Extreme Scaling on Petascale-class Systems

Cray XE/XT Porting, Scaling, and Optimization
VECTORIZATION:
LOOP ORDER MATTERS
Poor loop order results in poor striding

The inner-most loop strides on a slow dimension of each array.
The best the compiler can do is unroll.
Little to no cache reuse.

55. 1           ii = 0
56. 1 2----------< do b = abmin, abmax
57. 1 2 3--------<   do j=ijmin, ijmax
58. 1 2 3           ii = ii+1
59. 1 2 3           jj = 0
60. 1 2 3 4--------< do a = abmin, abmax
61. 1 2 3 4 r8------<   do i = ijmin, ijmax
62. 1 2 3 4 r8    jj = jj+1
63. 1 2 3 4 r8   f5d(a,b,i,j) =
                   +
                   tmat7(ii,jj)
64. 1 2 3 4 r8  f5d(b,a,i,j) =
                   -
                   tmat7(ii,jj)
65. 1 2 3 4 r8  f5d(a,b,j,i) =
                   -
                   tmat7(ii,jj)
66. 1 2 3 4 r8 f5d(b,a,j,i) =
                   +
                   tmat7(ii,jj)
67. 1 2 3 4 r8---->   end do
68. 1 2 3 4-------->   end do
69. 1 2 3-------->   end do
70. 1 2---------->   end do
Poor loop order results in poor cache reuse

<table>
<thead>
<tr>
<th></th>
<th>Time%</th>
<th>Time</th>
<th>Imb.Time</th>
<th>Imb.Time%</th>
<th>Calls</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>55.0%</td>
<td>13.938244 secs</td>
<td>0.075369 secs</td>
<td>0.6%</td>
<td>0.1 /sec</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**DATA_CACHE_REFILLS:**
- L2_MODIFIED:L2_OWNED: 11.858M/sec 165279602 fills
- L2_EXCLUSIVE:L2_SHARED

**DATA_CACHE_REFILLS_FROM_SYSTEM:**
- ALL 11.931M/sec 166291054 fills

**PAPI_L1_DCM**
- 23.499M/sec 327533338 misses

**PAPI_L1_DCA**
- 34.635M/sec 482751044 refs

**User time (approx)**
- 13.938 secs 36239439807 cycles 100.0% Time

**Average Time per Call**
- 13.938244 sec

**CrayPat Overhead : Time**
- 0.0%

**D1 cache hit,miss ratios**
- 32.2% hits 67.8% misses

**D2 cache hit,miss ratio**
- 49.8% hits 50.2% misses

**D1+D2 cache hit,miss ratio**
- 66.0% hits 34.0% misses
Loop Memory Access (Poor Stride)

\[ L = i + jI \]
Loop Memory Access (Good Stride)

\[ L = i + jI \]
Reordered loop nest

Now, the inner-most loop is stride-1 on both arrays.

Now memory accesses happen along the cache line, allowing reuse.

Compiler is able to vectorize and better-use SSE instructions.

75. 1 2--------------< do i = ijmin, ijmax
76. 1 2
jj = 0
77. 1 2 3--------------< do a = abmin, abmax
78. 1 2 3 4-------------< do j=ijmin, ijmax
79. 1 2 3 4
jj = jj+1
80. 1 2 3 4
ii = 0
81. 1 2 3 4 Vcr2--< do b = abmin, abmax
82. 1 2 3 4 Vcr2
ii = ii+1
83. 1 2 3 4 Vcr2 f5d(a,b,i,j) =
   f5d(a,b,i,j) +
   tmat7(ii,jj)
84. 1 2 3 4 Vcr2 f5d(b,a,i,j) =
   f5d(b,a,i,j) -
   tmat7(ii,jj)
85. 1 2 3 4 Vcr2 f5d(a,b,j,i) =
   f5d(a,b,j,i) -
   tmat7(ii,jj)
86. 1 2 3 4 Vcr2 f5d(b,a,j,i) =
   f5d(b,a,j,i) +
   tmat7(ii,jj)
87. 1 2 3 4 Vcr2--> end do
88. 1 2 3 4------------> end do
89. 1 2 3------------> end do
90. 1 2-----------> end do
### Improved striding greatly improved cache reuse

