POP'16: Sugestões sobre as Soluções Submetidas e Apresentação da Solução de Referência

Ricardo Nobre
ricardo.nobre@fe.up.pt

9 Novembro 2016

http://specs.fe.up.pt/

DEI Departamento de Engenharia Informática

Sobre as Submissões

Em 14 submissões finais:

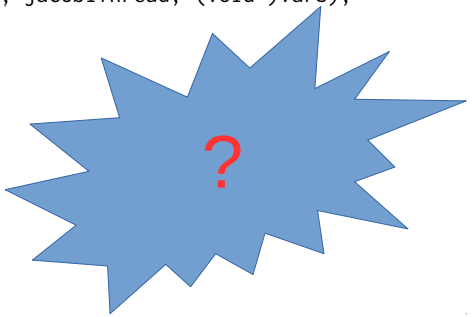
- 5 não passam os testes com inputs aleatórios
- 5 não usaram os niveis de otimização standard (p.ex., -O2) nem otimizações individuais do GCC
- Maior parte não gera codigo vectorizado
- Não houve otimizações especificas para a arquitetura do ANTAREX em termos do assembly gerado nem da arquitetura da memória

Sobre as Submissões

2rey:

- Problema na sincronização das pthreads, e além disso nas flags submetidas só usam “-pthread”

```
float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd,
Matrix *wrk1, Matrix *wrk2) {

    int i, imax = p->mrows - 1;
    size_t jacobiVarsSize = sizeof(struct jacobiVars);

    if(sem_init(&numThreads, 0, MACHINE_NUM_THREADS * 16) != 0)
        printf("Unable to initialize semaphore.\n");
    pthread_t th;

    for (i = 1; i < imax; i++) {
                sem_wait(&numThreads);
                struct jacobiVars *vars = (struct
jacobiVars*)malloc(jacobiVarsSize);
                vars->a = a;
                vars->b = b;
                vars->c = c;
                vars->p = p;
                vars->bnd = bnd;
                vars->wrk1 = wrk1;
                vars->wrk2 = wrk2;
                vars->i = i;
                pthread_create(&th, NULL, jacobiThread, (void*)vars);
    }

    return (gosa);
}
```




```
float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd,
Matrix *wrk1, Matrix *wrk2) {

    int i, imax = p->mrows - 1;
    size_t jacobiVarsSize = sizeof(struct jacobiVars);

    if(sem_init(&numThreads, 0, MACHINE_NUM_THREADS * 16) != 0)
        printf("Unable to initialize semaphore.\n");
    pthread_t* threads = malloc(sizeof(*threads) * (imax - 1));

    for (i = 1; i < imax; i++) {
                sem_wait(&numThreads);
                struct jacobiVars *vars = (struct
jacobiVars*)malloc(jacobiVarsSize);
                vars->a = a;
                vars->b = b;
                vars->c = c;
                vars->p = p;
                vars->bnd = bnd;
                vars->wrk1 = wrk1;
                vars->wrk2 = wrk2;
                vars->i = i;
                pthread_create(&(threads[i - 1]), NULL, jacobiThread,
(void*)vars);
    }

    for (int i = 1; i < imax; ++i) {
        pthread_join(threads[i - 1], NULL);
    }

    return (gosa);
}
```

Sobre as Submissões

6ucr:

- Variáveis *ss* e *s0* não deveriam ser *shared*, código funciona por acaso (o standard do OpenMP não garante)

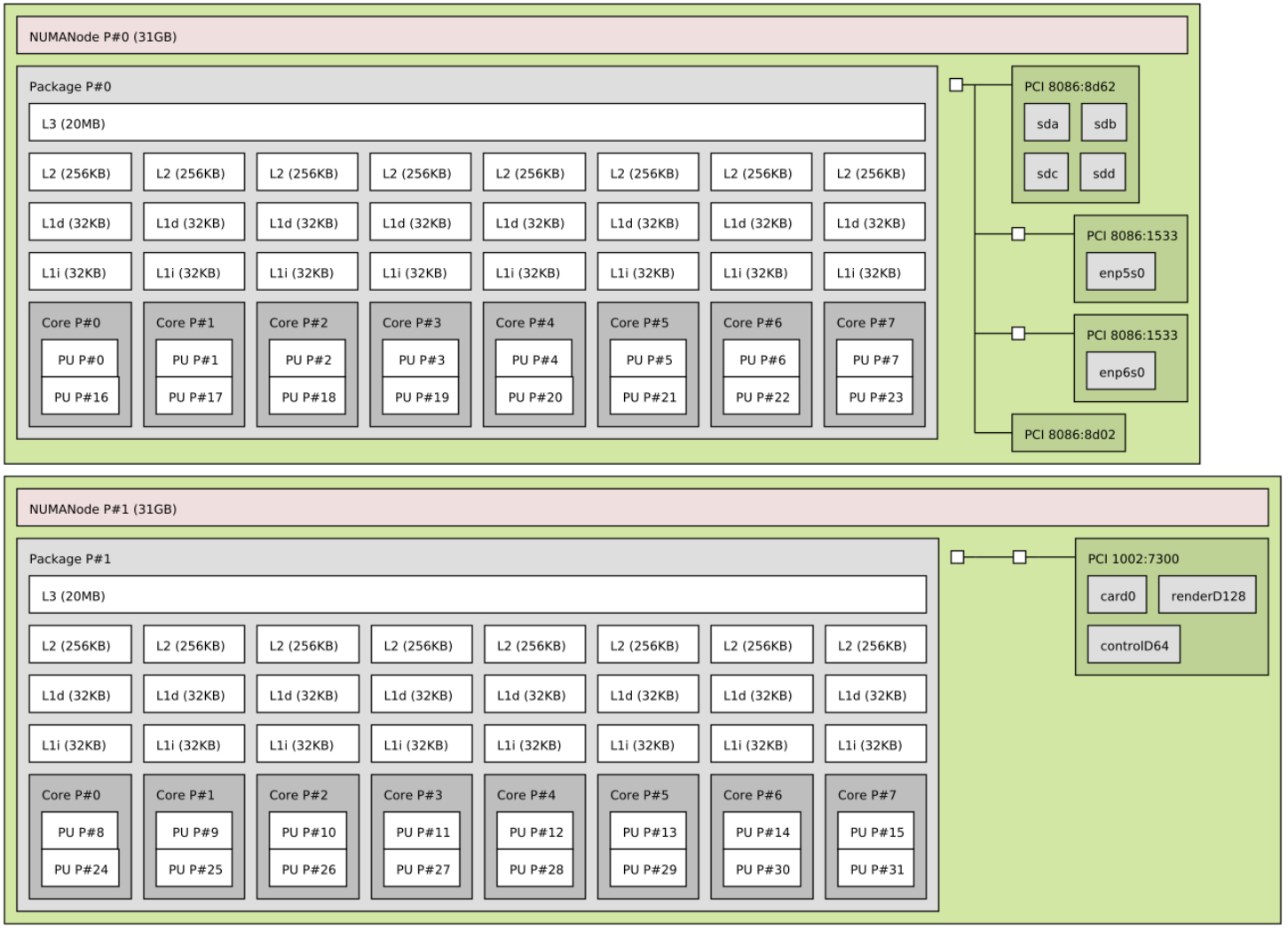
```
 float jacobi(Matrix * a, Matrix * b, Matrix * c, Matrix * p,
Matrix * bnd, Matrix * wrk1, Matrix * wrk2) {

   int i, j, k, imax, jmax, kmax;
   float gosa, s0, ss;
   const float omega = 0.8;

   imax = p->mrows - 1;
   jmax = p->mcols - 1;
   kmax = p->mdeps - 1;

   gosa = 0.0;
   #pragma omp parallel for num_threads(32) private(j,k)
shared(a,b,c,wrk1,wrk2,p,ss,s0,) reduction(+:gosa)

   for (i = 1; i < imax; i++) {
       for (j = 1; j < jmax; j++) {
           for (k = 1; k < kmax; k++) {
               ...
           }
       }
   }

   return (gosa);
 }
```

