Comparing Optimizations of GTC for the Cray X1E and XT3

James B. White III* Nathan Wichmann† Stephane Ethier‡

Abstract

We describe a variety of optimizations of GTC, a 3D gyrokinetic particle-in-cell code, for the Cray X1E. Using performance data from multiple points in the optimization process, we compare the relative importance of different optimizations on the Cray X1E and their effects on Cray XT3 performance. We also describe the performance difference between small pages and large pages on the XT3, a difference that is particularly significant for GTC.

1 Introduction

We describe a variety of optimizations of GTC, a 3D gyrokinetic particle-in-cell code, for the Cray X1E. Using performance data from multiple points in the optimization process, we compare the relative importance of different optimizations on the Cray X1E and their effects on Cray XT3 performance.

GTC is a prominent application within the Department of Energy (DOE) Office of Science used to simulate turbulent transport in fusion plasmas. It uses a 3D gyrokinetic particle-in-cell algorithm that is intrinsically global and has scaled to more than ten-thousand processors. Versions of GTC have been vectorized for the Cray X1 and NEC SX series, including the Earth Simulator, and their performance was presented at past CUG meeting and SC conferences.[1][2] We started with a current version of GTC without vector modifications to specifically target the X1E and record the full performance history of optimization process.

The X1E and XT3 we used are part of the DOE Leadership Computing Facility at the National Center for Computational Sciences (NCCS).[3] The X1E, named "Phoenix", has 1,024 multi-streaming processors (MSPs), with 2 GB of memory per MSP. All executables were generated with version 5.4 of the Programming Environment. The XT3, named "Jaguar", has 5,212 processors in the compute partition, where each processor is a 2.4 GHz AMD Opteron with 2 GB of memory. The initial XT3 run used an executable generated with version 6.0 of the Portland Group (PGI) Fortran compiler, while later runs used version 6.1.3.

In the following section, we describe the optimization process used for the X1E, including our changes to the GTC source code and the resulting performance measurements. We then describe a few performance experiments performed on the XT3 using versions of GTC created as part of X1E optimization. We end with a summary of our performance measurements and comparison of GTC performance on other prominent high-performance computers.

2 GTC on X1E

We tuned GTC on the X1E using a 64-process benchmark that is representative of current science runs. We used an iterative process that started with a benchmark run using an executable instrumented for performance measurement using the Cray "`pat_build`" tool. After each run we generated a performance profile with statistics listed by procedure and by line, using a command of the following form.

*Oak Ridge National Laboratory
†Cray, Inc.
‡Princeton Plasma Physics Laboratory

```
pat_report -b functions,lines ...
```

We then committed the source code, benchmark output, and performance results to a Subversion[4] repository.

Using the profile from "`pat_report`" and loopmark listings from the Cray Fortran compiler, we identified expensive code blocks and potential optimizations. After implementing a potential optimization, we regenerated loopmarks to determine if the change improved vectorization and/or multistreaming. We then repeated the process starting with a run of the benchmark using the modified source code.

Figure 1 shows the performance results for each revision of GTC for which the benchmark was run. Each "commit" of software modifications to the repository generated a new revision number, and we did not run the benchmark for every revision. The performance metric is the runtime as reported by internal timers for the main time-stepping loop. The column labeled "X1E r2" shows the runtime for the first revision measured for performance on the X1E.

2.1 Revisions 2 to 3

The profile for revision 2 begins as follows.

```
      Samp% | Cum.Samp% |     Samp |Function
                                   |Line

     100.0% |    100.0% | 22049624 |Total
    |-----------------------------------------
    |  61.5% |     61.5% | 13557450 |chargei_
    ||----------------------------------------
    ||  12.2% |     12.2% |  2698466 |line.140
    ||   9.2% |     21.5% |  2031530 |line.132
    ||   6.4% |     27.8% |  1404587 |line.102
    ||   4.8% |     32.6% |  1047693 |line.133
```

The procedure "`chargei`" dominates the runtime. The following excerpt from the "`chargei`" loopmark listing indicates the performance problem.

```
   90.  1-------------<   do m=1,mi
    ..
   93.  1                    kk=kzion(m)
    ..
   97.  1 Vs----------<      do larmor=1,4
    ..
  127.  1 Vs                   ij=jtion0(larmor,m)
  128.  1 Vs                   densityi(kk,ij) = densityi(kk,ij) + wz0*wt00
  129.  1 Vs                   densityi(kk+1,ij)   = densityi(kk+1,ij)   + wz1*wt00
  130.  1 Vs
  131.  1 Vs                   ij=ij+1
  132.  1 Vs                   densityi(kk,ij) = densityi(kk,ij) + wz0*wt10
  133.  1 Vs                   densityi(kk+1,ij)   = densityi(kk+1,ij)   + wz1*wt10
    ..
  144.  1 Vs---------->      enddo
  145.  1------------->   enddo

ftn-6289 ftn: VECTOR File = chargei.F90, Line = 90
  A loop starting at line 90 was not vectorized because a recurrence was found
  on "DENSITYI" between lines 128 and 129.
```

The index variables "kk" and "ij" are not permutations, so the updates to the variable "densityi" are not independent, and the loop over "m" cannot be vectorized or multistreamed. The fix is in the following excerpt from revision 3.

```
 ...
 15.                  #ifdef _UNICOSMP
 16.                    integer, parameter :: vlen = 256
 17.                    integer :: mv, v
 18.                    real(wp) dnipart(mgrid,0:mzeta,vlen)
 19.                  #endif
 ...
 98.                  #ifdef _UNICOSMP
 99.  m-------------<   do mv=1,mi,vlen
100.  m                 !dir$ prefervector
101.  m MVs---------<   do m=mv,min(mv+vlen-1,mi)
102.  m MVs                v=m-mv+1
103.  m MVs           #else
104.  m MVs             do m=1,mi
105.  m MVs           #endif
 ...
140.  m MVs 3               ij=jtion0(larmor,m)
141.  m MVs 3               dnipart(ij,kk,v) = dnipart(ij,kk,v) + wz0*wt00
142.  m MVs 3               dnipart(ij,kk+1,v)   = dnipart(ij,kk+1,v)   + wz1*wt00
 ...
194.  ir------------<   do v=1,vlen
195.  ir 2----------<      do kk=0,mzeta
196.  ir 2                   !dir$ preferstream
197.  ir 2 MV-------<         do ij=1,mgrid
198.  ir 2 MV                   densityi(kk,ij) = densityi(kk,ij) + dnipart(ij,kk,v)
199.  ir 2 MV------->         enddo
200.  ir 2---------->      enddo
201.  ir------------>   enddo
```

Preprocessor statements cause the revised code to only appear to the compiler of the Cray X1 series, for the macro "_UNICOSMP" is automatically defined for this series.