Runtime was cut nearly in half. Still, some 20% of all references are cache misses.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time%</td>
<td>31.4%</td>
</tr>
<tr>
<td>Time (secs)</td>
<td>7.955379</td>
</tr>
<tr>
<td>Imb.Time (secs)</td>
<td>0.260492</td>
</tr>
<tr>
<td>Imb.Time%</td>
<td>3.8%</td>
</tr>
<tr>
<td>Calls</td>
<td>0.1 /sec</td>
</tr>
<tr>
<td>calls</td>
<td>1.0</td>
</tr>
<tr>
<td>DATA CACHE REFILLS:</td>
<td></td>
</tr>
<tr>
<td>L2_MODIFIED:L2_OWNED:</td>
<td></td>
</tr>
<tr>
<td>L2_EXCLUSIVE:L2_SHARED fills</td>
<td>0.419M/sec 3331289</td>
</tr>
<tr>
<td>DATA_CACHE_REFILLS_FROM_SYSTEM:</td>
<td></td>
</tr>
<tr>
<td>ALL fills</td>
<td>15.285M/sec 121598284</td>
</tr>
<tr>
<td>PAPI_L1_DCM misses</td>
<td>13.330M/sec 106046801</td>
</tr>
<tr>
<td>PAPI_L1_DCA refs</td>
<td>66.226M/sec 526855581</td>
</tr>
<tr>
<td>User time (approx)</td>
<td></td>
</tr>
<tr>
<td>cycles</td>
<td>7.955 secs 20684020425</td>
</tr>
<tr>
<td>cycles 100.0%Time</td>
<td></td>
</tr>
<tr>
<td>Average Time per Call sec</td>
<td>7.955379</td>
</tr>
<tr>
<td>CrayPat Overhead : Time</td>
<td>0.0%</td>
</tr>
<tr>
<td>D1 cache hit,miss ratios</td>
<td>79.9% hits 20.1%</td>
</tr>
<tr>
<td>D2 cache hit,miss ratios</td>
<td>2.7% hits 97.3%</td>
</tr>
<tr>
<td>D1+D2 cache hit,miss ratios</td>
<td>80.4% hits 19.6%</td>
</tr>
</tbody>
</table>
Fissioned loops for better vectorization

First loop, partially vectorized and unrolled by 4

95.  1  
96.  1 2----------------< do j = ijmin, ijmax  
97.  1 2 i----------------< do b = abmin, abmax  
98.  1 2 i  
99.  1 2 i  
100. 1 2 i i--------< do i = ijmin, ijmax  
101. 1 2 i i Vpr4--< do a = abmin, abmax  
102. 1 2 i i Vpr4  
103. 1 2 i i Vpr4  
104. 1 2 i i Vpr4  
105. 1 2 i i Vpr4--> end do  
106. 1 2 i i--------> end do  
107. 1 2 i---------> end do  
108. 1 2-----------------> end do  

Second loop, vectorized and unrolled by 4

109.  1  
110.  1 2----------------< do i = ijmin, ijmax  
111.  1 2 3--------< do a = abmin, abmax  
112.  1 2 3  
113.  1 2 3  
114.  1 2 3 4--------< do j = ijmin, ijmax  
115.  1 2 3 4 Vr4--< do b = abmin, abmax  
116.  1 2 3 4 Vr4  
117.  1 2 3 4 Vr4  
118.  1 2 3 4 Vr4  
119.  1 2 3 4 Vr4--> end do  
120.  1 2 3 4--------> end do  
121.  1 2 3---------> end do  
122.  1 2-----------------> end do  

Fissioned loops for better vectorization
Fissioning further improved cache reuse and resulted in better vectorization

Runtime further reduced.
Cache hit/miss ratio improved slightly
Loopmark file points to better vectorization from the fissioned loops
VECTORIZATION:
MINIMIZE IFS IN THE INNER LOOP
Finite difference loop with several ifs inside the inner loop

(52) C      THE ORIGINAL
(53)
(54) DO 47020  J = 1, JMAX
(55) DO 47020  K = 1, KMAX
(56) DO 47020  I = 1, IMAX
(57)  JP = J + 1
(58)  JR = J - 1
(59)  KP = K + 1
(60)  KR = K - 1
(61)  IP = I + 1
(62)  IR = I - 1
(63) IF (J .EQ. 1)   GO TO 50
(64) IF (J .EQ. JMAX) GO TO 51
(65) XJ = ( A(I,JP,K) - A(I,JR,K) ) * DA2
(66) YJ = ( B(I,JP,K) - B(I,JR,K) ) * DA2
(67) ZJ = ( C(I,JP,K) - C(I,JR,K) ) * DA2
(68) CONTINUE
(69)  50 J1 = J + 1
(70)  J2 = J + 2
(71) XJ = (-3. * A(I,J,K) + 4. * A(I,J1,K) - A(I,J2,K) ) * DA2
(72) YJ = (-3. * B(I,J,K) + 4. * B(I,J1,K) - B(I,J2,K) ) * DA2
(73) ZJ = (-3. * C(I,J,K) + 4. * C(I,J1,K) - C(I,J2,K) ) * DA2
(74) CONTINUE
(75)  51 J1 = J - 1
(76)  J2 = J - 2
(77) XJ = ( 3. * A(I,J,K) - 4. * A(I,J1,K) + A(I,J2,K) ) * DA2
(79) ZJ = ( 3. * C(I,J,K) - 4. * C(I,J1,K) + C(I,J2,K) ) * DA2
(80) CONTINUE
(81) IF (K .EQ. 1)   GO TO 52
(82) IF (K .EQ. KMAX) GO TO 53
(83) XK = ( A(I,J,KP) - A(I,J,KR) ) * DB2
(84) YK = ( B(I,J,KP) - B(I,J,KR) ) * DB2
(85) ZK = ( C(I,J,KP) - C(I,J,KR) ) * DB2
(86) CONTINUE