~1000

```
 float jacobi(Matrix * a, Matrix * b, Matrix * c, Matrix * p,
Matrix * bnd, Matrix * wrk1, Matrix * wrk2) {

   int i, j, k, imax, jmax, kmax;
   float gosa, s0, ss;
   const float omega = 0.8;

   imax = p->mrows - 1;
   jmax = p->mcols - 1;
   kmax = p->mdeps - 1;

   gosa = 0.0;
   #pragma omp parallel for num_threads(32) private(j,k,ss,s0)
 shared(a,b,c,wrk1,wrk2,p) reduction(+:gosa)

   for (i = 1; i < imax; i++) {
       for (j = 1; j < jmax; j++) {
           for (k = 1; k < kmax; k++) {
               ...
           }
       }
   }

   return (gosa);
 }
```

~2500

Topology of ANTAREX Machine



Non-uniform memory access (NUMA) architecture.

2×Xeon E5 2630 V3 @2.4GHz (Boost disabled) 16/32 physical/logical cores

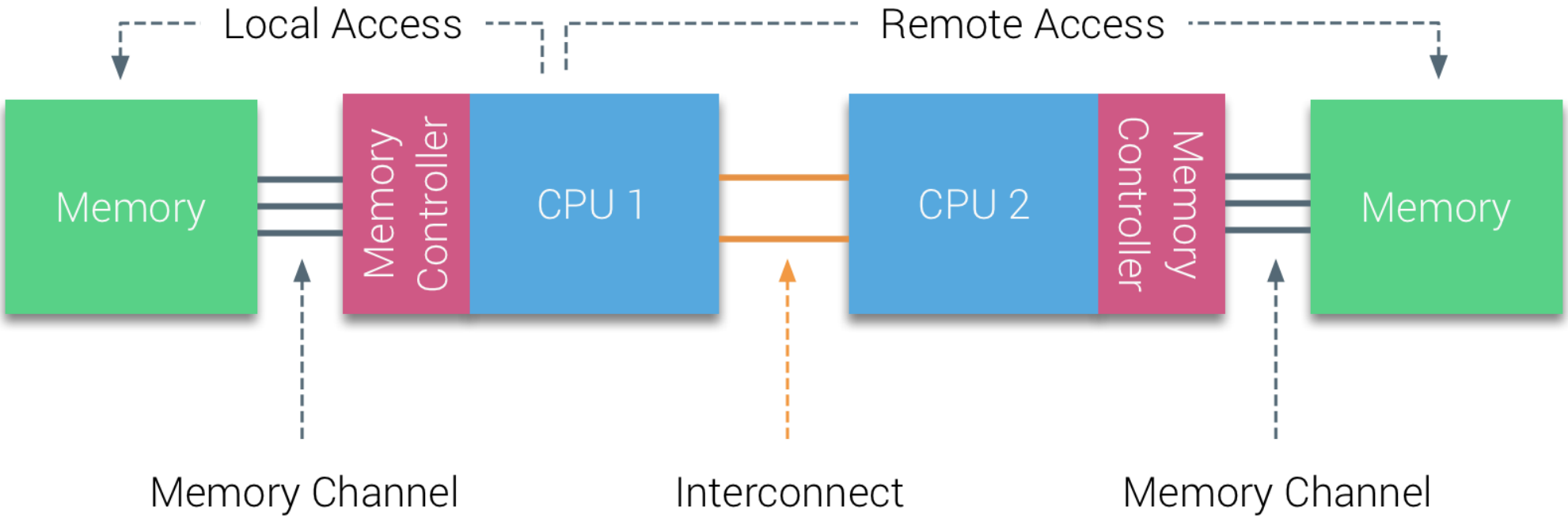
Each CPU has 32GB (16GB per channel).

Can access RAM of other CPU but there is an overhead.

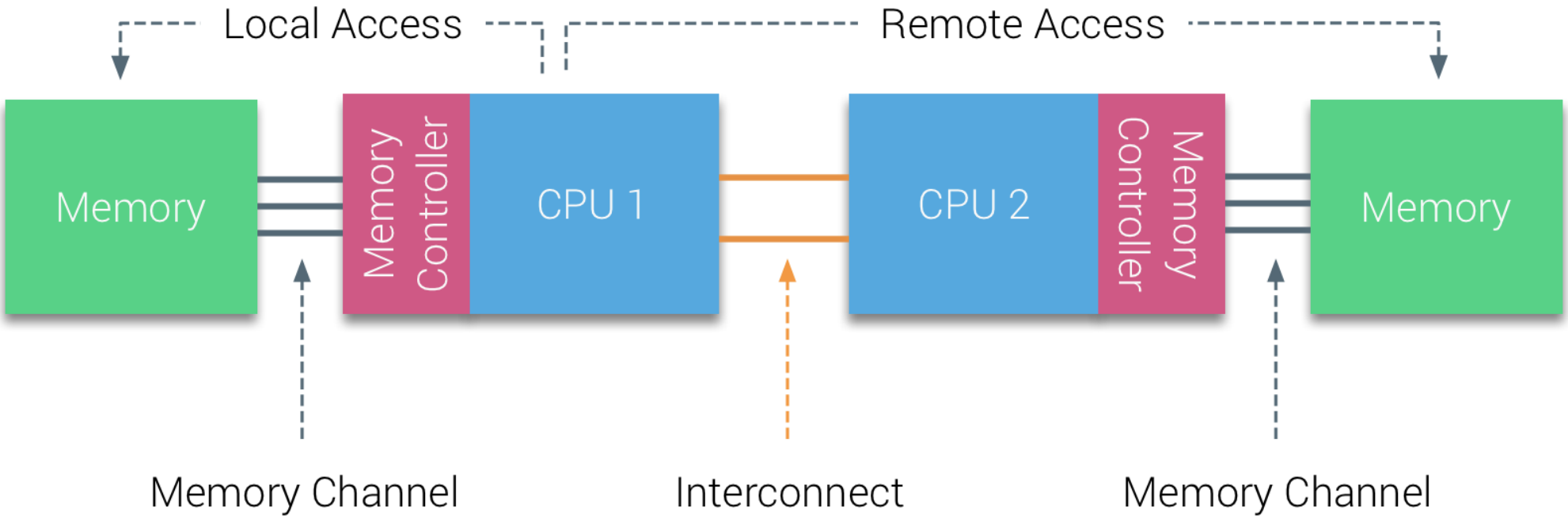
Access to local RAM is much faster than remote RAM.