The work array "dnipart" has an extra dimension compared to "densityi", and the size of that dimension is "vlen=256", where 256 is the vector length of the X1E when the entire MSP is used for a single loop. An additional loop iterates over blocks of size "vlen", and the "m" loop iterates over the elements of a block. The added dimension makes the updates independent within the innermost loop, so they are vectorized and multistreamed ("MV"). The "s" in "MVs" indicates the loop was short enough so a single series of vector instructions performs all iterations of the loop.

A new loop nest appears after the original loops to accumulate the vectorized results. This loop is extra work, but the work is worth it to enable vectorization. The value of "mgrid" is large, over 32,000, while the value of "mzeta" is small for moderate to large numbers of parallel processes. For 64 or more processes, it is only 1. Therefore we vectorize and multistream over "mgrid".

Column "X1E r3" of Figure 1 shows the resulting performance for revision 3, a 3.2× improvement over revision 2.

2.2 Revisions 3 to 5

The profile for revision 3 begins as follows.

```
Samp% | Cum.Samp% |    Samp |Function
```

```
                                  |Line

 100.0% |    100.0% | 8768310 |Total
|---------------------------------------
|  30.3% |     30.3% | 2660905 |shifti_
||--------------------------------------
||  12.3% |     12.3% | 1080482 |line.154
||   7.3% |     19.7% |  644222 |line.206
||   6.1% |     25.8% |  535231 |line.128
||   2.1% |     27.9% |  186416 |line.207
||   1.4% |     29.3% |  120175 |line.132
||   0.2% |     29.5% |   17864 |line.190
...
|   7.5% |     93.1% |  660755 |chargei_
```

The procedure "`chargei`" is much faster, and the procedure "`shifti`" now dominates. The following excerpt from "`shifti`" shows the performance problem.

```
  128.  r-------------<      do m=m0,mi
  129.  r                      zetaright=min(2.0*pi,zion(3,m))-zetamax
  130.  r                      zetaleft=zion(3,m)-zetamin
  131.  r
  132.  r                      if( zetaright*zetaleft > 0 )then
  133.  r                         zetaright=zetaright*0.5*pi_inv
  134.  r                         zetaright=zetaright-real(floor(zetaright))
  135.  r                         msend=msend+1
  136.  r                         kzi(msend)=m
  137.  r
  138.  r                         if( zetaright < 0.5 )then
  139.  r               ! # of particle to move right
  140.  r                            msendright(1)=msendright(1)+1
  141.  r                            iright(msendright(1))=m
  142.  r               ! keep track of tracer
  143.  r                            if( nhybrid == 0 .and. m == ntracer )then
  144.  r                               msendright(2)=msendright(1)
  145.  r                               ntracer=0
  146.  r                            endif
  147.  r
  148.  r               ! # of particle to move left
  149.  r                         else
  150.  r                            msendleft(1)=msendleft(1)+1
  151.  r                            ileft(msendleft(1))=m
  152.  r                            if( nhybrid == 0 .and. m == ntracer )then
  153.  r                               msendleft(2)=msendleft(1)
  154.  r                               ntracer=0
  155.  r                            endif
  156.  r                         endif
  157.  r                      endif
  158.  r-------------->      enddo
  ...
ftn-6005 ftn: SCALAR File = shifti.F90, Line = 128
  A loop starting at line 128 was unrolled 2 times.
```

```
ftn-6254 ftn: VECTOR File = shifti.F90, Line = 128
  A loop starting at line 128 was not vectorized because a recurrence was found
  on "NTRACER" at line 145.
```

The loop starting at line 128 iterates over particles that might move to another processor. The "`if`" statement on line 132 tests whether or not the particle does indeed move to another processor. The block on lines 139-146 is for particles moving "right" around the ring of the fusion device, and the block on lines 148-156 is for particles moving "left".

A critical computation inhibiting vectorization is on lines 142-146, a test of whether the simulation's tracer particle is on this processor and is about to move to another processor. This computation has a dependence on line 145, where the value of "`ntracer`" is reset.

The above code block from revision 3 was replaced by a completely different block in revision 5.

```
 123.                  #elif defined _UNICOSMP
 124.
 125.  V-------------<      do m=m0,mi
 126.  V                      zetaright=min(2.0*pi,zion(3,m))-zetamax
 127.  V                      zetaleft=zion(3,m)-zetamin
 128.  V                      if (zetaright*zetaleft > 0) then
 129.  V                         msend=msend+1
 130.  V                         kzi(msend)=m
 131.  V                      endif
 132.  V------------->      enddo
 133.
 134.  V-------------<      do i=1,msend
 135.  V                      m=kzi(i)
 136.  V                      zetaright=min(2.0*pi,zion(3,m))-zetamax
 137.  V                      zetaright=zetaright*0.5*pi_inv
 138.  V                      zetaright=zetaright-real(floor(zetaright))
 139.  V                      if (zetaright < 0.5) then
 140.  V               ! # of particle to move right
 141.  V                         msendright(1)=msendright(1)+1
 142.  V                         iright(msendright(1))=m
 143.  V               ! # of particle to move left
 144.  V                      else
 145.  V                         msendleft(1)=msendleft(1)+1
 146.  V                         ileft(msendleft(1))=m
 147.  V                      endif
 148.  V------------->      enddo
 149.
 150.                  ! keep track of tracer
 151.                       if ((nhybrid == 0) .and. (ntracer > 0)) then
 152.  MV------------<         do i=1,msendright(1)
 153.  MV                         if (iright(i) == ntracer) msendright(2)=i
 154.  MV------------>         enddo
 155.                         if (msendright(2) /= 0) then
 156.                            ntracer=0
 157.                         else
 158.  MV------------<            do i=1,msendleft(1)
 159.  MV                            if (ileft(i) == ntracer) msendleft(2)=i
 160.  MV------------>            enddo
 161.                            if (msendleft(2) /= 0) ntracer=0
 162.                         endif
```

```
163.                        endif
```