continues...
PGI
55, **Invariant if transformation**
   Loop not vectorized: loop count too small
56, Invariant if transformation
Re-Write to move i loop inside the ifs

```
( 141) C THE RESTRUCTURED
( 142)  
( 143) DO 47029 J = 1, JMAX
( 144) DO 47029 K = 1, KMAX
( 145)  
( 146)       IF(J.EQ.1)THEN
( 147)  
( 148)         J1         = 2
( 149)         J2         = 3
( 150)       DO 47021 I = 1, IMAX
( 151)         VAJ(I) = (-3. * A(I,J,K) + 4. * A(I,J1,K) - A(I,J2,K) ) * DA2
( 152)         VBJ(I) = (-3. * B(I,J,K) + 4. * B(I,J1,K) - B(I,J2,K) ) * DA2
( 153)         VCJ(I) = (-3. * C(I,J,K) + 4. * C(I,J1,K) - C(I,J2,K) ) * DA2
( 154) 47021 CONTINUE
( 155)  
( 156) ELSE IF(J.NE.JMAX) THEN
( 157)  
( 158)         JP         = J+1
( 159)         JR         = J-1
( 160)       DO 47022 I = 1, IMAX
( 161)         VAJ(I) = ( A(I,JP,K) - A(I,JR,K) ) * DA2
( 162)         VBJ(I) = ( B(I,JP,K) - B(I,JR,K) ) * DA2
( 163)         VCJ(I) = ( C(I,JP,K) - C(I,JR,K) ) * DA2
( 164) 47022 CONTINUE
( 165)  
( 166) ELSE
( 167)  
( 168)         J1         = JMAX-1
( 169)         J2         = JMAX-2
( 170)       DO 47023 I = 1, IMAX
( 173)         VCJ(I) = ( 3. * C(I,J,K) - 4. * C(I,J1,K) + C(I,J2,K) ) * DA2
( 174) 47023 CONTINUE
( 175)  
( 176) ENDF
Continues...
```
Compiler messages after re-write

PGI
144, Invariant if transformation
   Loop not vectorized: loop count too small
150, Generated 3 alternate loops for the inner loop
   Generated vector sse code for inner loop
   Generated 8 prefetch instructions for this loop
   Generated vector sse code for inner loop
   Generated 8 prefetch instructions for this loop
   Generated vector sse code for inner loop
   Generated 8 prefetch instructions for this loop
   Generated vector sse code for inner loop
   Generated 8 prefetch instructions for this loop
160, Generated 4 alternate loops for the inner loop
   Generated vector sse code for inner loop
   Generated 6 prefetch instructions for this loop
   Generated vector sse code for inner loop
   Generated 6 prefetch instructions for this loop
   Generated vector sse code for inner loop
   Generated 6 prefetch instructions for this loop
   Generated vector sse code for inner loop
   Generated 6 prefetch instructions for this loop
AVX (Advanced Vector Extensions)

- Max Vector length doubled to 256 bit
- Much cleaner instruction set
  - Result register is unique from the source registers
  - Old SSE instruction set always destroyed a source register
- Floating point multiple-accumulate
  - \( A(1:4) = B(1:4) \times C(1:4) + D(1:4) \) ! Now one instruction
- Next gen of both AMD and Intel will have AVX

- Vectors are becoming more important, not less
CACHE BLOCKING
What is Cache Blocking?

• **Cache blocking** is a combination of strip mining and loop interchange, designed to increase data reuse.
  – Takes advantage of temporal reuse: re-reference array elements already referenced
  – Good blocking will take advantage of spatial reuse: work with the cache lines!

• Many ways to block any given loop nest
  – Which loops get blocked?
  – What block size(s) to use?

• Analysis can reveal which ways are beneficial
• But trial-and-error is probably faster
Cache Blocking: Quick Aside

• Before continuing, it’s important you know the following about cache blocking:
  – Most compilers already attempt to optimize for cache. They often do a very good job
  – Cache Pre-fetching can reduce the need for cache blocking on modern CPUs.
  – It’s not for the weak of heart...