How much faster?

NUMA: Local vs Remote Access



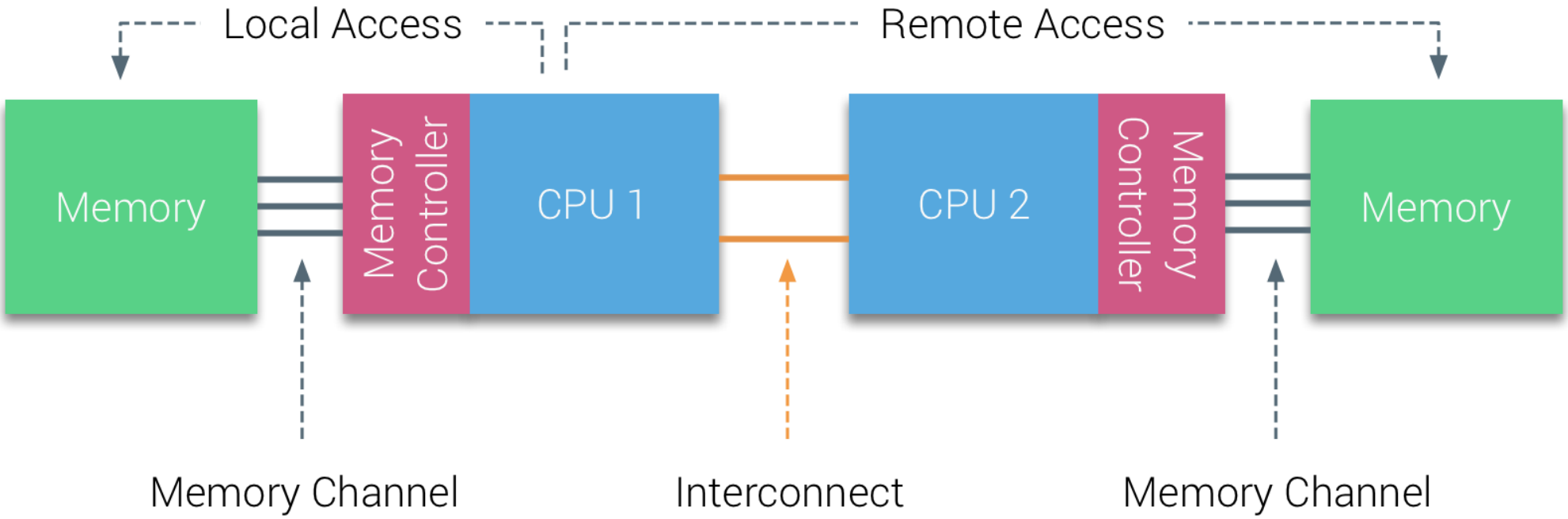
NUMA: Local vs Remote Access



NUMA	CPU 0	CPU 1
CPU 0	77.3 ns	121.7 ns
CPU 1	121.4 ns	74.8 ns

local memory latencies and cross-socket memory latencies in ns

NUMA: Local vs Remote Access



NUMA	CPU 0	CPU 1
CPU 0	77.3 ns	121.7 ns
CPU 1	121.4 ns	74.8 ns

local memory latencies and cross-socket memory latencies in ns

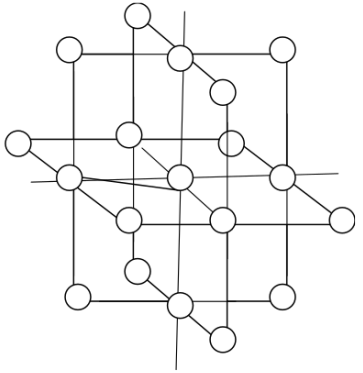
NUMA	CPU 0	CPU 1
CPU 0	28499.6	8566.5
CPU 1	8535.2	28506.5

local memory bandwidth and cross-socket memory bandwidth in MB/s

jacobi.c

```
float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int i, j, k, imax, jmax, kmax;
    float gosa, s0, ss;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    for (i = 1; i < imax; i++) {
        for (j = 1; j < jmax; j++) {
            for (k = 1; k < kmax; k++) {
                s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                     + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                     + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                     + MR(b, 0, i, j, k)
                       * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                          - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                     + MR(b, 1, i, j, k)
                       * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                          - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                     + MR(b, 2, i, j, k)
                       * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                          - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                     + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                     + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                     + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                     + MR(wrk1, 0, i, j, k);
                ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                gosa += ss * ss;
                MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
            }
        }
    }
    return (gosa);
}
```

19 accesses to *p* per iteration



13 pointwise accesses:

- a0, a1, a2, a3
- b0, b1, b2
- c0, c1, c2
- bnd
- wrk1, wrk2

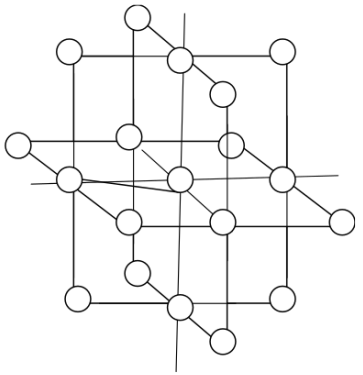
32 memory accesses per iteration in total

Original code & no optimization

jacobi.c

```
float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int i, j, k, imax, jmax, kmax;
    float gosa, s0, ss;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    for (i = 1; i < imax; i++) {
        for (j = 1; j < jmax; j++) {
            for (k = 1; k < kmax; k++) {
                s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                     + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                     + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                     + MR(b, 0, i, j, k)
                       * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                          - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                     + MR(b, 1, i, j, k)
                       * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                          - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                     + MR(b, 2, i, j, k)
                       * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                          - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                     + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                     + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                     + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                     + MR(wrk1, 0, i, j, k);
                ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                gosa += ss * ss;
                MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
            }
        }
    }
    return (gosa);
}
```

19 accesses to *p* per iteration



13 pointwise accesses:

- a0, a1, a2, a3
- b0, b1, b2
- c0, c1, c2
- bnd
- wrk1, wrk2

32 memory accesses per iteration in total

Xeon E5 2630 V3 @2.4GHz:

- 8 Intel Haswell cores
- 32 FMA SP FLOPs/cycle

2.4×32×8 → **614.4 GFLOPS** per CPU

Real life GFLOPS will be limited by how fast the memory subsystem can feed the CPUs