The modified code is again protected by a preprocessor macro so that it is only used on the X1 series.

Lines 125-132 first find the particles that will move. The computation is similar to the Fortran "`pack`" intrinsic, a pattern the compiler recognizes, and it vectorizes but does not multistream. Lines 134-148 test whether the particles move left or right using a further generalization of "`pack`" that vectorizes but doesn't multistream. Lines 150-163 check for the tracer particle using reduction operations, "`logical`" analogies of the "`sum`" intrinsic, which vectorize and multistream.

Column "X1E r5" of Figure 1 shows the resulting performance for revision 5, a 1.4× improvement over revision 3.

2.3 Revisions 5 to 6

The profile for revision 5 begins as follows.

```
 Samp% | Cum.Samp% |    Samp |Function
                             |Line

 100.0% |    100.0% | 7126472 |Total
|-----------------------------------
|  34.0% |     34.0% | 2422227 |pushi_
||----------------------------------
||   9.6% |      9.6% |  683490 |line.93
||   4.6% |     14.2% |  330526 |line.80
||   4.3% |     18.6% |  309770 |line.92
||   3.8% |     22.4% |  269578 |line.94
||   1.9% |     24.2% |  131984 |line.62
||   1.5% |     25.7% |  107877 |line.148
||   0.9% |     26.6% |   62389 |line.105
...
|  13.7% |     81.3% |  973509 |shifti_
...
|   9.2% |     90.5% |  654013 |chargei_
```

The "`shifti`" procedure no longer dominates, replaced by "`pushi`". The following excerpt from "`pushi`" shows the performance problem.

```
60.  M-----------<   do m=1,mi
...
67.  M
68.  M Vs--------<      do larmor=1,4
69.  M Vs
70.  M Vs                  ij=jtion0(larmor,m)
71.  M Vs                  wp0=1.0-wpion(larmor,m)
72.  M Vs                  wt00=1.0-wtion0(larmor,m)
73.  M Vs                  e1=e1+wp0*wt00*(wz0*evector(1,kk,ij)+wz1*evector(1,kk+1,ij))
74.  M Vs                  e2=e2+wp0*wt00*(wz0*evector(2,kk,ij)+wz1*evector(2,kk+1,ij))
75.  M Vs                  e3=e3+wp0*wt00*(wz0*evector(3,kk,ij)+wz1*evector(3,kk+1,ij))
...
96.  M Vs-------->      enddo
97.  M
98.  M                  wpi(1,m)=0.25*e1
99.  M                  wpi(2,m)=0.25*e2
```

```
100.  M                  wpi(3,m)=0.25*e3
101.  M
102.  M----------->   enddo
```

The compiler vectorizes a loop over just 4 elements. The outer loop is over more than 32,000 elements, so it is appropriate to vectorize and multistream. A single directive does the trick.

```
 58.               !dir$ prefervector
 ...
 61.  MV----------<   do m=1,mi
 ...
 69.  MV 2--------<      do larmor=1,4
 70.  MV 2
 71.  MV 2                  ij=jtion0(larmor,m)
 72.  MV 2                  wp0=1.0-wpion(larmor,m)
 73.  MV 2                  wt00=1.0-wtion0(larmor,m)
 74.  MV 2                  e1=e1+wp0*wt00*(wz0*evector(1,kk,ij)+wz1*evector(1,kk+1,ij))
 75.  MV 2                  e2=e2+wp0*wt00*(wz0*evector(2,kk,ij)+wz1*evector(2,kk+1,ij))
 76.  MV 2                  e3=e3+wp0*wt00*(wz0*evector(3,kk,ij)+wz1*evector(3,kk+1,ij))
 ...
 97.  MV 2-------->      enddo
 98.  MV
 99.  MV                  wpi(1,m)=0.25*e1
100.  MV                  wpi(2,m)=0.25*e2
101.  MV                  wpi(3,m)=0.25*e3
102.  MV
103.  MV---------->   enddo
```

The directive on line 58 is the only change, and it appears as a comment to non-vector compilers.

Column "X1E r6" of Figure 1 shows the resulting performance for revision 6, a 1.8× improvement over revision 5.

2.4 Revisions 6 to 8

The profile for revision 6 begins as follows.

```
  Samp% | Cum.Samp% |    Samp |Function
                             |Line

 100.0% |    100.0% | 5146367 |Total
|-----------------------------------------
|  25.4% |     25.4% | 1306364 |MPI_CRAY_fast_barrier
...
||=======================================
|  22.2% |     47.6% | 1144908 |rng_number_s1_
...
||=======================================
|  19.2% |     66.9% |  989615 |shifti_
||---------------------------------------
||  12.3% |     59.9% |  631146 |line.248
||   4.2% |     64.1% |  214897 |line.249
||   1.0% |     65.1% |   51156 |line.128
```

The procedure "MPI_CRAY_fast_barrier" is inside the Cray MPI library, and it dominates because of load imbalance in the setup of the benchmark. The benchmark run is short, and the setup time is not included in the performance measurement, so this procedure is not relevant for optimization targeting real science runs. The procedure "rng_number_s1" is used to generate reproducible random numbers so the output of benchmark runs can be easily compared for correctness. It is also not included in the performance measurement, and it is replaced by more-efficient random-number generators for science runs.

