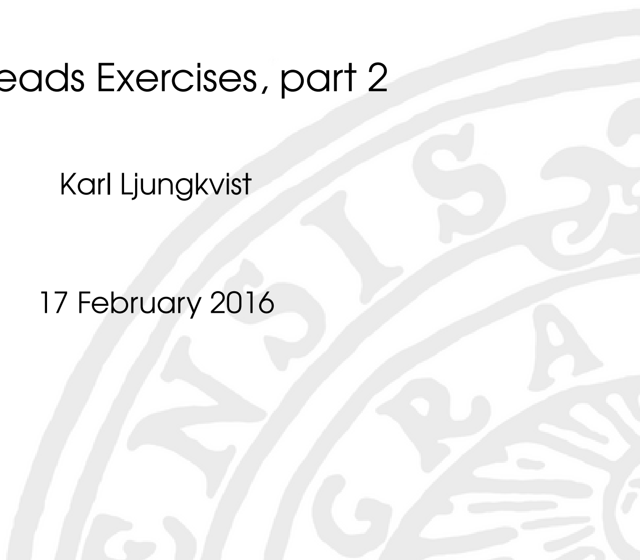
Pthreads Exercises, part 2

Karl Ljungkvist

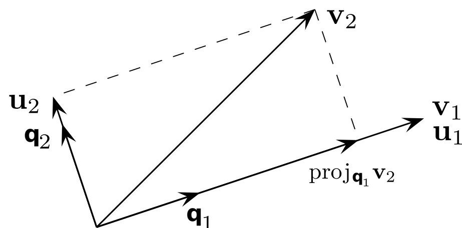
17 February 2016



Example: Gram-Schmidt process

Problem:

- Orthogonalize a set of vectors



$$\begin{aligned}
\mathbf{u}_0 &= \mathbf{v}_0 \\
\mathbf{u}_1 &= \mathbf{v}_1 - \mathrm{proj}_{\mathbf{q}_0}(\mathbf{v}_1) \\
\mathbf{u}_2 &= \mathbf{v}_2 - \mathrm{proj}_{\mathbf{q}_0}(\mathbf{v}_2) - \mathrm{proj}_{\mathbf{q}_1}(\mathbf{v}_2{}') \\
\mathbf{u}_3 &= \mathbf{v}_3 - \mathrm{proj}_{\mathbf{q}_0}(\mathbf{v}_3) - \mathrm{proj}_{\mathbf{q}_1}(\mathbf{v}_3{}') - \mathrm{proj}_{\mathbf{q}_2}(\mathbf{v}_3{}'') \\
&\vdots \\
\mathbf{u}_k &= \mathbf{v}_k - \mathrm{proj}_{\mathbf{q}_0}(\mathbf{v}_k) - \ldots - \mathrm{proj}_{\mathbf{q}_{k-1}}(\mathbf{v}_k{}^{(k-1)})
\end{aligned}$$

$$\mathbf{q}_i = \frac{\mathbf{u}_i}{||\mathbf{u}_i||}$$

Numerically unstable:

- Errors accumulate
- *Modified* Gram-Schmidt
- Iterative projection

Example: Gram-Schmidt process

Algorithm code:

```
for (i=0; i<n, i++){

  /* Normalize Q[i] */
  norm=VecNorm(V[i]);
  for (k=0; k<n; k++)
    Q[i][k]=V[i][k]/norm;

  /* Orthogonal projection */
  for (j=i+1; j<n; j++){
    s=ScalarProd(Q[i],V[j]);
    for (k=0; k<n; k++)
      V[j][k]=V[j][k]-s*Q[i][k];
  }
}
```

Discussion: Where is the parallelism?

The orthogonal projections of `Q[i]` on all `V[j]` for `j=i+1` to `n` are perfectly parallel tasks.