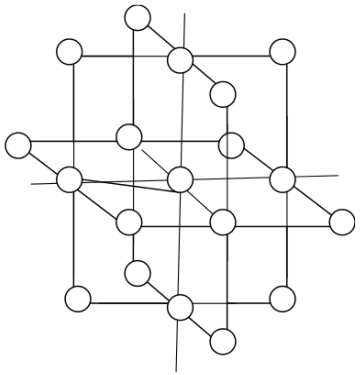
Memory is the bottleneck

Original code & no optimization

jacobi.c

```
float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int i, j, k, imax, jmax, kmax;
    float gosa, s0, ss;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    for (i = 1; i < imax; i++) {
        for (j = 1; j < jmax; j++) {
            for (k = 1; k < kmax; k++) {
                s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                     + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                     + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                     + MR(b, 0, i, j, k)
                       * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                          - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                     + MR(b, 1, i, j, k)
                       * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                          - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                     + MR(b, 2, i, j, k)
                       * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                          - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                     + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                     + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                     + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                     + MR(wrk1, 0, i, j, k);
                ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                gosa += ss * ss;
                MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
            }
        }
    }
    return (gosa);
}
```

19 accesses to *p* per iteration



13 pointwise accesses:

- a0, a1, a2, a3
- b0, b1, b2
- c0, c1, c2
- bnd
- wrk1, wrk2

32 memory accesses per iteration in total

float → 4 bytes

L matrix = 256×256×512

Bandwidth (local) = 2×28.5 GB

Minimum possible execution time for size L in a memory bandwidth limited scenario and supposing no cache?

32×4×256×256×512 bytes in memory accesses

32×4×256×256×512 /(28.5*2*10^9)→**75.4 ms**

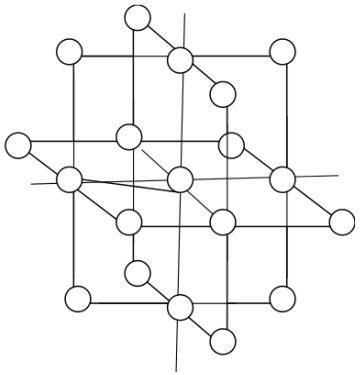
And considering cache?

Original code & no optimization

jacobi.c

```
float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int i, j, k, imax, jmax, kmax;
    float gosa, s0, ss;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    for (i = 1; i < imax; i++) {
        for (j = 1; j < jmax; j++) {
            for (k = 1; k < kmax; k++) {
                s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                     + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                     + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                     + MR(b, 0, i, j, k)
                       * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                          - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                     + MR(b, 1, i, j, k)
                       * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                          - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                     + MR(b, 2, i, j, k)
                       * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                          - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                     + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                     + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                     + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                     + MR(wrk1, 0, i, j, k);
                ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                gosa += ss * ss;
                MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
            }
        }
    }
    return (gosa);
}
```

19 accesses to *p* per iteration



13 pointwise accesses:

- a0, a1, a2, a3
- b0, b1, b2
- c0, c1, c2
- bnd
- wrk1, wrk2

32 memory accesses per iteration in total

Xeon E5 2630 V3:

- 8×32KB L1 data cache
- 8×256KB L2 / 20MB L3

The number of accesses to *p* in RAM per iteration is reduced by the number of accesses that are in cache when requested

Were *(i, j)* and *(i,j+1)* in the previous iteration of the *j* loop

Supposing that cache can hold the (i,j-1) and (i,j) *p* data accesses then only 6 accesses to *p* have to be done in RAM

32 memory accesses are reduced to 19 (13+6)

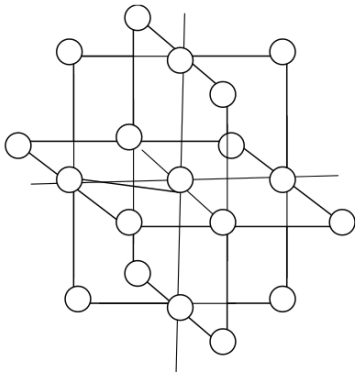
19×4×256×256×512 bytes in memory accesses→**44.7 ms**

Original code & no optimization

jacobi.c

```
float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int i, j, k, imax, jmax, kmax;
    float gosa, s0, ss;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    for (i = 1; i < imax; i++) {
        for (j = 1; j < jmax; j++) {
            for (k = 1; k < kmax; k++) {
                s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                     + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                     + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                     + MR(b, 0, i, j, k)
                       * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                          - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                     + MR(b, 1, i, j, k)
                       * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                          - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                     + MR(b, 2, i, j, k)
                       * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                          - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                     + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                     + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                     + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                     + MR(wrk1, 0, i, j, k);
                ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                gosa += ss * ss;
                MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
            }
        }
    }
    return (gosa);
}
```

19 accesses to *p* per iteration



13 pointwise accesses:
- a0, a1, a2, a3
- b0, b1, b2
- c0, c1, c2
- bnd
- wrk1, wrk2

32 memory accesses per iteration in total

100

3400 ms
(76× slower)

compared with simplified model with cache (44.7 ms)

Original code & no optimization

Optimization Level 1

```
float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int i, j, k, imax, jmax, kmax;
    float gosa, s0, ss;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    for (i = 1; i < imax; i++) {
        for (j = 1; j < jmax; j++) {
            for (k = 1; k < kmax; k++) {
                s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                     + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                     + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                     + MR(b, 0, i, j, k)
                       * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                          - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                     + MR(b, 1, i, j, k)
                       * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                          - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                     + MR(b, 2, i, j, k)
                       * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                          - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                     + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                     + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                     + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                     + MR(wrk1, 0, i, j, k);
                ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                gosa += ss * ss;
                MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
            }
        }
    }
    return (gosa);
}
```

1524 ms
(34× slower)

GCC flags: **-O1**

Optimization Level 2

```
float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int i, j, k, imax, jmax, kmax;
    float gosa, s0, ss;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    for (i = 1; i < imax; i++) {
        for (j = 1; j < jmax; j++) {
            for (k = 1; k < kmax; k++) {
                s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                     + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                     + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                     + MR(b, 0, i, j, k)
                       * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                          - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                     + MR(b, 1, i, j, k)
                       * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                          - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                     + MR(b, 2, i, j, k)
                       * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                          - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                     + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                     + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                     + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                     + MR(wrk1, 0, i, j, k);
                ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                gosa += ss * ss;
                MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
            }
        }
    }
    return (gosa);
}
```

855 ms
(19× slower)

GCC flags: **-O2**

Vectorization

```
float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int i, j, k, imax, jmax, kmax;
    float gosa, s0, ss;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    for (i = 1; i < imax; i++) {
        for (j = 1; j < jmax; j++) {
            #pragma omp simd reduction(+:gosa)
            for (k = 1; k < kmax; k++) {
                s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                     + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                     + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                     + MR(b, 0, i, j, k)
                       * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                          - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                     + MR(b, 1, i, j, k)
                       * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                          - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                     + MR(b, 2, i, j, k)
                       * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                          - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                     + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                     + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                     + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                     + MR(wrk1, 0, i, j, k);
                ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                gosa += ss * ss;
                MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
            }
        }
    }
    return (gosa);
}
```

vectorize loop and perform reduction

387 ms
(9× slower)