Thus the procedure "shifti" returns as the bottleneck. The following excerpt from "shifti" shows the performance problem.

```
 240.                        lasth=msend
 241.  1-------------<       do i=1,msend
 242.  1                      m=kzi(i)
 243.  1                      if (m > mi) exit  !Break out of the DO loop if m > mi
 244.  1 2-----------<        do while(mtop == kzi(lasth))
 245.  1 2                      mtop=mtop-1
 246.  1 2                      lasth=lasth-1
 247.  1 2----------->        enddo
 248.  1                      if( nhybrid == 0 .and. mtop == ntracer )ntracer=m
 249.  1 r V M--<><><>        zion(1:nparam,m)=zion(1:nparam,mtop)
 250.  1 f----------<>         zion0(1:nparam,m)=zion0(1:nparam,mtop)
 251.  1                      mtop=mtop-1
 252.  1                      if (mtop == mi) exit  !Break out of the DO loop
 253.  1------------->       enddo

ftn-6254 ftn: VECTOR File = shifti.F90, Line = 241
  A loop starting at line 241 was not vectorized because a recurrence was found
  on "NTRACER" at line 248.
```

This code block packs local particles into the holes left by sent particles. The loop starting on line 241 iterates over the sent particles. Again the computation of the tracer particle inhibits vectorization. Loops on lines 249-250 are vectorized and multistreamed, but the value of "nparam" is no larger than 7, so the loops are very short.

This block from revision 6 was replaced by a completely different block in revision 8.

```
 246.                     #ifdef _UNICOSMP
 247.  r V M-----<><><>       zmask(mi+1:mtop) = .true.
 248.  V M---------<><>       i0=count(kzi(1:msend) <= mi)
 249.  r V M-----<><><>       zmask(kzi(i0+1:msend)) = .false.
 250.                         nholes=0
 251.  V-------------<        do m=mtop,mi+1,-1
 252.  V                         if (zmask(m)) then
 253.  V                            nholes=nholes+1
 254.  V                            jzi(nholes)=m
 255.  V                         endif
 256.  V------------->        enddo
 257.                         !dir$ concurrent, prefervector
 258.  Vm------------<        do i=1,nholes
 259.  Vm r 3 M--<><><>          zion(1:nparam,kzi(i))=zion(1:nparam,jzi(i))
 260.  Vm f----------<>           zion0(1:nparam,kzi(i))=zion0(1:nparam,jzi(i))
 261.  Vm------------->        enddo
 262.                         if (nhybrid == 0 .and. ntracer > mi) then
 263.                            itracer=0
 264.  MV------------<           do i=1,nholes
```

```
 265.  MV                          if (ntracer == jzi(i)) itracer=i
 266.  MV------------->          enddo
 267.                            ntracer=kzi(itracer)
 268.                         endif
```

Lines 247-256 identify the particles at the end of the array that will be used to fill in the holes. Lines 248-249 ensure that holes are not used to fill in other holes. Lines 251-256 perform a "`pack`"-like operation. Lines 258-261 copy the particles into the holes, vectorizing over holes and multistreaming over "`nparam`", which again is at most 7. Lines 262-268 move the tracer particle if necessary, again using a "`logical`" reduction.

Revision 8 corrects an additional performance problem appearing later in "`shifti`". The version from revision 6 is as follows.

```
 332.  M------------<   do m=1,mrecvleft(1)
 333.  M V 3------<><>      zion(1:nparam,m+mi)=recvleft(1:nparam,m)
 334.  M f----------<>      zion0(1:nparam,m+mi)=recvleft(nparam+1:nzion,m)
 335.  M------------>   enddo
  ...
 339.  M------------<   do m=1,mrecvright(1)
 340.  M V 3------<><>      zion(1:nparam,m+mi+mrecvleft(1))=recvright(1:nparam,m)
 341.  M f----------<>      zion0(1:nparam,m+mi+mrecvleft(1))=recvright(nparam+1:nzion,m)
 342.  M------------>   enddo

ftn-6294 ftn: VECTOR File = shifti.F90, Line = 332
  A loop starting at line 332 was not vectorized because a better candidate was
  found at line 333.

ftn-6294 ftn: VECTOR File = shifti.F90, Line = 339
  A loop starting at line 339 was not vectorized because a better candidate was
  found at line 340.
```

The compiler does not know that "`nparam`" is small and chooses to vectorize the wrong loop. Directives solve the vectorization problem, and the compiler multistreams over the "`nparam`" loop, which is appropriate.

```
 363.                   !dir$ prefervector
 364.  Vm------------<   do m=1,mrecvleft(1)
 365.  Vm r 3 M--<><><>      zion(1:nparam,m+mi)=recvleft(1:nparam,m)
 366.  Vm f----------<>      zion0(1:nparam,m+mi)=recvleft(nparam+1:nzion,m)
 367.  Vm------------>   enddo

 371.                   !dir$ prefervector
 372.  Vm------------<   do m=1,mrecvright(1)
 373.  Vm r 3 M--<><><>      zion(1:nparam,m+mi+mrecvleft(1))=recvright(1:nparam,m)
 374.  Vm f----------<>      zion0(1:nparam,m+mi+mrecvleft(1))=recvright(nparam+1:nzion,m)
 375.  Vm------------>   enddo
```

Column "X1E r8" of Figure 1 shows the resulting performance for revision 8, a 1.6× improvement over revision 6.

