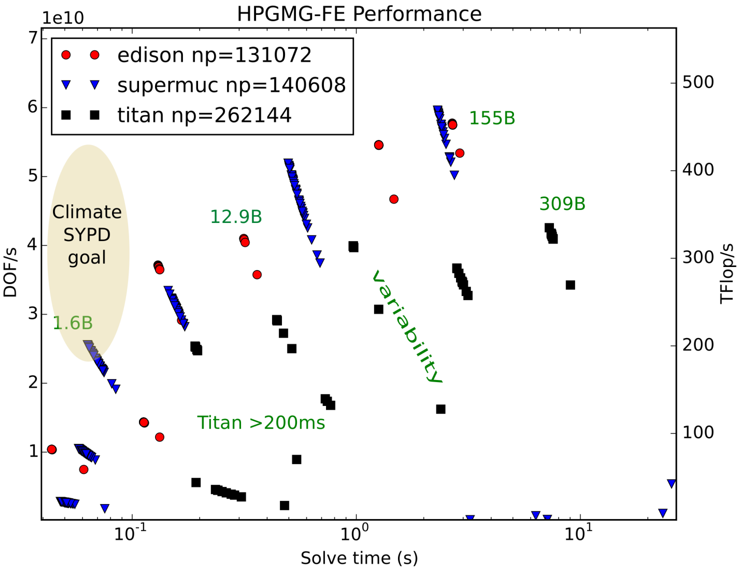
Jed Brown

Collaborators: Barry Smith, Karl Rupp, Matthew Knepley, Mark Adams, Lois Curfman McInnes

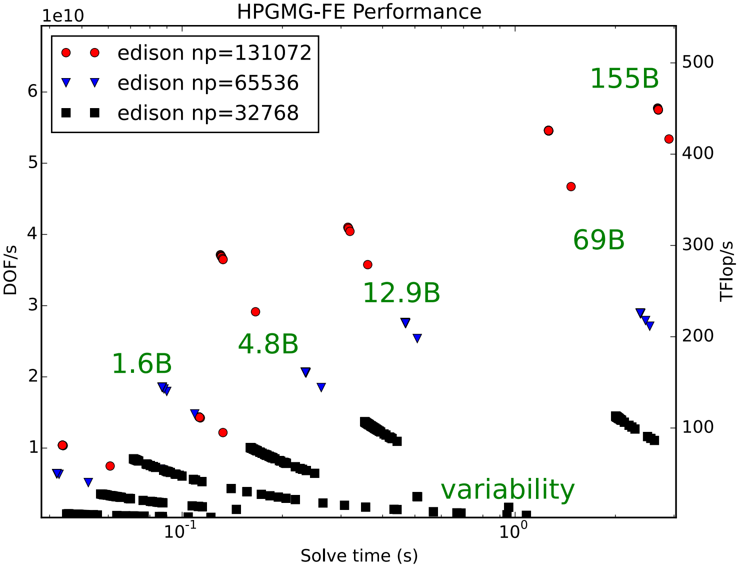
CU Boulder

SIAM Parallel Processing, 2016-04-13

Scaling regime: HPGMG-FE on Edison, SuperMUC, Titan



Scaling regime: HPGMG-FE on Edison at various scales



CAM-SE dynamics numbers

- 25 km resolution, 18 seconds/RK stage
- Current performance at strong scaling limit

Edison	3 SYPD
Titan	2 SYPD
Mira	0.9 SYPD

- Performance requirement: 5 SYPD (about 2000x faster than real time)
  - 10 ms budget per dynamics stage
  - Increasing spatial resolution decreases this budget (CFL)
- Null hypothesis: Edison will run ACME faster than any DOE machine through 2020
  - Difficult to get large allocations

Party line

- Processes are heavy abstractions compared to threads
- Halo exchange is expensive – sharing is better
- OpenMP is lighter weight than MPI
- Processes have substantial memory overhead

Question

What is the difference between a thread and a process?

Question

What is the difference between a thread and a process?