GCC flags: **-O2 -fopenmp**

Vectorization + march=native

```
float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int i, j, k, imax, jmax, kmax;
    float gosa, s0, ss;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    for (i = 1; i < imax; i++) {
        for (j = 1; j < jmax; j++) {
            #pragma omp simd reduction(+:gosa)
            for (k = 1; k < kmax; k++) {
                s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                     + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                     + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                     + MR(b, 0, i, j, k)
                       * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                          - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                     + MR(b, 1, i, j, k)
                       * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                          - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                     + MR(b, 2, i, j, k)
                       * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                          - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                     + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                     + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                     + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                     + MR(wrk1, 0, i, j, k);
                ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                gosa += ss * ss;
                MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
            }
        }
    }
    return (gosa);
}
```

vectorize loop and perform reduction

253 ms
(6× slower)

GCC flags: **-O2 -fopenmp -march=native**

Vectorization + march=native

```
float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int i, j, k, imax, jmax, kmax;
    float gosa, s0, ss;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    for (i = 1; i < imax; i++) {
        for (j = 1; j < jmax; j++) {
            #pragma omp simd reduction(+:gosa)
            for (k = 1; k < kmax; k++) {
                s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                     + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                     + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                     + MR(b, 0, i, j, k)
                       * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                          - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                     + MR(b, 1, i, j, k)
                       * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                          - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                     + MR(b, 2, i, j, k)
                       * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                          - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                     + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                     + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                     + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                     + MR(wrk1, 0, i, j, k);
                ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                gosa += ss * ss;
                MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
            }
        }
    }
    return (gosa);
}
```

vectorize loop and perform reduction

Uses AVX/AVX2 SIMD instructions

253 ms
(6× slower)

GCC flags: **-O2 -fopenmp -march=native**

OpenMP threads

```
#define N_THREADS 16

float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int imax, jmax, kmax;
    float gosa;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    #pragma omp parallel num_threads(N_THREADS)
    {
        int i, j, k;
        float s0, ss;

        #pragma omp for reduction(+:gosa) schedule(static,a->mrows/N_THREADS)
        for (i = 1; i < imax; i++) {
            for (j = 1; j < jmax; j++) {
                 #pragma omp simd reduction(+:gosa)
                for (k = 1; k < kmax; k++) {
                    s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                         + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                         + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                         + MR(b, 0, i, j, k)
                           * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                              - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                         + MR(b, 1, i, j, k)
                           * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                              - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                         + MR(b, 2, i, j, k)
                           * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                              - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                         + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                         + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                         + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                         + MR(wrk1, 0, i, j, k);
                    ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                    gosa += ss * ss;
                    MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
                }
            }
        }
    }
    return (gosa);
}
```

OpenMP threads

```
#define N_THREADS 16

float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int imax, jmax, kmax;
    float gosa;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    #pragma omp parallel num_threads(N_THREADS)
    {
        int i, j, k;
        float s0, ss;

        #pragma omp for reduction(+:gosa) schedule(static,a->mrows/N_THREADS)
        for (i = 1; i < imax; i++) {
            for (j = 1; j < jmax; j++) {
                 #pragma omp simd reduction(+:gosa)
                for (k = 1; k < kmax; k++) {
                    s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                         + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                         + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                         + MR(b, 0, i, j, k)
                           * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                              - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                         + MR(b, 1, i, j, k)
                           * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                              - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                         + MR(b, 2, i, j, k)
                           * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                              - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                         + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                         + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                         + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                         + MR(wrk1, 0, i, j, k);
                    ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                    gosa += ss * ss;
                    MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
                }
            }
        }
    }
    return (gosa);
}
```

vars declared outside “omp parallel” are **shared**

OpenMP threads

```
#define N_THREADS 16

float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int imax, jmax, kmax;
    float gosa;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    #pragma omp parallel num_threads(N_THREADS)
    {
        int i, j, k;
        float s0, ss;

        #pragma omp for reduction(+:gosa) schedule(static,a->mrows/N_THREADS)
        for (i = 1; i < imax; i++) {
            for (j = 1; j < jmax; j++) {
                #pragma omp simd reduction(+:gosa)
                for (k = 1; k < kmax; k++) {
                    s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                         + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                         + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                         + MR(b, 0, i, j, k)
                           * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                              - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                         + MR(b, 1, i, j, k)
                           * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                              - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                         + MR(b, 2, i, j, k)
                           * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                              - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                         + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                         + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                         + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                         + MR(wrk1, 0, i, j, k);
                    ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                    gosa += ss * ss;
                    MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
                }
            }
        }
    }
    return (gosa);
}
```

vars declared outside “omp parallel” are **shared**

vars declared inside “omp parallel” are **private**

OpenMP threads

```
#define N_THREADS 16

float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int imax, jmax, kmax;
    float gosa;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    #pragma omp parallel num_threads(N_THREADS)
    {
        int i, j, k;
        float s0, ss;

        #pragma omp for reduction(+:gosa) schedule(static,a->mrows/N_THREADS)
        for (i = 1; i < imax; i++) {
            for (j = 1; j < jmax; j++) {
                 #pragma omp simd reduction(+:gosa)
                for (k = 1; k < kmax; k++) {
                    s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                         + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                         + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                         + MR(b, 0, i, j, k)
                           * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                              - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                         + MR(b, 1, i, j, k)
                           * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                              - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                         + MR(b, 2, i, j, k)
                           * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                              - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                         + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                         + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                         + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                         + MR(wrk1, 0, i, j, k);
                    ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                    gosa += ss * ss;
                    MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
                }
            }
        }
    }
    return (gosa);
}
```

vars declared outside “omp parallel” are **shared**

vars declared inside “omp parallel” are **private**

Alternative?