2.5 Revisions 8 to 9

The profile for revision 8 begins as follows.

```
      Samp% | Cum.Samp% |    Samp |Function
                                 |Line
```

```
 100.0% |    100.0% | 4205531 |Total
|-----------------------------------------
|  30.9% |     30.9% | 1297644 |MPI_CRAY_fast_barrier
||----------------------------------------
...
||========================================
|  26.8% |     57.7% | 1127354 |rng_number_s1_
||----------------------------------------
...
||========================================
|  15.5% |     73.2% |  652792 |chargei_
||----------------------------------------
...
||========================================
|  11.6% |     84.7% |  486147 |pushi_
||----------------------------------------
...
||========================================
|   2.4% |     92.4% |  100005 |shifti_
||----------------------------------------
```

Again, "MPI_CRAY_fast_barrier" and "rng_number_s1" are not relevant. The "shifti" procedure is now much faster, but "chargei" and "shifti" are back near the top of the profile. The loopmark listing for "chargei" showed no obvious performance problems, but the loopmark listing for "pushi" did. The problems were not clear, however, looking at just the loops themselves. Here is an example.

```
 259.  MV----------<           do m=1,mi
 260.  MV                        ip=max(1,min(mflux,1+int((wpi(1,m)-a0)*d_inv)))
 261.  MV                        dtem(ip)=dtem(ip)+wpi(2,m)*zion(5,m)
 262.  MV                        dden(ip)=dden(ip)+1.0
 263.  MV---------->           enddo

 301.  MV----------<     do m=1,mi
  ...
 305.  MV                    ip=max(1,min(mflux,1+int((r-a0)*d_inv)))
 306.  MV                    ii=max(0,min(mpsi,int((r-a0)*delr+0.5)))
  ...
 310.  MV                    hfluxpsi(ii)=hfluxpsi(ii)+vdrenergy ! energy flux profile
  ...
 313.  MV                    rmarker(ip)=rmarker(ip)+zion0(6,m)
 314.  MV                    eflux(ip)=eflux(ip)+vdrenergy
  ...
 324.  MV                    dmark(ip)=dmark(ip)+wpi(1,m)*r
 325.  MV                    dden(ip)=dden(ip)+1.0
 326.  MV---------->     enddo
```

These loops look fine; they are vectorized and multistreamed. Looks are deceiving, however, as indicated by the comments later in the loopmark listing.

```
ftn-6371 ftn: VECTOR File = pushi.f90, Line = 259
  A vectorized loop contains potential conflicts due to indirect addressing at
```

```
  line 262, causing less efficient code to be generated.

ftn-6371 ftn: VECTOR File = pushi.f90, Line = 259
  A vectorized loop contains potential conflicts due to indirect addressing at
  line 261, causing less efficient code to be generated.

ftn-6371 ftn: VECTOR File = pushi.f90, Line = 301
  A vectorized loop contains potential conflicts due to indirect addressing at
  line 310, causing less efficient code to be generated.

ftn-6371 ftn: VECTOR File = pushi.f90, Line = 301
  A vectorized loop contains potential conflicts due to indirect addressing at
  line 314, causing less efficient code to be generated.

ftn-6371 ftn: VECTOR File = pushi.f90, Line = 301
  A vectorized loop contains potential conflicts due to indirect addressing at
  line 325, causing less efficient code to be generated.

ftn-6371 ftn: VECTOR File = pushi.f90, Line = 301
  A vectorized loop contains potential conflicts due to indirect addressing at
  line 313, causing less efficient code to be generated.

ftn-6371 ftn: VECTOR File = pushi.f90, Line = 301
  A vectorized loop contains potential conflicts due to indirect addressing at
  line 324, causing less efficient code to be generated.
```

The problem in the above code fragment is that the index variables "`ii`" and "`ip`" are not permutations. The fix is to use work arrays with an extra, vectorizable dimension, like for "`chargei`" in revision 3.

```
   19.                  #ifdef _UNICOSMP
   20.                    integer, parameter :: vlen=256
   21.                    integer :: mv, v
   22.                    real(wp) :: vdtem(vlen,mflux), vdden(vlen,mflux)
   23.                    real(wp) :: vhfluxpsi(0:mpsi,vlen),
                                      vrmarker(vlen,mflux), veflux(vlen,mflux),
                                      vdmark(vlen,mflux)
   24.                  #endif
```

It is important to note the size of the original dimension for each variable to efficiently implement the reduction over the added dimension. The value of "`mpsi`" is as much as 192, so it is good to vectorize over when reducing "`vhfluxpsi`" and thus it appears as the first dimension. The value of "`mflux`", however, is just 5, so it is better to perform a vector reduction over "`vlen`" for the other variables, with the "`mflux`" dimension second.

As an example of using these work variables, lines 259-263 above were replaced with the following.

```
  264.                  #ifdef _UNICOSMP
  265.  Vw V M----<><><>           vdtem=0
  266.  f-------------<>           vdden=0
  267.  m--------------<           do mv=1,mi,vlen
  268.  m MVs----------<              do m=mv,min(mv+vlen-1,mi)
  269.  m MVs                            v=m-mv+1
  270.  m MVs                            ip=max(1,min(mflux,1+int((wpi(1,m)-a0)*d_inv)))
  271.  m MVs                            vdtem(v,ip)=vdtem(v,ip)+wpi(2,m)*zion(5,m)
```

```
272.  m MVs                              vdden(v,ip)=vdden(v,ip)+1.0
273.  m MVs---------->                 enddo
274.  m-------------->               enddo
275.  M--------------<               do i=1,mflux
276.  M Vw V 4--<><><>                  dtem(i)=sum(vdtem(:,i))
277.  M f-----------<>                  dden(i)=sum(vdden(:,i))
278.  M-------------->               enddo
```

Lines 265-274 update the new work variables, and lines 275-278 sum the results. This is a small example within "pushi"; other loops with more operations were also modified.

Column "X1E r9" of Figure 1 shows the resulting performance for revision 9, a 1.1× improvement over revision 8.

2.6 Revisions 9 to 12

Here is an excerpt from the profile for revision 9.

```
|  16.3% |     77.1% |  651612 |chargei_
...
|   9.4% |     86.5% |  375015 |pushi_
...
|   1.6% |     93.2% |   64880 |shifti_
...
|   0.6% |     96.1% |   22933 |poisson_
...
|   0.2% |     98.1% |    9468 |poisson_initial_
```

There is no clear target for optimization among the most-costly procedures. We instead went for the remaining "low-hanging fruit", scattered optimizations that should not adversely affect performance on non-vector systems.