Example: Gram-Schmidt process

Solution:

```
for (i=0; i<n, i++){

  /* Normalize Q[i] */
  norm=VecNorm(V[i]);
  for (k=0; k<n; k++) Q[i][k]=V[i][k]/norm;

  /* Orthogonal projection */
  for(t=0; t<NUM_THREADS; t++){
    j1=i+1+(n-i-1)/NUM_THREADS*t;
    j2=i+1+(n-i-1)/NUM_THREADS*(t+1);
    pthread_create(&thread[t],&attr,proj,func_arg);
  }
  for(t=0; t<NUM_THREADS; t++)
    pthread_join(thread[t], &status);
}
```

Proj:

```
for(j=j1;j<j2;j++){
   s= scalarProd(Q[i],V[j],n);
   for(k=0;k<n;k++) V[j][k] -= s*Q[i][k];}
```

Example: Gram-Schmidt process

Performance (8 cores):

N_{thr}	T (1000)	T (2000)
1	2.13	28.5
2	1.93	21.1
3	1.92	17.1
4	2.10	16.2
5	2.46	16.7
6	2.83	16.8
7	3.15	17.0
8	3.36	18.4

Discussion:
What went wrong here?

```
for (i=0; i<n, i++){

  /* Normalize Q[i] */
  norm=VecNorm(V[i]);
  for (k=0; k<n; k++)
    Q[i][k]=V[i][k]/norm;

  /* Orthogonal projection */
  for(t=0; t<NUM_THREADS; t++){
    j1=i+1+(n-i-1)/NUM_THREADS*t;
    j2=i+1+(n-i-1)/NUM_THREADS*(t+1);
    pthread_create(&thread[t],&attr,
                   proj,func_arg);
  }
  for(t=0; t<NUM_THREADS; t++)
    pthread_join(thread[t], &status);
}
/* Proj: */
for(j=j1;j<j2;j++){
   s= scalarProd(Q[i],V[j],n);
   for(k=0;k<n;k++)
     V[j][k] -= s*Q[i][k];
}
```

Example: Gram-Schmidt process

Parallel overheads:

- Frequent creation & termination of threads $\Rightarrow$ synchronization in each iteration `i`.
- Serial section, normalization of `Q[i]` is not a part of the tasks (master computes).
- Data locality loss in projection between different iterations (`j` iterations scheduled differently between different iteratins).

Example: Gram-Schmidt process

i=0

```
Q[0]=V[0]/norm(V[0])

  s=Q[0]*V[1]
  V[1]=V[1]-s*Q[0]

  s=Q[0]*V[2]
  V[2]=V[2]-s*Q[0]

  s=Q[0]*V[3]
  V[3]=V[3]-s*Q[0]

Etc all V[j]
```

Parallel tasks

Can compute `Q[1]` here!

Let thread 0 work on vector `V[0]`,
thread 1 `V[1]`, etc
thread 2 `V[2]`
cyclicly for all vectors

No need to synchronize between iterations, check if `Q[1]` computed

i=1

```
Q[1]=V[1]/norm(V[1])

  s=Q[1]*V[2]
  V[2]=V[2]-s*Q[1]

  s=Q[1]*V[3]
  V[3]=V[3]-s*Q[1]

Etc all V[j]
```

Can compute `Q[2]` here!

Let thread 0 work on vector `V[0]`,
thread 1 `V[1]`, etc
thread 2 `V[2]`
cyclicly for all vectors

Example: Gram-Schmidt process

Solution 2:

```
int main() {

  /* Create one lock per vector */
  lock=(pthread_mutex_t *)malloc(n*sizeof(pthread_mutex_t));
  for (i=0;i<n;i++) pthread_mutex_init(&lock[i], NULL);

  /* 1:st Vector */
  Q[0]=V[0]/norm(V[0]);

  /* Start parallel algorithm */
  for (t=0; t<NUM_THREADS-1; t++)
    pthread_create(&thread[t], &attr, gram, (void *)t);

  /* Master thread join computations */
  t=NUM_THREADS-1;
  gram((void *)t);

  /* Synchronize threads, end parallel */
  for (t=0; t<NUM_THREADS-1; t++)
    pthread_join(thread[t], &status);
}
```