• That said, the following is an introduction to cache blocking
Cache Use in Stencil Computations

• 2D Laplacian

\[
\begin{align*}
\text{do } j &= 1, 8 \\
\text{do } i &= 1, 16 \\
&\quad a = u(i-1,j) + u(i+1,j) \& \\
&\quad - 4u(i,j) \& \\
&\quad + u(i,j-1) + u(i,j+1) \\
\text{end do} \\
\text{end do}
\end{align*}
\]

• Cache structure for this example:
  – Each line holds 4 array elements
  – Cache can hold 12 lines of \( u \) data

• No cache reuse between outer loop iterations
Blocking to Increase Reuse

- Unblocked loop: 120 cache misses
- Block the inner loop

```fortran
do IBLOCK = 1, 16, 4
  do j = 1, 8
    do i = IBLOCK, IBLOCK + 3
      a(i,j) = u(i-1,j) + u(i+1,j) &
      - 2*u(i,j) &
      + u(i,j-1) + u(i,j+1)
    end do
  end do
end do
```

- Now we have reuse of the “j + 1” data
Blocking to Increase Reuse

- One-dimensional blocking reduced misses from 120 to 80
- Iterate over $4 \times 4$ blocks

```plaintext
    do JBLOCK = 1, 8, 4
        do IBLOCK = 1, 16, 4
            do j = JBLOCK, JBLOCK + 3
                do i = IBLOCK, IBLOCK + 3
                    a(i,j) = u(i-1,j) + u(i+1,j) &
                                          - 2*u(i,j)       &
                                          + u(i,j-1) + u(i,j+1)
                end do
            end do
        end do
    end do
```

- Better use of spatial locality (cache lines)
Obligatory GEMM discussion

- Matrix-matrix multiply (GEMM) is the canonical cache-blocking example because of the wealth of available locality.
- Operations can be arranged to create multiple levels of blocking
  - Block for registers
  - Block for cache (L1, L2, L3)
  - Block for TLB
  - ...
- Numerous, very good linear algebra libraries exist, so no further discussion here. Interested readers can see
  - Any book on code optimization
    - Sun’s *Techniques for Optimizing Applications: High Performance Computing* contains a decent introductory discussion in Chapter 8
    - Insert your favorite book here
    - Develops algorithms and cost models for GEMM in hierarchical memories
    - Description of GotoBLAS DGEMM
What Could Go Wrong?

“I tried cache-blocking my code, but it didn’t help”

• You’re doing it wrong.
  – Your block size is too small (too much loop overhead).
  – Your block size is too big (data is falling out of cache).
  – You’re targeting the wrong cache level (?)
  – You haven’t selected the correct subset of loops to block.

• The compiler is already blocking that loop.
• Prefetching is acting to minimize cache misses.
• Computational intensity within the loop nest is very large, making blocking less important.
A Real-Life Example: NPB MG

- Multigrid PDE solver
- Class D, 64 MPI ranks
  - Global grid is $1024 \times 1024 \times 1024$
  - Local grid is $258 \times 258 \times 258$
- Two similar loop nests account for >50% of run time
- 27-point 3D stencil
  - There is good data reuse along leading dimension, even without blocking

```plaintext
do i3 = 2, 257
  do i2 = 2, 257
    do i1 = 2, 257
      update u(i1,i2,i3)
    end do
  end do
end do
```

![Cache lines diagram]
I’m Doing It Wrong

- Block the inner two loops
- Creates blocks extending along \textbf{i3} direction

\begin{verbatim}
  do I2BLOCK = 2, 257, BS2
    do I1BLOCK = 2, 257, BS1
      do i3 = 2, 257
        do i2 = I2BLOCK, min(I2BLOCK+BS2-1, 257)
          do i1 = I1BLOCK, min(I1BLOCK+BS1-1, 257)
            update u(i1,i2,i3)
            using 27-point stencil
          end do
        end do
      end do
    end do
  end do
\end{verbatim}

\begin{table}
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Block size} & \textbf{Mop/s/process} \\
\hline
unblocked & 531.50 \\
16 × 16 & 279.89 \\
22 × 22 & 321.26 \\
28 × 28 & 358.96 \\
34 × 34 & 385.33 \\
40 × 40 & 408.53 \\
46 × 46 & 443.94 \\
52 × 52 & 468.58 \\
58 × 58 & 470.32 \\
64 × 64 & 512.03 \\
70 × 70 & 506.92 \\
\hline
\end{tabular}
\end{table}
That’s Better

- Block the outer two loops
- Preserves spatial locality along \( i_1 \) direction

```plaintext
do I3BLOCK = 2, 257, BS3
  do I2BLOCK = 2, 257, BS2
    do i3 = I3BLOCK, min(I3BLOCK+BS3-1, 257)
      do i2 = I2BLOCK, min(I2BLOCK+BS2-1, 257)
        do i1 = 2, 257
          ! update \( u(i_1,i_2,i_3) \)
          ! using 27-point stencil
          end do
        end do
      end do
    end do
  end do
end do
```