In "pushi", we undid the modification to lines 259-263 described above because it was not worth the additional overhead. The analogous modifications for other, larger loops in "pushi" were worth the modification, however. In "shifti", we manually blocked a "pack" operation for multistreaming and moved multistreaming away from some "nparam" loops. In "smooth", we again applied the added-dimension technique to vectorize dependent updates.

We did perform one optimization technique not already described, in "poisson". Here is the original code.

```
 86.  m iW M-----------<         do i=1,mgrid
 87.  m iW M                        ptilde(i)=0.0
 88.  m iW M Vr--------<            do j=1,nindex(i,k)
 89.  m iW M Vr                        ptilde(i)=ptilde(i)
                                       +ring(j,i,k)*phitmp(indexp(j,i,k))
 90.  m iW M Vr-------->            enddo
 91.  m iW M----------->         enddo
```

All values of "nindex" are 65 or less, while "mgrid" is over 32,000. Therefore, it is better to vectorize the "i" loop, but the "j" loop must have constant loop bounds to allow this. Because unused elements of "ring" are set to zero, we can use the maximum value of "nindex" and vectorize the outer loop at the potential cost of extra work for the inner loop.

```
312.  V M-----------<>*<>   max_nindex=maxval(nindex)

 87.  m iW MVr---------<         do i=1,mgrid
```

```
88.  m iW MVr                        ptilde(i)=0.0
89.  m iW MVr           #ifdef _UNICOSMP
90.  m iW MVr r-------<              do j=1,max_nindex
91.  m iW MVr r         #else
92.  m iW MVr r                     do j=1,nindex(i,k)
93.  m iW MVr r         #endif
94.  m iW MVr r                        ptilde(i)=ptilde(i)
                                         +ring(j,i,k)*phitmp(indexp(j,i,k))
95.  m iW MVr r------->              enddo
96.  m iW MVr--------->           enddo
```

Line 312 comes from a separate initialization routine, but it is listed above the dependent code fragment for simplicity.

Column "X1E r12" of Figure 1 shows the resulting performance for revision 12, a 1.06× improvement over revision 9.

2.7 Revisions 12 to 24

The profile for revision 12 begins as follows.

```
 Samp% | Cum.Samp% |    Samp |Function
                             |Line

 100.0% |    100.0% | 3994895 |Total
|---------------------------------------
...
||======================================
|  16.3% |     77.1% |  651612 |chargei_
||--------------------------------------
||   3.6% |     64.5% |  145767 |line.120
||   2.3% |     66.8% |   93233 |line.76
||   2.0% |     68.8% |   80616 |line.55
||   1.9% |     70.8% |   77887 |line.153
||   1.1% |     71.9% |   44914 |line.149
...
||======================================
|   9.4% |     86.5% |  375015 |pushi_
||--------------------------------------
||   1.3% |     78.4% |   50273 |line.99
||   1.2% |     79.6% |   48823 |line.100
```

Performance of "`chargei`" can be improved no further without affecting performance on other systems or dramatically decreasing code maintainability, for the improvements require changing the order of dimensions for global variables. The following excerpt from "`chargei`" shows a representative performance problem.

```
265.  M-------------<   do i=0,mpsi
266.  M                    !dir$ prefervector
267.  M V-----------<      do j=1,mtheta(i)
268.  M V r---------<         do k=1,mzeta
269.  M V r                    ij=igrid(i)+j
270.  M V r                    zonali(i)=zonali(i)+0.25*densityi(k,ij)
271.  M V r                    densityi(k,ij)=0.25*densityi(k,ij)*markeri(k,ij)
272.  M V r--------->         enddo
```

```
273.  M V----------->      enddo
274.  M------------->   enddo
```

Vectorization is over the second dimension of "density" and "markeri". A different code block has a similar problem.

```
101.  m MVs---------<   do m=mv,min(mv+vlen-1,mi)
102.  m MVs                v=m-mv+1
 ...
112.  m MVs 3-------<      do larmor=1,4
113.  m MVs 3                 wp1=wpion(larmor,m)
114.  m MVs 3                 wp0=1.0-wp1
 ...
140.  m MVs 3                 ij=jtion0(larmor,m)
141.  m MVs 3                 vdensityi(ij,kk,v) = vdensityi(ij,kk,v) + wz0*wt00
 ...
175.  m MVs 3------->      enddo
176.  m MVs--------->   enddo
177.  m               #ifdef _UNICOSMP
178.  m------------->   enddo
```

Vectorization is again over the second dimension, this time for "wpion", "jtion0", and others. (Note that the "larmor" loop iterates over strided vector operations.)

We improved performance on the X1E by changing the order of dimensions for various variables so that the vectorized dimension was first. We modified the following variables: "densityi", "dnitmp", "markeri", "jtion0", "jtion1", "wpion", "wtion0", "wtion1", "markere", "densitye", "zion", "zion0". These modifications affected the following files, all with the ".F90" suffix: "shifte", "shifti", "pushi", "poisson", "chargee", "snapshot", "setup", "chargei", "smooth", "restart", "tracking", "set_random_values".

Previous optimizations only affect the X1 series; on other systems, they are either ignored as comments or deleted by the preprocessor. The changes in dimension order affect all systems, however.

Column "X1E r24" of Figure 1 shows the resulting performance for revision 24, a 1.26× improvement over revision 12.

3 GTC on XT3

We did not perform an orderly performance-optimization process on the XT3, unlike on the X1E. Optimizing for Opterons or other super-scalar processors is not as systematic as optimizing for vector processors, but they can also be far less sensitive than vector processors to performance degradation from un-optimized code.