Example: Gram-Schmidt process

gram():

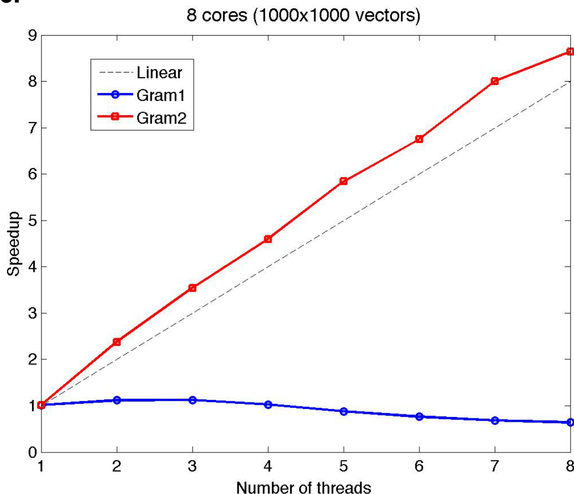
```
/* Lock all vectors and unlock first vector */
for (j=thid;j<n;j+=NUM_THREADS) pthread_mutex_lock(&lock[j]);
Barrier();
if (thrid==0) pthread_mutex_unlock(&lock[0]);

for (i=1;i<n;i++){
  /* Check if Q[i-1] is computed */
  pthread_mutex_lock(&lock[i-1]);
  pthread_mutex_unlock(&lock[i-1]);

  /* Compute projection */
  start=(i/NUM_THREADS+(i%NUM_THREADS > thid))*NUM_THREADS;

  for (j=start+thid;j<n;j+=NUM_THREADS){
    s = scalarProd(Q[i-1],V[j]);
    V[j]=V[j]-s*Q[i-1];

    if (j==i) { /* Compute Q[i] for next iteration */
      Q[i]=V[i]/norm(V[i]);
      pthread_mutex_unlock(&lock[i]);
  }}}
```

Example: Gram-Schmidt process

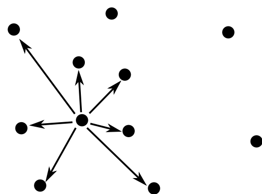
Performance results:

Note: Works for other algorithms too, e.g., LU factorization.

Note #2: Superlinear speedup – cache effect



Example: N-body simulation



- System of N pair-wise interacting particles:

$$m_i \ddot{\mathbf{x}}_i = \sum_{j=1}^{N} \mathbf{F}(\mathbf{x}_i, \mathbf{x}_j) \, , \; i = 1, \ldots, N$$

- Astrophysics, Molecular dynamics, etc.
- E.g. molecular dynamics, Lennard-Jones potential

$$\mathbf{F}(\mathbf{x}_i, \mathbf{x}_j) = \left(\frac{A}{|\mathbf{r}_{ij}|^8} - \frac{A}{|\mathbf{r}_{ij}|^{14}} \right) \mathbf{r}_{ij} \quad , \quad \mathbf{r}_{ij} = \mathbf{x}_i - \mathbf{x}_j$$

- System of ODEs – Time-stepping

Example: N-body simulation

Force calculation:

- Main part – $\mathcal{O}(N^2)$

Algorithm:

```
vec_t force[N];
vec_t pos[N];
for(i=0; i<N; ++i)
  for(j=i+1; j<N; ++j) {
    evalForce(&pos[i],&pos[j],&fx,&fy);
    force[i].x += fx; force[i].y += fy;
    force[j].x -= fx; force[j].y -= fy;
  }
```

Where is the parallelism? Are there complications?