<table>
<thead>
<tr>
<th>Block size</th>
<th>Mop/s/process</th>
</tr>
</thead>
<tbody>
<tr>
<td>unblocked</td>
<td>531.50</td>
</tr>
<tr>
<td>16 × 16</td>
<td>674.76</td>
</tr>
<tr>
<td>22 × 22</td>
<td>680.16</td>
</tr>
<tr>
<td>28 × 28</td>
<td>688.64</td>
</tr>
<tr>
<td>34 × 34</td>
<td>683.84</td>
</tr>
<tr>
<td>40 × 40</td>
<td>698.47</td>
</tr>
<tr>
<td>46 × 46</td>
<td>689.14</td>
</tr>
<tr>
<td>52 × 52</td>
<td>706.62</td>
</tr>
<tr>
<td>58 × 58</td>
<td>692.57</td>
</tr>
<tr>
<td>64 × 64</td>
<td>703.40</td>
</tr>
<tr>
<td>70 × 70</td>
<td>693.87</td>
</tr>
</tbody>
</table>
MAKE C POINTERS “SAFE”
C pointers

C pointers don’t carry the same rules as Fortran Arrays.
The compiler has no way to know whether *a, *b, and *c overlap or are referenced differently elsewhere.
The compiler must assume the worst, thus a false data dependency.
C pointers, restricted

C99 introduces the `restrict` keyword, which allows the programmer to promise not to reference the memory via another pointer.

If you declare a restricted pointer and break the rules, behavior is **undefined** by the standard.

```c
void mat_mul_daxpy(double* restrict a, double* restrict b, double* restrict c, int rowa, int cola, int colb)
{
    int i, j, k;          /* loop counters */
    int rowc, colc, rowb; /* sizes not passed as arguments */
    double con;           /* constant value */
    rowb = cola;
    rowc = rowa;
    colc = colb;
    for(i=0;i<rowc;i++) {
        for(k=0;k<cola;k++) {
            con = *(a + i*cola +k);
            for(j=0;j<colc;j++) {
                *(c + i*colc + j) += con * *(b + k*colb + j);
            }
        }
    }
}
```
C pointers, rewrite

66, Generated alternate loop with no peeling – executed if loop count ≤ 24
   Generated vector sse code for inner loop
   Generated 2 prefetch instructions for this loop
   Generated vector sse code for inner loop
   Generated 2 prefetch instructions for this loop
   Generated alternate loop with no peeling and more aligned moves – executed if loop count ≤ 24 and alignment test is passed
   Generated vector sse code for inner loop
   Generated 2 prefetch instructions for this loop
   Generated alternate loop with more aligned moves – executed if loop count ≥ 25 and alignment test is passed
   Generated vector sse code for inner loop
   Generated 2 prefetch instructions for this loop

• This can also be achieved with the PGI safe pragma or –Msafeptr compiler option or Pathscale –OPT:alias option
TUNING MALLOC
GNU Malloc

• GNU malloc library
  – malloc, calloc, realloc, free calls
    • Fortran dynamic variables

• Malloc library system calls
  – Mmap, munmap => for larger allocations
  – Brk, sbrk => increase/decrease heap

• Malloc library optimized for low system memory use
  – Can result in system calls/minor page faults
Detecting “bad” malloc behavior
- Profile data => “excessive system time”

Correcting “bad” malloc behavior
- Eliminate mmap use by malloc
- Increase threshold to release heap memory

Use environment variables to alter malloc
- MALLOC_MMAP_MAX_ = 0
- MALLOC_TRIM_THRESHOLD_ = 536870912

Possible downsides
- Heap fragmentation
- User process may call mmap directly
- User process may launch other processes

PGI’s – Msmartalloc does something similar for you at compile time
Google TCMalloc

• Google created a replacement “malloc” library
  – “Minimal” TCMalloc replaces GNU malloc
• Limited testing indicates TCMalloc as good or better than GNU malloc
  – Environment variables not required
  – TCMalloc almost certainly better for allocations in OpenMP parallel regions
• There’s currently no pre-built tcmalloc for Cray XT/XE, but some users have successfully built it.
Memory Allocation: Make it local

• Linux has a “first touch policy” for memory allocation
  – *alloc functions don’t actually allocate your memory
  – Memory gets allocated when “touched”

• Problem: A code can allocate more memory than available
  – Linux assumed “swap space,” we don’t have any
  – Applications won’t fail from over-allocation until the memory is finally touched

• Problem: Memory will be put on the core of the “touching” thread
  – Only a problem if thread 0 allocates all memory for a node

• Solution: Always initialize your memory immediately after allocating it
  – If you over-allocate, it will fail immediately, rather than a strange place in your code
  – If every thread touches its own memory, it will be allocated on the proper socket
This may help both compute and communication.

TRY HUGE PAGES
Why use Huge Pages

• Opterons support 4K, 2M, and 1G pages
  – Cray doesn’t support 1G pages
  – 4K pages are used by default
• 2M pages are more difficult to use, but...