Add **`private(j,k,s0,ss)`** to OpenMP pragma

OpenMP threads

```
#define N_THREADS 16

float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int imax, jmax, kmax;
    float gosa;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    #pragma omp parallel num_threads(N_THREADS)
    {
        int i, j, k;
        float s0, ss;

        #pragma omp for reduction(+:gosa) schedule(static,a->mrows/N_THREADS)
        for (i = 1; i < imax; i++) {
            for (j = 1; j < jmax; j++) {
                #pragma omp simd reduction(+:gosa)
                for (k = 1; k < kmax; k++) {
                    s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                         + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                         + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                         + MR(b, 0, i, j, k)
                           * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                              - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                         + MR(b, 1, i, j, k)
                           * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                              - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                         + MR(b, 2, i, j, k)
                           * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                              - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                         + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                         + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                         + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                         + MR(wrk1, 0, i, j, k);
                    ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                    gosa += ss * ss;
                    MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
                }
            }
        }
    }
    return (gosa);
}
```

vars declared outside “omp parallel” are **shared**

vars declared inside “omp parallel” are **private**

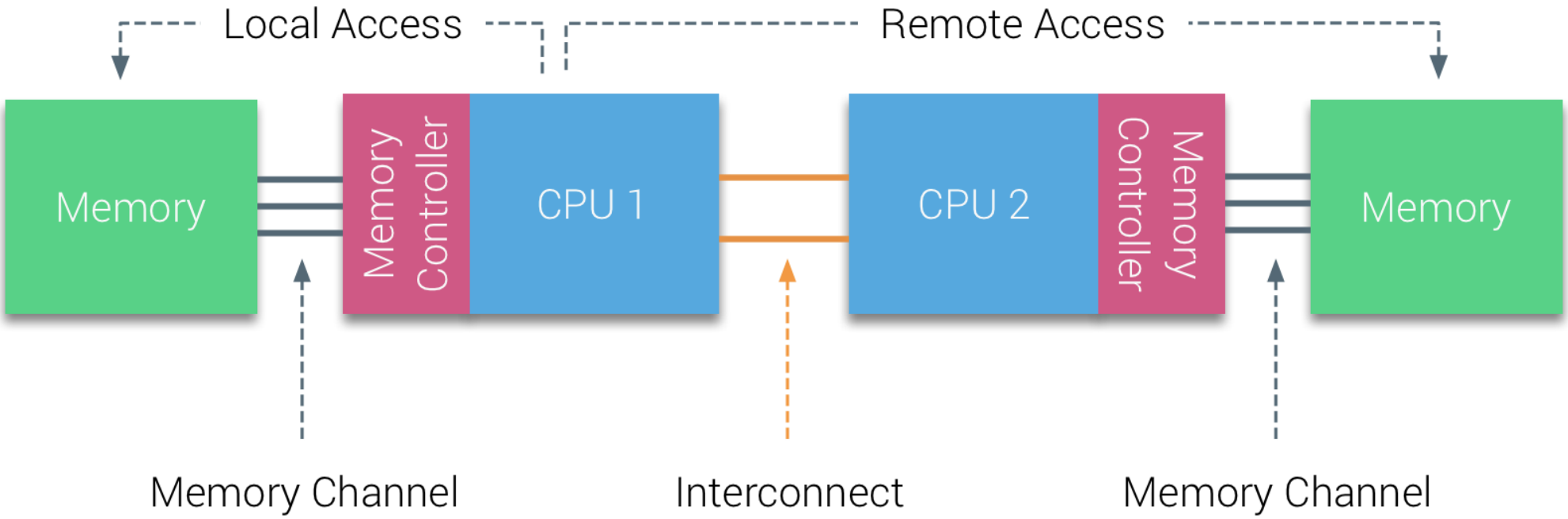
Alternative?

Add **`private(j,k,s0,ss)`** to OpenMP pragma

157 ms
(4× slower)

GCC flags: **-O2 -fopenmp -march=native**

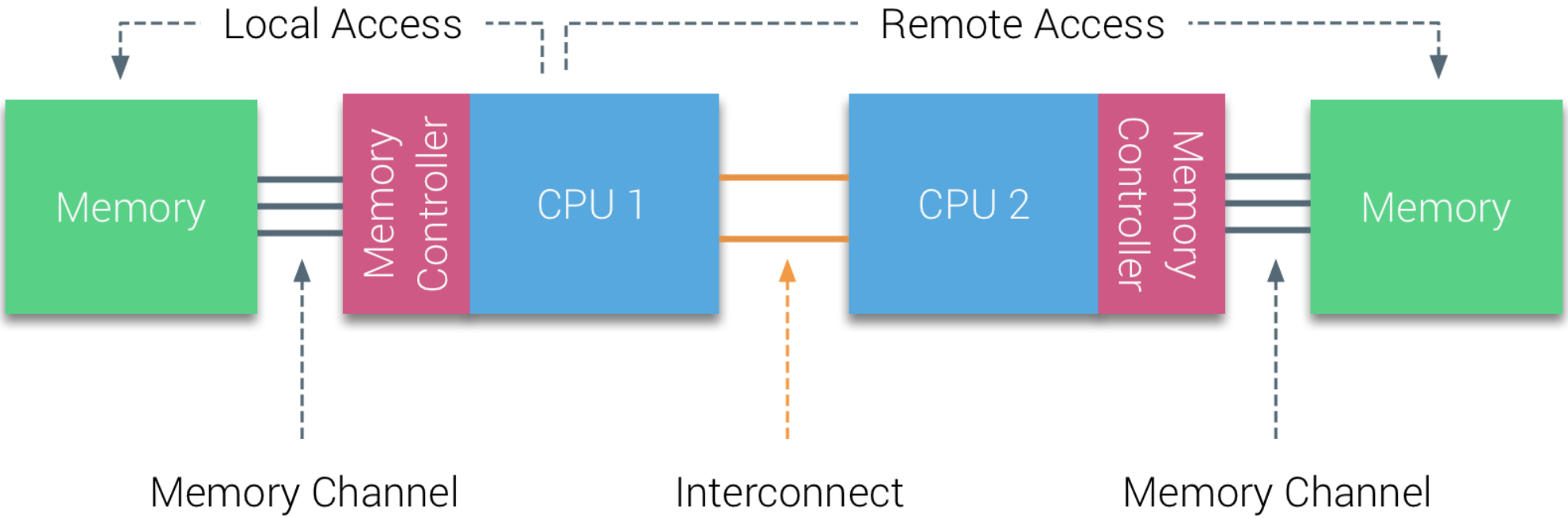
NUMA: Local vs Remote Access



Local memory latency is **1.6× faster** than cross-socket memory latency

Local memory bandwidth is **3.3× faster** than cross-socket bandwidth

NUMA: Local vs Remote Access



How to avoid low latency and bandwidth?

Initialize data on a thread executing on a socket were later the same or other thread will request access to the data

Matrix Initialization

```
/**
 * Matrix initialization - Type 1
 */
void mat_set_init(Matrix *Mat) {

    int i, j, k;

    for (i = 0; i < Mat->mrows; i++)
        for (j = 0; j < Mat->mcols; j++)
            for (k = 0; k < Mat->mdeps; k++)
                MR(Mat, 0, i, j, k) = (float) (i * i)
                                      / (float) ((Mat->mrows - 1) * (Mat->mrows - 1));
}

/**
 * Matrix initialization - Type 2
 */
void mat_set(Matrix *Mat, int l, float val) {
    int i, j, k;

    for (i = 0; i < Mat->mrows; i++)
        for (j = 0; j < Mat->mcols; j++)
            for (k = 0; k < Mat->mdeps; k++)
                MR(Mat, l, i, j, k) = val;
}
```

initializes matrix *p* (called 1 time)

initializes matrix all other matrices (called 13 times)

Matrix Initialization

```
/**
 * Matrix initialization - Type 1
 */
void mat_set_init(Matrix *Mat) {

    int i, j, k;

    for (i = 0; i < Mat->mrows; i++)
        for (j = 0; j < Mat->mcols; j++)
            for (k = 0; k < Mat->mdeps; k++)
                MR(Mat, 0, i, j, k) = (float) (i * i)
                                      / (float) ((Mat->mrows - 1) * (Mat->mrows - 1));
}

/**
 * Matrix initialization - Type 2
 */
void mat_set(Matrix *Mat, int l, float val) {
    int i, j, k;

    for (i = 0; i < Mat->mrows; i++)
        for (j = 0; j < Mat->mcols; j++)
            for (k = 0; k < Mat->mdeps; k++)
                MR(Mat, l, i, j, k) = val;
}
```

initializes matrix *p* (called 1 time)

initializes matrix all other matrices (called 13 times)

Idea?