- Can divide the particles in chunks
- Note: many force calculations contribute to a single location – must protect updates

First idea: static chunks

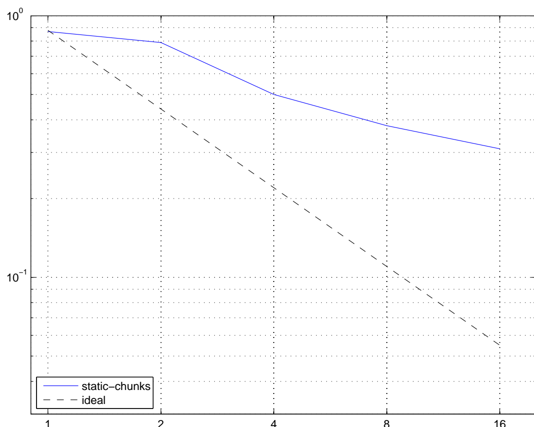
In setup:

```
for(i=0; i<N; ++i) pthread_mutex_create(&lock_particle[i]);
```

In force calculation:

```
// even chunks
int start = thrid * chunk;
int end = start + chunk;
// temporary local variables for force[i]
double tmpx,tmpy;
for(i = start; i < end; ++i) {
  tmpx=0; tmpy=0;
  for(j = i+1; j < N; ++j) {
    evalForce(&pos[i],&pos[j],&fx,&fy);
    tmpx += fx; tmpy += fy;
    pthread_mutex_lock(&lock_particle[j]);
    force[j].x -= fx; force[j].y -= fy;
    pthread_mutex_unlock(&lock_particle[j]);
  }
  pthread_mutex_lock(&lock_particle[i]);
  force[i].x += tmpx; force[i].y += tmpy;
  pthread_mutex_unlock(&lock_particle[i]);
}}
```

First idea: static chunks



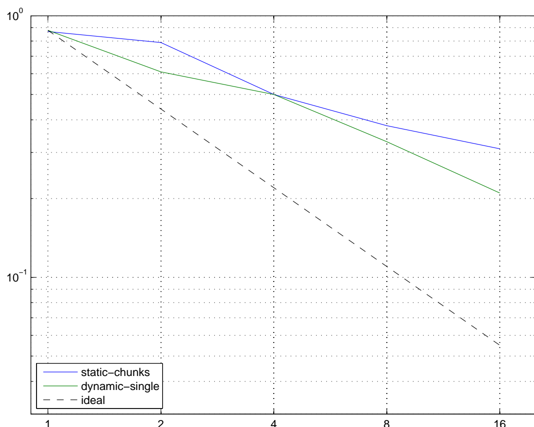
- Slow
- Uneven work load

Idea 2: dynamic single

In force calculation:

```
pthread_mutex_lock(&current); // dynamically get index
i = current_part; current_part++;
pthread_mutex_unlock(&current);

while(i < N) {
  tmpx=0;tmpy=0;
  for(j = i+1; j < N; ++j) {
    evalForce(&pos[i],&pos[j],&fx,&fy);
    tmpx += fx; tmpy += fy;
    pthread_mutex_lock(&lock_particle[j]);
    force[j].x -= fx; force[j].y -= fy;
    pthread_mutex_unlock(&lock_particle[j]);
  }
  pthread_mutex_lock(&lock_particle[i]);
  force[i].x += tmpx; force[i].y += tmpy;
  pthread_mutex_unlock(&lock_particle[i]);

  pthread_mutex_lock(&current); // dynamically get index
  i = current_part; current_part++;
  pthread_mutex_unlock(&current);
}
```