• Your code may run with fewer TLB misses (hence faster).
  – The TLB can address more physical memory with 2M pages than with 4K pages
• The Gemini perform better with 2M pages than with 4K pages.
  – 2M pages use less GEMINI resources than 4k pages (fewer bytes).
Huge Pages – How to use

• Load craype-hugepages2M module
  – Other sizes also available

• Use the aprun option –m###h to ask for ### Meg of HUGE pages.
  – Example: aprun –m500h (Request 500 Megs of HUGE pages as available, use 4K pages thereafter)
  – Example: aprun –m500hs (Request 500 Megs of HUGE pages, if not available terminate launch)

• Note: If not enough huge pages are available, the cost of filling the remaining with 4K pages may degrade performance compared to all huge pages.
USE MPI RANK REORDERING
Rank Placement

• The default ordering can be changed using the following environment variable:

  MPICH_RANK_REORDER_METHOD

• These are the different values that you can set it to:

  0: Round-robin placement – Sequential ranks are placed on the next node in the list. Placement starts over with the first node upon reaching the end of the list.

  1: SMP-style placement – Sequential ranks fill up each node before moving to the next.

  2: Folded rank placement – Similar to round-robin placement except that each pass over the node list is in the opposite direction of the previous pass.

  3: Custom ordering. The ordering is specified in a file named MPICH_RANK_ORDER.

• When is this useful?

  – Point-to-point communication consumes a significant fraction of program time and a load imbalance detected
  – Also shown to help for collectives (alltoall) on subcommunicators (GYRO)
  – Spread out IO across nodes (POP)
Rank Order and CrayPAT

• One can also use the CrayPat performance measurement tools to generate a suggested custom ordering.
  – Available if MPI functions traced (-g mpi or –O apa)
  – pat_build –O apa my_program
    • see Examples section of pat_build man page
• pat_report options:
  – mpi_sm_rank_order
    • Uses message data from tracing MPI to generate suggested MPI rank order. Requires the program to be instrumented using the pat_build -g mpi option.
  – mpi_rank_order
    • Uses time in user functions, or alternatively, any other metric specified by using the -s mro_metric options, to generate suggested MPI rank order.
Reordering with CrayPAT Workflow

- module load xt-crtpat
- Rebuild your code
- pat_build –O apa a.out
- Run a.out+pat
- pat_report –Ompi_sm_rank_order a.out+pat+...sdt/ > pat.report
- Creates MPICH_RANK_REORDER_METHOD.x file
- Then set env var MPICH_RANK_REORDER_METHOD=3 AND
- Link the file MPICH_RANK_ORDER.x to MPICH_RANK_ORDER
- Rerun code
Weak Scaling Study of S3D

Number of cores vs. Micoseconds per grid point
### Profile at 6000 processors

<table>
<thead>
<tr>
<th>Time %</th>
<th>Time</th>
<th>Imb. Time</th>
<th>Imb. Time</th>
<th>Calls</th>
<th>Experiment=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>1530.892958</td>
<td>--</td>
<td>--</td>
<td>27414118.0</td>
<td>Total</td>
</tr>
</tbody>
</table>

| 52.0% | 796.046937 | -- | -- | 22403802.0 | USER |

| 22.3% | 341.176468 | 3.482338 | 1.0% | 19200000.0 | getrates_ |
| 17.4% | 266.542501 | 35.451437 | 11.7% | 1200.0 | rhsf_ |
| 5.1% | 78.772615 | 0.532703 | 0.7% | 3200000.0 | mcvais_new_looptool_ |
| 2.6% | 40.477488 | 2.889609 | 6.7% | 1200.0 | diffflux_proc_looptool_ |
| 2.1% | 31.666938 | 6.785575 | 17.6% | 200.0 | integrate_erk_jstage_lt_ |
| 1.4% | 21.318895 | 5.042270 | 19.1% | 1200.0 | computeheatflux_looptool_ |
| 1.1% | 16.091956 | 6.863891 | 29.9% | 1.0 | main |

| 47.4% | 725.049709 | -- | -- | 5006632.0 | MPI |

| 43.8% | 670.742304 | 83.143600 | 11.0% | 2389440.0 | mpi_wait_ |
| 1.9% | 28.821882 | 281.694997 | 90.7% | 1284320.0 | mpi_isend_ |
Profile at 48,000 processors