Initialize matrix positions in the same core as they are going to be accessed

Matrix Initialization

```
/**
 * Matrix initialization - Type 1
 */
void mat_set_init(Matrix *Mat) {

    int i, j, k;

    for (i = 0; i < Mat->mrows; i++)
        for (j = 0; j < Mat->mcols; j++)
            for (k = 0; k < Mat->mdeps; k++)
                MR(Mat, 0, i, j, k) = (float) (i * i)
                                      / (float) ((Mat->mrows - 1) * (Mat->mrows - 1));
}

/**
 * Matrix initialization - Type 2
 */
void mat_set(Matrix *Mat, int l, float val) {
    int i, j, k;

    for (i = 0; i < Mat->mrows; i++)
        for (j = 0; j < Mat->mcols; j++)
            for (k = 0; k < Mat->mdeps; k++)
                MR(Mat, l, i, j, k) = val;
}
```

initializes matrix *p* (called 1 time)

initializes matrix all other matrices (called 13 times)

Idea?

Initialize matrix positions in the same core as they are going to be accessed

How?

Parallelize initialization loops in the same way that they are accessed in *jacobi()*

Matrix Initialization

```
/**
 * Matrix initialization - Type 1
 */
void mat_set(Matrix *Mat, int l, float val) {

    #pragma omp parallel num_threads(N_THREADS)
    {
        int i, j, k;

        #pragma omp for schedule(static, Mat->mrows / N_THREADS)
        for (i = 0; i < Mat->mrows; i++)
            for (j = 0; j < Mat->mcols; j++)
                for (k = 0; k < Mat->mdeps; k++)
                    MR(Mat, l, i, j, k) = val;
    }
}

/**
 * Matrix initialization - Type 2
 */
void mat_set_init(Matrix *Mat) {

    #pragma omp parallel num_threads(N_THREADS)
    {
        int i, j, k;

        #pragma omp for schedule(static, Mat->mrows / N_THREADS)
        for (i = 0; i < Mat->mrows; i++)
            for (j = 0; j < Mat->mcols; j++)
                for (k = 0; k < Mat->mdeps; k++)
                    MR(Mat, 0, i, j, k) = (float) (i * i)
                                        / (float) ((Mat->mrows - 1) * (Mat->mrows - 1));
    }
}
```

42 ms

(very close to predicted performance!)

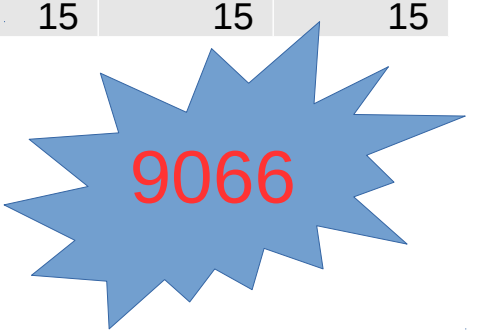
GCC flags: **-O2 -fopenmp -march=native**

Thread placement with OMP_PROC_BIND

```
/**
 * Matrix initialization - Type 2
 */
void mat_set_init(Matrix *Mat) {

    #pragma omp parallel num_threads(N_THREADS)
    {
        int inum, cpu;
        inum = omp_get_thread_num();
        cpu = sched_getcpu();               //Confirm affinity
        printf("thread %d on logical core %d\n", inum, cpu);
    }

    #pragma omp parallel num_threads(N_THREADS)
    {
        int inum, cpu;
        inum = omp_get_thread_num();
        cpu = sched_getcpu();               //Confirm affinity
        printf("thread %d on logical core %d\n", inum, cpu);
    }

    #pragma omp parallel num_threads(N_THREADS)
    {
        int i, j, k;

        #pragma omp for schedule(static, Mat->mrows / N_THREADS)
        for (i = 0; i < Mat->mrows; i++)
            for (j = 0; j < Mat->mcols; j++)
                for (k = 0; k < Mat->mdeps; k++)
                    MR(Mat, 0, i, j, k) = (float) (i * i)
                                        / (float) ((Mat->mrows - 1) * (Mat->mrows - 1));
    }
}
```

Check thread/core pairs for two subsequent *parallel* regions

Thread	logical core (parallel #1)	logical core (parallel #2)
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
10	10	10
11	11	11
12	12	12
13	13	13
14	14	14
15	15	15



standard deviation: 48

What if PROC_BIND was not set?

```
/**
 * Matrix initialization - Type 2
 */
void mat_set_init(Matrix *Mat) {

    #pragma omp parallel num_threads(N_THREADS)
    {
        int inum, cpu;
        inum = omp_get_thread_num();
        cpu = sched_getcpu();               //Confirm affinity
        printf("thread %d on logical core %d\n", inum, cpu);
    }

    #pragma omp parallel num_threads(N_THREADS)
    {
        int inum, cpu;
        inum = omp_get_thread_num();
        cpu = sched_getcpu();               //Confirm affinity
        printf("thread %d on logical core %d\n", inum, cpu);
    }

    #pragma omp parallel num_threads(N_THREADS)
    {
        int i, j, k;

        #pragma omp for schedule(static, Mat->mrows / N_THREADS)
        for (i = 0; i < Mat->mrows; i++)
            for (j = 0; j < Mat->mcols; j++)
                for (k = 0; k < Mat->mdeps; k++)
                    MR(Mat, 0, i, j, k) = (float) (i * i)
                                          / (float) ((Mat->mrows - 1) * (Mat->mrows - 1));
    }
}
```

Check thread/core pairs for two subsequent *parallel* regions

Thread	logical core (parallel #1)	logical core (parallel #2)
0	23	23
1	13	13
2	1	26
3	2	10
4	3	12
5	5	5
6	6	6
7	4	4
8	16	16
9	0	14
10	22	22
11	0	9
12	0	0
13	7	1
14	7	15
15	8	8

