Multi-Core Programming with Pthreads

Jarmo Rantakokko

What is a thread?

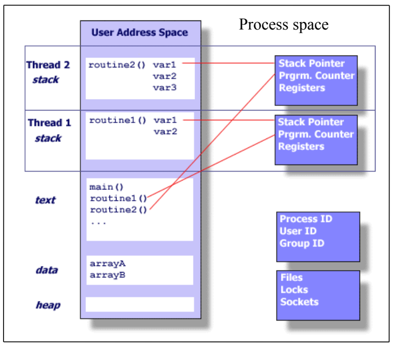
"Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system."

- Exists within a process and uses the process resources
- Has its own independent flow of control as long as its parent process exists and the OS supports it
- Duplicates only the essential resources it needs to be independently schedulable, e.g., program counter
- May share the process resources with other threads that act equally independently, e.g., shared memory
- Dies if the parent process dies - or something similar
- Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.

A thread is a "lightweight process", it contains it own:
- Program counter
- Registers and stack pointer
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals

Compare with a ***Unix process***:
- Process ID, process group ID, user ID, and group ID
- Environment
- Working directory
- Program instructions
- Registers, Stack, Heap
- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools

fork()
pthread_create() } 10:1



Two threads in a process space

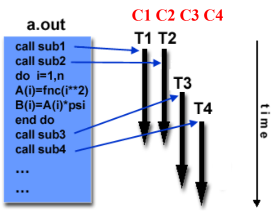
Multiple threads sharing process resources =>

- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
- Two pointers having the same value point to the same data.
- Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

Traditionally threads have been used to:

- Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
- Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks
- Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.

On **Multi-Core processors** we can use the parallel cores to run several processes/applications in parallel or we can parallelize on application using threads and schedule the threads to different cores



Several common models for threaded programs exist:

- ***Manager/worker:*** a single thread, the manager assigns work to other threads, the workers. Typically used when we have a dynamic pool of tasks with irregular work load.

- ***Peer:*** similar to the manager/worker model, but after the main thread creates other threads, it participates in the work. Typically used for static homogeneous tasks.

- ***Pipeline:*** a task is broken into a series of sub operations, each of which is handled in series, but concurrently, by a different thread. An automobile assembly line best describes this model.

POSIX threads or pthreads

*Portable **O**perating **S**ystem **I**nterface for UNI**X***

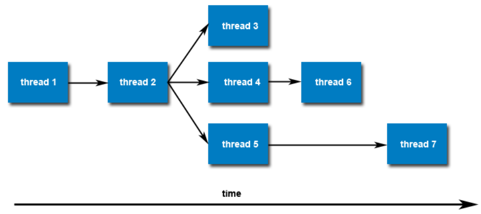
Portable standard for thread programming, specified by the IEEE POSIX 1003.1c standard (1995). C Language only!

The Pthreads API contains over 60 subroutines which can be grouped into three major classes:

- ***Thread management***: creating, terminating, joining
- ***Mutexes***: provides exclusive access to code segments and variables with the use of locks (mutual exclusion)
- ***Condition variables***: provides synchronization and communication between threads that share a mutex

Creating and terminating threads:

Pthread_create(threadptr, threadattr, func, funcarg)
Creates a thread which starts running the specified function
Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.



There are several ways in which a Pthread may be terminated:

- The thread returns from its starting routine
- The thread makes a call to *pthread_exit()*
- The thread is canceled by another thread via the *pthread_cancel()* routine
- The entire process is terminated, i.e., main() finishes without self calling *pthread_exit()*

Note: By calling *pthread_exit()* also in main(), i.e., on the *master thread*, all threads are kept alive even though all of the code in main() has been executed.
Can also do explicit wait with *pthread_join()*

Example HelloWorld:

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *HelloWorld(void *arg){
    printf("Hello World!\n");
    pthread_exit(NULL);}

int main (int argc, char *argv[]){
   pthread_t threads[NUM_THREADS];
   int t;
   for(t=0; t<NUM_THREADS; t++)
      pthread_create(&threads[t], NULL, HelloWorld, NULL);
   pthread_exit(NULL);}
```

Passing arguments:

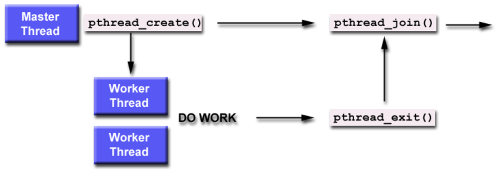
Note, can only pass one argument of type void* (see hello1)
Use structs and type cast to void*