<table>
<thead>
<tr>
<th>Time %</th>
<th>Time</th>
<th>Imb. Time</th>
<th>Imb. Time</th>
<th>Calls</th>
<th>Experiment=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>1730.555208</td>
<td>--</td>
<td>--</td>
<td>16090113.8</td>
<td>Total</td>
</tr>
<tr>
<td>76.9%</td>
<td>1330.111350</td>
<td>--</td>
<td>--</td>
<td>4882627.8</td>
<td>MPI</td>
</tr>
<tr>
<td>72.1%</td>
<td>1247.436960</td>
<td>54.277263</td>
<td>4.2%</td>
<td>2389440.0</td>
<td>mpi_wait_</td>
</tr>
<tr>
<td>1.3%</td>
<td>22.712017</td>
<td>101.212360</td>
<td>81.7%</td>
<td>1234718.3</td>
<td>mpi_isend_</td>
</tr>
<tr>
<td>1.0%</td>
<td>17.623757</td>
<td>4.642004</td>
<td>20.9%</td>
<td>1.0</td>
<td>mpi_comm_dup_</td>
</tr>
<tr>
<td>1.0%</td>
<td>16.849281</td>
<td>71.805979</td>
<td>81.0%</td>
<td>1234718.3</td>
<td>mpi_irecv_</td>
</tr>
<tr>
<td>1.0%</td>
<td>16.835691</td>
<td>192.820387</td>
<td>92.0%</td>
<td>19999.2</td>
<td>mpi_waitall_</td>
</tr>
<tr>
<td>22.2%</td>
<td>384.978417</td>
<td>--</td>
<td>--</td>
<td>11203802.0</td>
<td>USER</td>
</tr>
<tr>
<td>9.9%</td>
<td>171.440025</td>
<td>1.929439</td>
<td>1.1%</td>
<td>9600000.0</td>
<td>getrates_</td>
</tr>
<tr>
<td>7.7%</td>
<td>133.599580</td>
<td>19.572807</td>
<td>12.8%</td>
<td>1200.0</td>
<td>rhsf_</td>
</tr>
<tr>
<td>2.3%</td>
<td>39.465572</td>
<td>0.600168</td>
<td>1.5%</td>
<td>1600000.0</td>
<td>mcavis_new_looptool_</td>
</tr>
</tbody>
</table>
S3D performs differencing in 3 directions

- Differencing in the X direction
  - MPI Task K +1
  - MPI Task K
  - MPI Task K - 1

- Differencing in the Y direction
  - MPI Task K +30
  - MPI Task K
  - MPI Task K -30

- Differencing in the Z direction
  - MPI Task K +1200
  - MPI Task K
  - MPI Task K-1200
Default mapping of the MPI tasks on a multi-core node

Code must perform one communication across each surface of a cube
12 cubes perform 72 communications, 63 of which go “off node”

Optimized mapping of the MPI tasks on the node
Still performs 72 communications, but now only 32 are off node
Application data is in a 3D space, $X \times Y \times Z$.

Communication is nearest-neighbor.

Default ordering results in $12 \times 1 \times 1$ block on each node.

A custom reordering is now generated: $3 \times 2 \times 2$ blocks per node, resulting in more on-node communication.
Suggested MPI rank order

% pat_report -O mpi_sm_rank_order -s rank_grid_dim=8,6 ...

Notes for table 1:

To maximize the locality of point to point communication, specify a Rank Order with small Max and Avg Sent Msg Total Bytes per node for the target number of cores per node.

To specify a Rank Order with a numerical value, set the environment variable MPICH_RANK_REORDER_METHOD to the given value.

To specify a Rank Order with a letter value 'x', set the environment variable MPICH_RANK_REORDER_METHOD to 3, and copy or link the file MPICH_RANK_ORDER.x to MPICH_RANK_ORDER.

Table 1: Sent Message Stats and Suggested MPI Rank Order

<table>
<thead>
<tr>
<th>Number Partners</th>
<th>Rank Count</th>
<th>Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>0 5 42 47</td>
</tr>
</tbody>
</table>
| 3               | 20         | 1 2 3 4 ...
| 4               | 24         | 7 8 9 10 ... |
Suggested MPI rank order ctn'd

Four cores per node: Sent Msg Total Bytes per node

<table>
<thead>
<tr>
<th>Rank Order</th>
<th>Max Total Bytes</th>
<th>Max/ SMP</th>
<th>Max Total Bytes</th>
<th>Avg SMP</th>
<th>Avg Node Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>121651200</td>
<td>73.9%</td>
<td>864000000</td>
<td>62.5%</td>
<td>14,20,15,21</td>
</tr>
<tr>
<td>h</td>
<td>121651200</td>
<td>73.9%</td>
<td>864000000</td>
<td>62.5%</td>
<td>14,20,21,15</td>
</tr>
<tr>
<td>u</td>
<td>152064000</td>
<td>92.4%</td>
<td>146534400</td>
<td>106.0%</td>
<td>13,12,10,4</td>
</tr>
<tr>
<td>1</td>
<td>164505600</td>
<td>100.0%</td>
<td>138240000</td>
<td>100.0%</td>
<td>16,17,18,19</td>
</tr>
<tr>
<td>d</td>
<td>164505600</td>
<td>100.0%</td>
<td>142387200</td>
<td>103.0%</td>
<td>16,17,19,18</td>
</tr>
<tr>
<td>0</td>
<td>224640000</td>
<td>136.6%</td>
<td>207360000</td>
<td>150.0%</td>
<td>1,13,25,37</td>
</tr>
<tr>
<td>2</td>
<td>241920000</td>
<td>147.1%</td>
<td>207360000</td>
<td>150.0%</td>
<td>7,16,31,40</td>
</tr>
</tbody>
</table>
MPI re-ordering Case Study 2