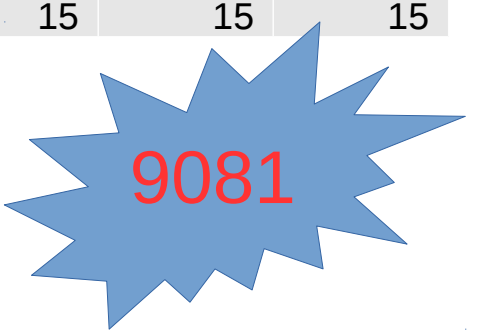
standard deviation: 851

How to manually bind threads if OMP_PROC_BIND is not set?

```
/**
 * Matrix initialization - Type 2
 */
void mat_set_init(Matrix *Mat) {

    #pragma omp parallel num_threads(N_THREADS)
    {
        int inum;
        cpu_set_t cpu_mask;                 //Allocate mask
        inum = omp_get_thread_num();
        CPU_ZERO(&cpu_mask);                //Set mask to zero
        CPU_SET(inum, &cpu_mask);           //Set mask with thread #
        sched_setaffinity( (pid_t)0, sizeof(cpu_mask), &cpu_mask); //Set the affinity
    }

    #pragma omp parallel num_threads(N_THREADS)
    {
        int inum, cpu;
        inum = omp_get_thread_num();
        cpu = sched_getcpu();               //Confirm affinity
        printf("thread %d on logical core %d\n", inum, cpu);
    }

    #pragma omp parallel num_threads(N_THREADS)
    {
        int inum, cpu;
        inum = omp_get_thread_num();
        cpu = sched_getcpu();               //Confirm affinity
        printf("thread %d on logical core %d\n", inum, cpu);
    }

    #pragma omp parallel num_threads(N_THREADS)
    {
        int i, j, k;

        #pragma omp for schedule(static, Mat->mrows / N_THREADS)
        for (i = 0; i < Mat->mrows; i++)
            for (j = 0; j < Mat->mcols; j++)
                for (k = 0; k < Mat->mdeps; k++)
                    MR(Mat, 0, i, j, k) = (float) (i * i)
                                          / (float) ((Mat->mrows - 1) * (Mat->mrows - 1));
    }
}
```

Bind threads with logical cores of same ID

Thread	logical core (parallel #1)	logical core (parallel #2)
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
10	10	10
11	11	11
12	12	12
13	13	13
14	14	14
15	15	15



standard deviation: 57

OpenMP threads + Cache

```
#define N_THREADS 16
#define BLOCK 64
float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int imax, jmax, kmax;
    float gosa;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    #pragma omp parallel num_threads(N_THREADS)
    {

        int i, j, k;
        float s0, ss;

        int j_start;
        for (j_start = 1; j_start < jmax; j_start+=BLOCK) {
            #pragma omp for reduction(+:gosa) schedule(static, a->mrows / N_THREADS)
            for (i = 1; i < imax; i++) {
                for (j = j_start; (j < jmax) && (j < (j_start + BLOCK)); j++) {
                    #pragma omp simd reduction(+:gosa)
                    for (k = 1; k < kmax; k++) {
                        s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                             + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                             + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                             + MR(b, 0, i, j, k)
                               * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                                  - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                             + MR(b, 1, i, j, k)
                               * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                                  - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                             + MR(b, 2, i, j, k)
                               * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                                  - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                             + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                             + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                             + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                             + MR(wrk1, 0, i, j, k);

                        ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                        gosa += ss * ss;
                        MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
                    }
                }
            }
        }
    }
    return (gosa);
}
```

subsequent accesses to *i* are stored in cache by accessing *j* in blocks of 64 elements instead iterating all *j* 256 elements once

OpenMP threads + Cache

```
#define N_THREADS 16
#define BLOCK 64
float jacobi(Matrix *a, Matrix *b, Matrix *c, Matrix *p, Matrix *bnd, Matrix *wrk1, Matrix *wrk2) {

    int imax, jmax, kmax;
    float gosa;
    const float omega = 0.8;
    imax = p->mrows - 1;
    jmax = p->mcols - 1;
    kmax = p->mdeps - 1;
    gosa = 0.0;

    #pragma omp parallel num_threads(N_THREADS)
    {

        int i, j, k;
        float s0, ss;

        int j_start;
        for (j_start = 1; j_start < jmax; j_start+=BLOCK) {
            #pragma omp for reduction(+:gosa) schedule(static, a->mrows / N_THREADS)
            for (i = 1; i < imax; i++) {
                for (j = j_start; (j < jmax) && (j < (j_start + BLOCK)); j++) {
                    #pragma omp simd reduction(+:gosa)
                    for (k = 1; k < kmax; k++) {
                        s0 = MR(a, 0, i, j, k) * MR(p, 0, i + 1, j, k)
                             + MR(a, 1, i, j, k) * MR(p, 0, i, j + 1, k)
                             + MR(a, 2, i, j, k) * MR(p, 0, i, j, k + 1)
                             + MR(b, 0, i, j, k)
                               * (MR(p, 0, i + 1, j + 1, k) - MR(p, 0, i + 1, j - 1, k)
                                  - MR(p, 0, i - 1, j + 1, k) + MR(p, 0, i - 1, j - 1, k))
                             + MR(b, 1, i, j, k)
                               * (MR(p, 0, i, j + 1, k + 1) - MR(p, 0, i, j - 1, k + 1)
                                  - MR(p, 0, i, j + 1, k - 1) + MR(p, 0, i, j - 1, k - 1))
                             + MR(b, 2, i, j, k)
                               * (MR(p, 0, i + 1, j, k + 1) - MR(p, 0, i - 1, j, k + 1)
                                  - MR(p, 0, i + 1, j, k - 1) + MR(p, 0, i - 1, j, k - 1))
                             + MR(c, 0, i, j, k) * MR(p, 0, i - 1, j, k)
                             + MR(c, 1, i, j, k) * MR(p, 0, i, j - 1, k)
                             + MR(c, 2, i, j, k) * MR(p, 0, i, j, k - 1)
                             + MR(wrk1, 0, i, j, k);

                        ss = (s0 * MR(a, 3, i, j, k) - MR(p, 0, i, j, k)) * MR(bnd, 0, i, j, k);
                        gosa += ss * ss;
                        MR(wrk2, 0, i, j, k) = MR(p, 0, i, j, k) + omega * ss;
                    }
                }
            }
        }
    }
    return (gosa);
}
```

subsequent accesses to *i* are stored in cache by accessing *j* in blocks of 64 elements instead iterating all *j* 256 elements once

39 ms

GCC flags: **-O2 -fopenmp -march=native**

Conclusion

90% of the “goods” for 10% of the effort

- OpenMP nested parallel regions + dynamic scheduling can be used to get more 10% of performance (~10000), but requires non-trivial code changes
- Using only OpenMP pragmas with GCC's -O2 and architecture specific code generation enabled results in 90× speedup
- If remaining 10% of the potential improvement will be worth it or not will depend on the application
  - Faster program may result in considerably less energy use in a HPC center that can typically spend thousands/millions in electricity

Kernel is very memory bandwidth dependent

- We predict that using RAM in quad-channel configuration (instead of dual-channel) will double the performance using the same OpenMP implementation