We present XT3 results that test sensitivity to three variations: small or large memory pages, changes in dimension order, and changes in compiler version. Column "XT3 r2 6.0" of Figure 1 shows the performance of revision 2 of the code, compiled with version 6.0 of the PGI compilers, using the default, large page size. Column "XT3 r2 6.1.3" shows the performance of the same code using version 6.1.3 of the compilers, a 1.05× improvement. Column "XT3 r2 small" shows the performance of the code compiled with the version 6.1.3 compilers and run with the option "-small_pages". Small pages provide a 1.4× performance improvement. Column "XT3 r24" shows the performance of revision 24 of the code, compiled with the 6.1.3 compilers and using the default page size. This column shows that the changes of dimension order that improved performance on X1E degrade performance on XT3 by a factor of 0.57. Column "XT3 r24 small" shows the performance of the same revision run with the "-small_pages" option. The degradation of performance versus revision 2 is smaller than for large pages, but performance is still significantly lower for revision 24, at 0.87×.

The performance of GTC on the XT3 has an unusually high sensitivity to the memory page size. We speculate that this sensitivity comes from the irregular memory access patterns that arise naturally from

"particle pushing". For large pages, the translation-lookaside buffer (TLB) on the Opteron has just 8 entries, whereas it has 512 entries for small pages.[5] GTC can use a number of variables within the same loop, with irregular access to those variables. With large pages, GTC may use more distinct pages within a single loop iteration than there are TLB entries, leading to "thrashing" of the TLB. With small pages, the TLB should have enough entries to cover the different pages used within a single loop iteration.

4 Results

Figure 2 shows performance results for the GTC benchmark used for X1E optimization.[2] We used 64-processor runs for optimization, while the figure has results for a wide range of processor counts and a wide variety of systems. The benchmark is weak scaling, meaning the number of particles per processor stays constant as the number of processors increases.

The X1E results use our revision 24, while the other vector systems use an earlier vector version not described here. The XT3 results use revision 2 with version 6.0 of the PGI compilers. Version 6.1.3 compilers should produce slightly better performance. The Blue Gene/L results come from using a single processor per node for computation, though two are available on each node. Also, the results are for a different configuration of the benchmark, one using a tenth of the number of particles per node because of memory limitations.

With the differing versions, the X1E is the fastest system per processor, faster than the Earth Simulator and the NEC SX-8. The Earth Simulator remains the fastest system overall, but it may be overtaken by the XT3 after upgrades scheduled for later in 2006.[3] The Blue Gene/L system at 16,000 nodes has performance equivalent to about 1,000 MSPs of the X1E, with the caveats that the problem size per processor is smaller and only 16,000 of the 32,000 dedicated processors were used for computation. Meanwhile, 4,000 processors on the XT3 beat 1,000 X1E MSPs.

Through significant effort, we were able to dramatically improve the performance of GTC on the X1E, making the X1E the fastest system per processor. Yet XT3 performance may be equally impressive; with the simple runtime switch to small pages, the XT3 provides better than one fourth the performance per processor relative to X1E without any source-code optimization. Since the peak performance of an X1E MSP is 18 GF and the peak a 2.4 GHz Opteron is 4.8 GF, the efficiencies are roughly equivalent.

In 2005, we performed GTC science runs on the X1E at groundbreaking resolutions, over a billion particles, to explore the convergence properties of GTC. Now, runs with *tens* of billions of particles are performed regularly on the XT3 using 4,800 processors at a time, consuming millions of processor hours. Future upgrades to the XT3 will enable runs that are larger still.

Acknowledgments

James Schwarzmeier of Cray, Inc., performed optimizations on earlier versions of GTC, and some of his ideas were applied here. In particular, he used the technique of adding a dimension to eliminate accumulation dependences, followed by a reduction over the added dimension.

This research was sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

References

[1] Stephane Ethier. "Performance Study of the 3D Particle-in-Cell Code GTC on the Cray X1." *CUG 2004.*

[2] Leonid Oliker, Jonathan Carter, Michael Wehner, Andrew Canning, Stephane Ethier, Art Mirin, David Parks, Patrick H. Worley, Shigemune Kitawaki, Yoshinori Tsuda. "Leading Computational Methods on Scalar and Vector HEC Platforms." *SC 2005.*

[3] Arthur Bland, Ann Baker, R. Shane Canon, Ricky Kendall, Jeffrey Nichols, and Julia White. "Leadership Computing at the NCCS." *CUG 2006*.

[4] `http://subversion.tigris.org/`

[5] Hans de Vries. "Understanding the detailed Architecture of AMD's 64 bit Core." September 21, 2003. `http://www.chip-architect.com/news/2003_09_21_Detailed_Architecture_of_AMDs_64bit_Core.html`