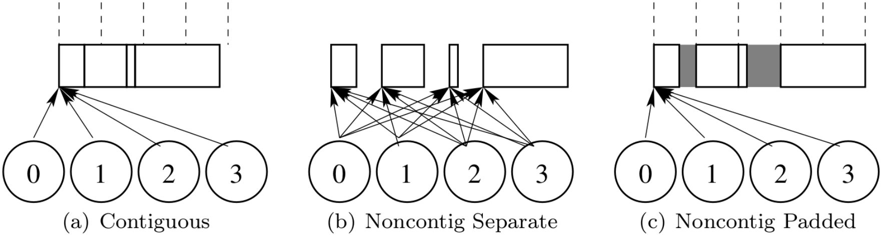
- Both are created using `clone(2)`
- Equivalent entries in kernel data structure
- Threads use `CLONE_VM`, processes have copy-on-write
- Rule of thumb
  - Threads cost $10\mu s$ to create
  - Processes cost $100\mu s$ to create
  - No difference in context switching
  - Only paid once – everyone uses thread pools anyway

Portable shared memory between MPI processes

- MPI-3 portable shared memory windows
- `MPI_Comm_split_type(comm, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &newcomm);`
- `int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win);`

[Hoefler et al, MPI+MPI, 2013]

Halos or contiguous memory?



(a) Contiguous (b) Noncontig Separate (c) Noncontig Padded

- Common assumption: halo copying is expensive
- Alternative is shared memory
- Cache utilization for 16^3 local domain with halos
  - Entire local region is contiguous; no partially filled cache lines
  - $18^3 * \texttt{sizeof(double)} = 46656B$
- 16^3 local domain embedded in contiguous memory
  - Avoid false sharing: align owned portion to cache-line boundaries
  - $32 \times 18 \times 18 * \texttt{sizeof(double)} = 82944B$
  - False sharing a serious problem if local sizes not divisible by line size

Messaging from threaded code

- Off-node messages need to be packed and unpacked
- Many MPI+threads apps pack in serial – bottleneck
- Extra software synchronization required to pack in parallel
    - Formally $O(\log T)$ critical path, T threads/NIC context
    - Typical OpenMP uses barrier – oversynchronizes
- `MPI_THREAD_MULTIPLE` – atomics and $O(T)$ critical path
- Choose serial or parallel packing based on T and message sizes?
- ≥ 1 hardware NIC context/core now, maybe not in future
- What is lowest overhead approach to message coalescing?

But processes can't work for hyperthreads (?)

- Can processes hyperthreaded onto the same core share L1 cache?
- Modern caches are physically tagged
  - Identical cache sharing to threads
- TLB is not shared between processes
  - Is your application TLB-limited?

But processes can't work for hyperthreads (?)

- Can processes hyperthreaded onto the same core share L1 cache?
- Modern caches are physically tagged
  - Identical cache sharing to threads
- TLB is not shared between processes
  - Is your application TLB-limited?

But processes can’t work for hyperthreads (?)

- Can processes hyperthreaded onto the same core share L1 cache?
- Modern caches are physically tagged
    - Identical cache sharing to threads
- TLB is not shared between processes
    - Is your application TLB-limited?

Does the code need to look different?

```c
void Laplace3D(int xs,int xm,int ys,int ym,
      int zs,int zm,double ***x,double ***y) {
  int i,j,k;
  for (i=xs; i<xs+xm; i++) {
    for (j=ys; j<ys+ym; j++) {
      for (k=zs; k<zs+zm; k++) {
        y[i][j][k] = 6*x[i][j][k]
            - x[i-1][j][k] - x[i][j-1][k]
            - x[i][j][k-1] - x[i+1][j][k]
            - x[i][j+1][k] - x[i][j][k+1];
      }
    }
  }
}
```

- No `const`, no `restrict`, soooo much pointer indirection.