```
struct thread_data{
       int field1;
       double field2};

void *HelloWorld(void *arg){
  struct thread_data *my_data = (struct thread_data*) arg;
  int f1 = my_data -> field1;
  double f2 = mydata -> field2;
  … }

int main (int argc, char *argv[]){
…
struct thread_data data
data.field1=5; data.field2=3.14;
pthread_create(&threads[t], NULL, HelloWorld, (void*)&data);
…
```

See example hello_arg2.c

Joining threads (waiting):

Pthread_join(threadptr, status)
Blocks the calling thread until the specified thread terminates.



When a thread is created, its attribute must define joinable

To explicitly create a thread as joinable:

- Declare a pthread attribute variable of the `pthread_attr_t` data type
- Initialize the attribute variable with `pthread_attr_init()`
- Set the attribute detached status with `pthread_attr_setdetachstate()`
- When done, free library resources used by the attribute with `pthread_attr_destroy()`

Example join: (join.c)

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

for (t=0; t<NUM_THREADS; t++)
   pthread_create(&thread[t],&attr,func,(void *)&data);

pthread_attr_destroy(&attr);

for (t=0; t<NUM_THREADS; t++)
    pthread_join(thread[t], &status);
```

Can also set the state to `PTHREAD_CREATE_DETACHED`
(Default value is joinable.)

Other attributes that can be set are stacksize and scheduling policy. (For more info see Pthreads manual.)

Global and local data:

Data allocated on the stack, i.e., within functions, is local and private to the threads. All other data is global.

```
// Global data accessible to all threads
int GlobData[Nsize];

void *threadfunc(void *arg){
// Local data private to the calling thread
int LocData[Nsize];
. . .
}

int main(int argc, char *argv){
// Global data but needs to be passed to threads
int GlobData2[Nsize];
. . .
}
```

See data.c

Mutex (*mutual exclusion)* variables:

Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.

Thread 1	Thread 2	Balance
Read: 2000 ←		2000
	Read: 2000 ←	
Withdraw: 1500 →		500
	Deposit: 1000 →	3000
Read: 3000?? ←		
	Read: 3000!! ←	

Example *without protection* of the shared Balance

A typical sequence in the use of a mutex is as follows:

- Create and initialize a mutex variable
- Several threads attempt to lock the mutex
- Only one succeeds and that thread owns the mutex
- The owner thread performs some set of actions
- The owner unlocks the mutex
- Another thread acquires the mutex and repeats the process
- Finally the mutex is destroyed

When several threads compete for a mutex, the losers block at that call - an unblocking call is available with "trylock" instead of the "lock" call. (Trylock is much faster, it does not block but it also does not have to deal with queues of multiple threads waiting on the lock.)

Mutex functions:

pthread_mutex_init(mutex, attr)
phtread_mutex_lock(mutex)
pthread_mutex_trylock(mutex)
pthread_mutex_unlock(mutex)
pthread_mutex_destroy(mutex)

The mutex attribute can be set to:

- PTHREAD_MUTEX_NORMAL_NP
- PTHREAD_MUTEX_RECURSIVE_NP
- PTHREAD_MUTEX_ERRORCHECK_NP

Or just use attr=NULL for default values.

Example Mutex: (mutex.c)

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5
pthread_mutex_t mutexsum;
int sum=0;

void *addone(void *arg){
    pthread_mutex_lock (&mutexsum);
    sum += 1;
    pthread_mutex_unlock (&mutexsum);
    pthread_exit(NULL);}

int main (int argc, char *argv[]){
   …
   pthread_mutex_init(&mutexsum, NULL);
   for(t=0; t<NUM_THREADS; t++)
      pthread_create(&threads[t], NULL, addone, NULL);
   for (t=0; t<NUM_THREADS; t++)
      pthread_join(thread[t], &status);
   printf("Sum = %d\n",sum);
```


Condition variables:

A condition variable is used for synchronization of threads. It allows a thread to block (sleep) until a specified condition is reached.

Pthread_cond_init(cond, attr)	- use attr=NULL
Pthread_cond_wait(cond, mutex)	- block thread
Pthread_cond_signal(cond)	- wake one thread
Pthread_cond_broadcast(cond)	- wake all threads
Pthread_cond_destroy(cond)	

A condition variable is always used in conjunction with a mutex lock. Proper locking and unlocking of the associated mutex variable is important.

Pthread_cond_wait():

```
pthread_mutex_lock(mutexvar);

If (status!="final")
    pthread_cond_wait(condvar,mutexvar);

pthread_mutex_unlock(mutexvar);
```

Pthread_cond_wait blocks a thread until the condition variable is signaled. It will automatically release the mutex while it waits. After the thread is awakened, mutex will be automatically locked for use by the thread. *Note*, wait does not use any CPU cycles until it is woken up (mutex_lock uses CPU cycles for polling)

Pthread_cond_signal(), pthread_cond_broadcast():