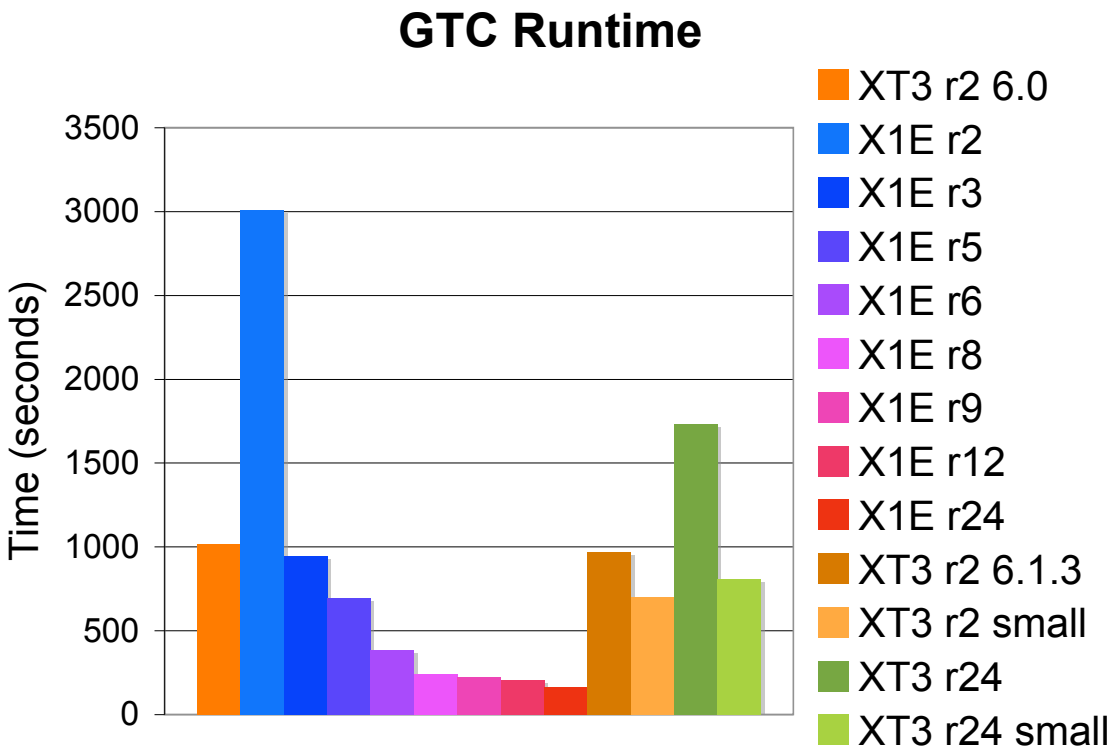

Figure 1: Runtime of a 64-process GTC benchmark on X1E and XT3 for various revisions of source code. "XT3 r2 6.0" is from revision 2 on the XT3 using version 6.0 of the PGI compiler. "X1E rn" is from revision n on the X1E. Columns marked "small" were run on the XT3 using small pages.

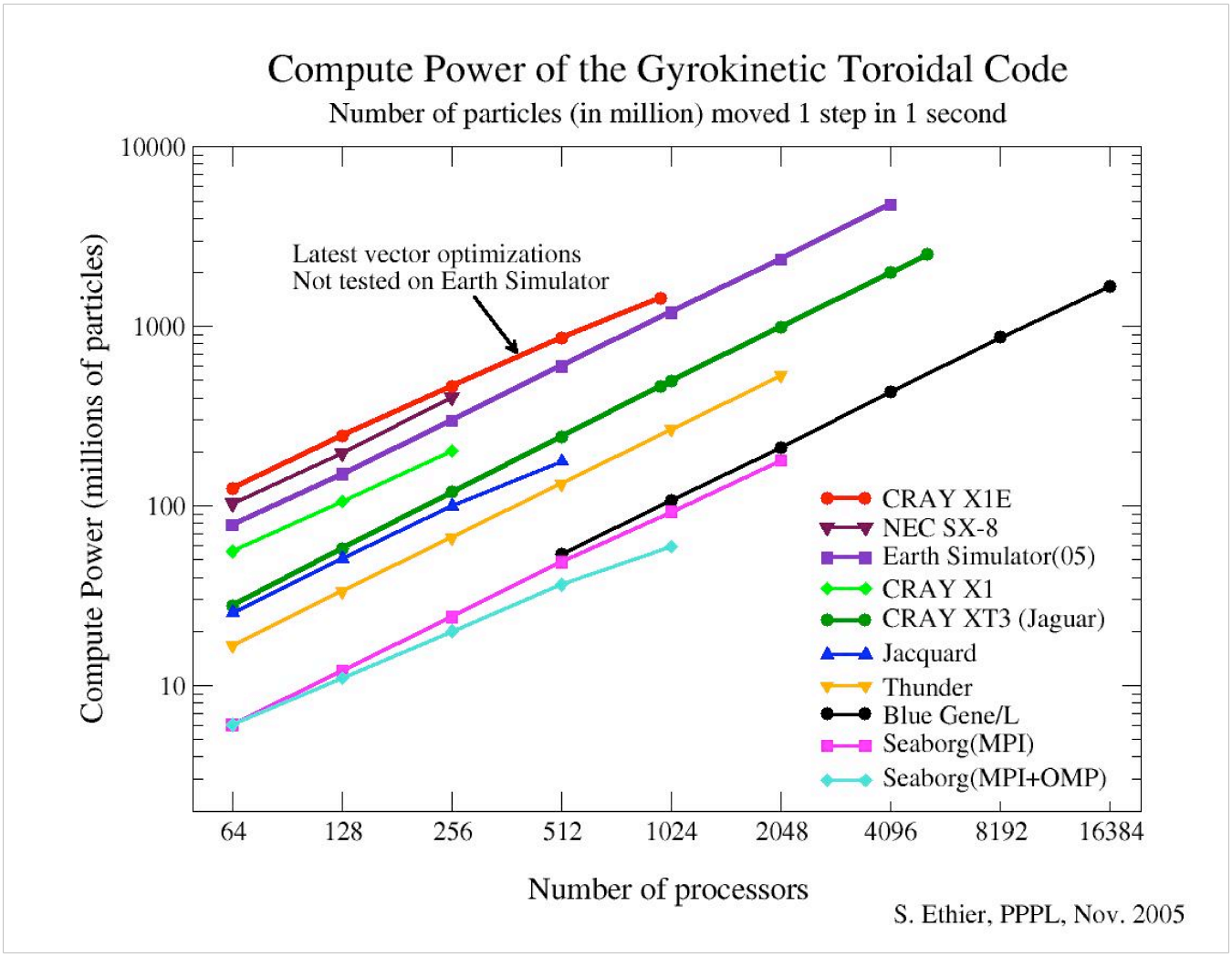

Figure 2: "Compute power" of GTC, as measured using a weak-scaling benchmark.[2] As noted, the X1E line represents a different version than the Earth Simulator and SX-8 lines—in particular, the version described in this paper. The XT3 line represents an older version of the OS and version 6.0 of the PGI compiler. The Blue Gene/L line uses a slightly different benchmark, one that has a tenth of the number of particles per processor because of memory limitations.