Assembly from `gcc -O3`

```
<Laplace3D+0x3bc> mov     rax,QWORD PTR [rbp-0x50]
<Laplace3D+0x3c0> add     r8d,0x1
<Laplace3D+0x3c4> vmovapd ymm0,YMMWORD PTR [r9+rcx*1]
<Laplace3D+0x3ca> vfmsub213pd ymm0,ymm4,YMMWORD PTR [rax+rcx*1]
<Laplace3D+0x3d0> mov     rax,QWORD PTR [rbp-0x58]
<Laplace3D+0x3d4> vsubpd ymm0,ymm0,YMMWORD PTR [rax+rcx*1]
<Laplace3D+0x3d9> mov     rax,QWORD PTR [rbp-0x60]
<Laplace3D+0x3dd> vsubpd ymm0,ymm0,YMMWORD PTR [rax+rcx*1]
<Laplace3D+0x3e2> mov     rax,QWORD PTR [rbp-0x68]
<Laplace3D+0x3e6> vsubpd ymm0,ymm0,YMMWORD PTR [rax+rcx*1]
<Laplace3D+0x3eb> mov     rax,QWORD PTR [rbp-0x78]
<Laplace3D+0x3ef> vsubpd ymm0,ymm0,YMMWORD PTR [rax+rcx*1]
<Laplace3D+0x3f4> mov     rax,QWORD PTR [rbp-0x80]
<Laplace3D+0x3f8> vsubpd ymm0,ymm0,YMMWORD PTR [rax+rcx*1]
<Laplace3D+0x3fd> vmovupd YMMWORD PTR [rdi+rcx*1],ymm0
<Laplace3D+0x402> add     rcx,0x20
<Laplace3D+0x406> cmp     r8d,r14d
<Laplace3D+0x409> jb      00000000000003bc <Laplace3D+0x3bc>
```

Sharing large read-only data/code

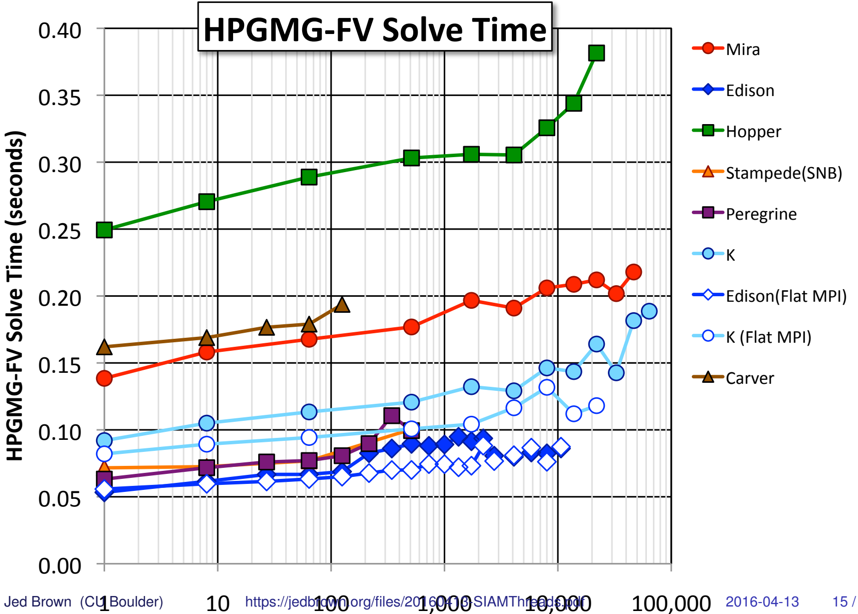
Memory hogs

- Templated/generated code
- Lookup tables
- Nonscalable replicated data structures

Solutions

- Threads: work around undesirable sharing
- Processes: allocate dynamically in a shared window
- Processes: compile into shared library: transparently shared

HPGMG-FV: flat MPI vs MPI+OpenMP (Aug 2014)



Outlook

- Application scaling mode must be scientifically relevant
- Threads and processes are more alike than usually acknowledged
- Processes versus threads is about shared versus private by default
    - No problem to share when desirable
    - Debuggability consequences
- Pointer indirection is handy; abstracts contiguity.
- Algorithmic barriers exist
    - Throughput architectures are not just “hard to program”
- Vectorization versus memory locality
- What is the cost of performance variability?
    - Measure best performance, average, median, 10th percentile?