• Application uses x, y, z structured grid
• MPI uses two communicators
  – Shared memory on node – fast
  – Interconnect between nodes – not as fast
• Application grid => 3-D grid of blocks
  – Each block mapped to a processor
  – Map blocks to node to minimize off-node communication
• Use MPI rank re-ordering to map blocks to nodes
Using grid_order

• If MPICH_RANK_REORDER_METHOD=3
  – then rank order => MPICH_RANK_ORDER file

• Use grid_order to generate MPICH_RANK_ORDER
  – Part of Cray’s perftools module
  – “module load perftools” to access command/man-page
  – grid_order –C –g x,y,z –c nx, ny, nz
    • -C: row major ordering
    • -g: x, y, z grid size
      – x*y*z = number of MPI processes
    • -c: nx, ny, nz of the grids on node
      – nx*ny*nz = number of MPI processes on a node
  – MPI re-order file written to stdout
MPI rank order for grid topology - grid_order

% grid_order -c 2,2 -g 8,6

# grid_order -c 2,2 -g 8,6
# Region 0: 0,0 (0..47)
0,1,6,7
2,3,8,9
4,5,10,11
12,13,18,19
14,15,20,21
16,17,22,23
24,25,30,31
26,27,32,33
28,29,34,35
36,37,42,43
38,39,44,45
40,41,46,47

This script will also handle the case that cells do not evenly partition the grid.
Application Example – Part 1

• Three test cases:
  – 256 processors: 16x2x8 grid
  – 512 processors: 16x4x8 grid
  – 1024 processors: 16x4x16 grid

• For 256 processors: 16x2x8 grid
  – IL-16 node has 32 cores
    • Possible grid block groups (nx, ny, nz) for a node:
      • 16x2x1: 64 neighbors off-node
      • 2x2x8: 32 neighbors off-node
      • 4x2x4: 24 neighbors off-node
Application Example – Part 2

• For 256 processors test case
  – Using 2x2x8 blocks/node was fastest
    • Default: 0.097 sec/compute iter
    • 2x2x8 blocks/node: 0.085 sec/compute iter
  – 12% faster than the default results!

• Final additions to the 256pe PBS batch script:
  – . ${MODULESHOME}/init/sh
  – module load perftools
  – export MPICH_RANK_REORDER_METHOD=3
  – /bin/rm --rf MPICH_RANK_ORDER
  – grid_order --C --g 16,2,8 --c 2,2,8 > MPICH_RANK_ORDER
ADVANCED OPENMP FEATURES
OpenMP 3.0: OMP TASK

• An OpenMP task is an explicit region of code whose execution can be deferred and/or executed in parallel with the surrounding code
  – Completion is guaranteed by synchronization or end of parallel region
  – Must be contained inside a OMP parallel region
  – A task is “put on a queue” to be executed “later”
  – Any thread of the same parallel region that is sitting on a sync point can grab a task off the queue and execute it

• Sort of like “futures” but with limitations
  – Don’t have ID’s, must wait for all or none
  – But maybe are good enough?
Multi-level OpenMP

- Nested OpenMP
  - OMP parallel region inside of an OMP parallel region
  - “New threads” are used at each level
    - OMP 3.1 updated OMP_NUM_THREADS to better support
      nested parallelism (OMP_NUM_THREADS=4, 3)

- OMP Tasks inside of parallel regions
  - Can be nested
  - Can be both more and less natural way of
    programming
Multi-level OpenMP

!$omp parallel do ...
do i=1,4
  call complex_matmul(...)  
enddo

Subroutine complex_matmul(...)
!$omp parallel do private(j,jend,jsize)
do j=1,n,nb
  jend = min(n, j+nb-1)
  jsize = jend - j + 1
  call zgemm( transA,transB, m,jsize,k, &
              alpha,A,ldA,B(j,1),ldb, beta,C(1,j),ldC)
endo
Case Study: PARQUET

4 x ZGEMM 1000x1000

Parallel method and Nthreads at each level: Serial ZGEMM, High Level OMP ZGEMM 4x1, Nested OMP ZGEMM 3x3, Nested OMP ZGEMM 4x2, Nested OMP ZGEMM 2x4, Low level OMP ZGEMM 1x8.
Case Study: PARQUET

4 x ZGEMM 100x100

Gflops

Serial ZGEMM
High Level OMP
Nested OMP
Nested OMP
Low Level

Parallel method and Nthreads at each level
Lessons from nested parallel regions

• Nested omp can GREATLY expand the amount of parallelism one can attack using OpenMP
• OpenMP 3.1 has expanded the meaning of OMP_NUM_THREADS to handle nested parallelism
• Nested parallel regions is a relatively static distribution

• OMP tasking may be a way of getting around some or all of these issues