Idea 2: dynamic single



- Still slow
- Lots of locks – overhead

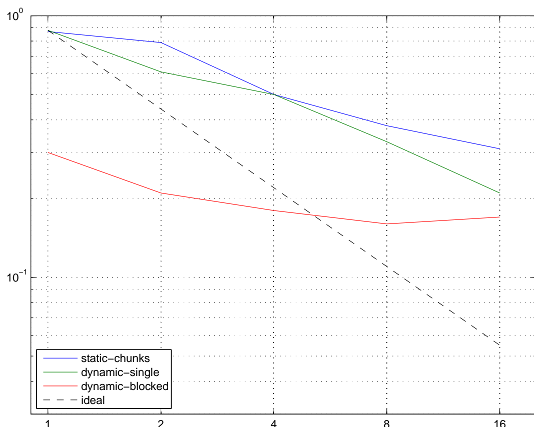
Idea 3: dynamic blocked

```
i = get_index(); // dynamically get index of *block*
while(i < NUM_BLOCKS) {
  start = i * BK_SIZE; end = start + BK_SIZE;

  /* Within block */
  pthread_mutex_lock(&lock_block[i]); // lock block
  for (j = start; j < end; ++j) {
    tmpx = 0.0; tmpy = 0.0;
    for (k = j+1; k < end; ++k) {
      evalForce(&pos[j], &pos[k], &fx, &fy);
      tmpx += fx; tmpy += fy;
      force[k].x -= fx; force[k].y -= fy;
    }
    force[j].x += tmpx; force[j].y += tmpy;
  }
  pthread_mutex_unlock(&lock_block[i]);

  /* Between blocks*/
...
```

Idea 3: dynamic blocked

```
...

  /* Between blocks*/
  for (l = i+1; l < NUM_BLOCKS; ++l) {
    start2 = l * BK_SIZE; end2 = start2 + BK_SIZE;

    for (j = start; j < end; ++j) {
      tmpx = 0.0; tmpy = 0.0;
      pthread_mutex_lock(&lock_block[l]);
      for (k = start2; k < end2; ++k) {
        evalForce(&pos[j], &pos[k], &fx, &fy);
        tmpx += fx; tmpy += fy;
        force[k].x -= fx; force[k].y -= fy;
      }
      pthread_mutex_unlock(&lock_block[l]);
      pthread_mutex_lock(&lock_block[i]);
      force[j].x += tmpx; force[j].y += tmpy;
      pthread_mutex_unlock(&lock_block[i]);
    }}

  i = get_index(); // dynamically get index of *block*
}
```

Idea 3: dynamic blocked



- Faster
- Still bad scaling

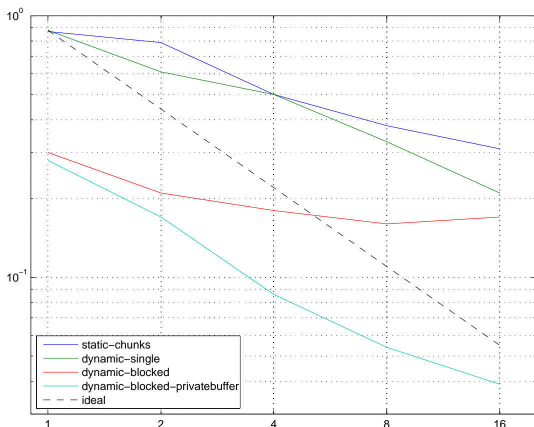
Idea 4: private buffers

```
vec_t fbuf[BK_SIZE], fbuf2[BK_SIZE]; // private buffers

i = get_index(); // dynamically get index of *block*
while(i < NUM_BLOCKS) {
  start = i * BK_SIZE; end = start + BK_SIZE;
  memset(fbuf,0,BK_SIZE*sizeof(vec_t)); // clear private buffer

  /* Within block */
  for (j = start; j < end; ++j) {
    tmpx = 0.0; tmpy = 0.0;
    for (k = j+1; k < end; ++k) {
      evalForce(&pos[j], &pos[k], &fx, &fy);
      tmpx += fx; tmpy += fy;
      fbuf[k-start].x -= fx; fbuf[k-start].y -= fy; // update private buffer
    }
    fbuf[j-start].x += tmpx; fbuf[j-start].y += tmpy; // update private buffer
  }

  pthread_mutex_lock(&lock_block[i]); // update global block
  for (j = start; j < end; ++j) {
    force[j].x += fbuf[j-start].x; force[j].y += fbuf[j-start].y; }
  pthread_mutex_unlock(&lock_block[i]);

  /* Between blocks*/
  for (l = i+1; l < NUM_BLOCKS; ++l) {
    start2 = l * BK_SIZE; end2 = start2 + BK_SIZE;
...
```

Idea 4: private buffers

```
...
  /* Between blocks*/
  for (l = i+1; l < NUM_BLOCKS; ++l) {
    start2 = l * BK_SIZE; end2 = start2 + BK_SIZE;
    memset(fbuf,0,BK_SIZE*sizeof(vec_t)); // clear private buffers
    memset(fbuf2,0,BK_SIZE*sizeof(vec_t));

    for (j = start; j < end; ++j) {
      tmpx = 0.0; tmpy = 0.0;
      for (k = start2; k < end2; ++k) {
        evalForce(&pos[j], &pos[k], &fx, &fy);
        tmpx += fx; tmpy += fy;
        fbuf2[k-start2].x -= fx; fbuf2[k-start2].y -= fy; // update private buffer
      }
      fbuf[j-start].x += tmpx; fbuf[j-start].y += tmpy; // update private buffer
    }
    pthread_mutex_lock(&lock_block[i]); // update 'our' block
    for (j = start; j < end; ++j) {
      force[j].x += fbuf[j-start].x; force[j].y += fbuf[j-start].y; }
    pthread_mutex_unlock(&lock_block[i]);

    pthread_mutex_lock(&lock_block[l]);  // update 'other' block
    for (k = start2; k < end2; ++k) {
      force[k].x += fbuf2[k-start2].x; force[k].y += fbuf2[k-start2].y; }
    pthread_mutex_unlock(&lock_block[l]);
    }
  i = get_index(); // dynamically get index of *block*
}
```

Idea 4: private buffers



- Few and small critical sections
- Very fast and scales well

Idea 5: Repeat work

Idea:

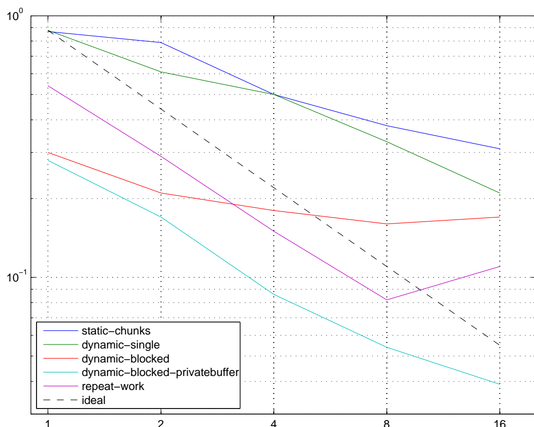
- Let each particle calculate only the force on itself

Code:

```
for(i = start; i < end; ++i) {
  tmpx=0;tmpy=0;
  for(j = 0; j < N; ++j)
  if(i != j) {
    evalForce(&pos[i],&pos[j],&fx,&fy);
    tmpx += fx;
    tmpy += fy;
  }
  force[i].x = tmpx;
  force[i].y = tmpy;
}
```

- No conflicting updates!
- Even load.

Idea 5: Repeat work



- Good scaling
- Some overhead
- Might be good on some systems

Summary:

- Balance load
- Avoid many locks / synchronization

Conclusion

To get good performance on multicores using Pthreads:

- Find and assign large tasks for the threads
- Avoid frequent synchronization
- Keem good cache locality
- Keep good load balance, e.g., let master participate as peer

Conclusion

Hardware to run on:

- Develop and debug on your computer, use GCC
- Run on IT-servers:
  `{geijer,berling,celsius,linne,...}.it.uu.se`
  2 quad core ⇒ 8 cores shared memory
- Linux servers: gullviva, tussilago, vitsippa
  16 cores shared memory
  Log on from a SunRay: `$ xrlogin gullviva`
  Log on remotely: first SSH into Solaris servers (above)
  then do: `$ rlogin gullviva`
- UPPMAX Systems:
  - Tintin, 16 core nodes
  - Halvan, one 64 core node (2048 GB RAM)