```
pthread_mutex_lock(mutexvar);

If (status=="final")
    pthread_cond_signal(condvar);

pthread_mutex_unlock(mutexvar);
```

The pthread_cond_signal() routine is used to wake up another thread which is waiting on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for pthread_cond_wait() routine to complete.

If more than one thread is in a blocking wait can then use pthread_cond_broadcast() to wake all.

Example: barrier

```
pthread_mutex_t lock;
pthread_cond_t signal;
int waiting=0, state=0;

void barrier(){
    int mystate;
    pthread_mutex_lock (&lock);
    mystate=state;
    waiting++;
    if (waiting==nthreads){
       waiting=0; state=1-mystate;
       pthread_cond_broadcast(&signal);}
    while (mystate==state)
       pthread_cond_wait(&signal,&lock);
    pthread_mutex_unlock (&lock);
}
```

Note: use while-statement as spurious wake ups of threads sleeping in wait may occur.

```
pthread_barrier_t barr;
pthread_barrier_init(&barr, null,nthreads)
pthread_barrier_wait(&barr);
```


Example: Enumeration sort

```
for (j=0;j<len;j++)
{
  rank=0;
  for (i=0;i<len;i++)
    if (indata[i]<indata[j]) rank++;
  outdata[rank]=indata[j];
}
```

Where is the parallelism? Identify parallel tasks!

For each element (j) check how many other elements (i) are smaller than it => rank
Perfectly parallel tasks for each element (j)

Solution 1: (enumeration_1.c *Manager-Worker*)

For each task (element) start a new thread, but start only a set of threads at a time. (What is the optimal number ?)

```
for (j=0;j<len;j+=NUM_THREADS){   /* Manager */

  for(t=0; t<NUM_THREADS; t++){
   el=j+t;
   pthread_create(&threads[t],&attr,findrank,(void*)el);}

  for(t=0; t<NUM_THREADS; t++)
      pthread_join(threads[t], &status);
}
```

```
void *findrank(void *arg){       /* Worker */

   int rank=0,i;long j=(long)arg;

   for (i=0;i<len;i++)
        if (indata[i]<indata[j]) rank++;

   outdata[rank]=indata[j];
   pthread_exit(NULL);}
```


Solution 1:

- Little work per task
- High overhead in creating and terminating threads
- More threads gives less synchronization points but more overhead in swapping threads in and out of cores

Solution 2:

Define larger tasks, let each task be to count the rank of *len/nthreads* elements => only one task per thread and totally nthreads tasks. Minimal synchronization and thread management overheads.

Solution 2: (enumeration_2.c *Peer workers*)

```
for (t=0; t<NUM_THREADS-1; t++)
{
  index[t].j1=t*len/NUM_THREADS;
  index[t].j2=(t+1)*len/NUM_THREADS;
  pthread_create(&threads[t], &attr, findrank,
                (void *)&index[t]);
}
findrank((void *)&index[NUM_THREADS-1])

for(t=0; t<NUM_THREADS-1; t++)
   pthread_join(threads[t], &status);
```

```
 void *findrank(void *arg){

   …
   for (j=j1;j<j2;j++){
     rank=0;
     for (i=0;i<len;i++)
       if (indata[i]<indata[j]) rank++;
     outdata[rank]=indata[j];}
```


Example: Numerical PDE Solver

$$u_t + u_x + u_y = F(t,x,y) \quad 0 \le x \le 1, 0 \le y \le 1$$

$$\begin{cases} u(t,0,y) = h_1(t,y) & 0 \le y \le 1 \\ u(t,x,0) = h_2(t,x) & 0 \le x \le 1 \end{cases} \quad \textit{Boundary Conditions}$$

$$u(0,x,y) = g(x,y) \quad \textit{Initial Conditions}$$

Solve with explicit Finite Difference Method (*Leapfrog*).

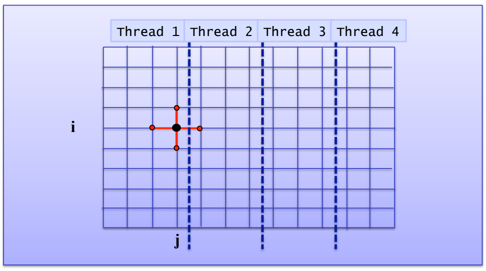
Core of the computations:

```
for k=2,Nt
   t=k*dt; Uold=U; U=Unew;
   for j=1,Ny-1
      for i=1,Nx-1
        x=i/Nx; y=j/Ny
        Unew(i,j)=Uold(i,j)+2*dt*(F(t,x,y)-
                (U(i+1,j)-U(i-1,j))/(2*dx)-
                (U(i,j+1)-U(i,j-1))/(2*dy))
      end for
   end for
end for
```

Where is the parallelism?

Update of each element (`Unew(i,j)`) is perfectly parallel within the k-loop.

Computational Stencil:



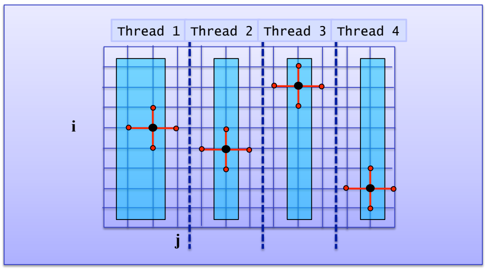
Divide grid over the threads, parallelize over j.

Parallel thread tasks: (leapfrog.c)

```
for k=2,Nt
   thread_barrier();
   t=k*dt; Uold=U; U=Unew;
   for j=j1,j2
      for i=1,Nx-1
        x=i/Nx; y=j/Ny
        Unew(i,j)=Uold(i,j)+2*dt*(F(t,x,y)-
                (U(i+1,j)-U(i-1,j))/(2*dx)-
                (U(i,j+1)-U(i,j-1))/(2*dy))
      end for
   end for
end for
```

=> Perfectly parallel computations but need to synchronize in each time step (k-iteration).

Note: No need to have a *barrier*, just make sure that all threads are working with the same time step (iteration k). The inner points do not depend on other threads data, start computing on these points.



After computing on inner points check if all threads have reach the same time step , i.e., started to compute on its inner points.

```
for k=2,Nt
   thread_barrier_start(); // thread starts a new step
   t=k*dt; Uold=U; U=Unew;
   for j=j1+1,j2-1
      for i=1,Nx-1
        x=i/Nx; y=j/Ny
        Unew(i,j)=Uold(i,j)+2*dt*(F(t,x,y)-
                (U(i+1,j)-U(i-1,j))/(2*dx)-
                (U(i,j+1)-U(i,j-1))/(2*dy))
      end for
   end for
   thread_barrier_end(); // wait until all threads have
                            called the start-routine
   update Unew(:,j1) and Unew(:,j2)
end for
```

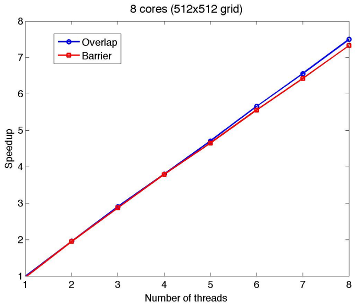

Thread_barrier_start():

```
pthread_mutex_lock(&lock);
ready++;
locstep++;
if (ready==nthreads){
    ready=0;
    step++;
    pthread_cond_broadcast(&signal);}
pthread_mutex_unlock(&lock);
```

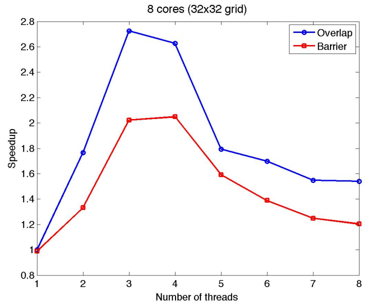
Thread_barrier_end():

```
pthread_mutex_lock(&lock);
while (locstep>step)
       pthread_cond_wait(&signal,&lock);
 pthread_mutex_unlock(&lock);
```

Performance 2x quad core processors:



Small grid: Synchronization overhead becomes significant



Remark: Reduce thread swapping by letting master thread be peer in computations. This can have large impact when computational work (thread task) is small, avoid thread re-scheduling if the number of threads match the number of cores. Performance results also becomes less random without the *extra thread*.

```
int main(int argc, char **argv){
…
for (i=0; i<nthreads-1; i++)
 pthread_create(&thread[i],&attr,leapfrog,(void*)&arg[i]);

leapfrog((void*)&arg[nthreads-1]);

for (i=0; i<nthreads-1; i++)
 pthread_join(thread[i],NULL);
```


Example: Gram-Schmidt orthogonalization

```
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];
     }
}
```

Where is the parallelism?

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

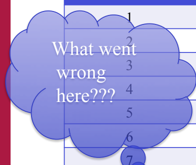
Solution:

```
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];}
```


Performance results (8 cores):

Number of threads	Time (1000)	Time (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
[illegible]	[illegible]	18.4



Parallel overheads:

- Frequent creation & termination of threads => 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 iterations).


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

        s=Q[0]*V[1]
        V[1]=V[1]-s*Q[0]
Parallel
tasks   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]
- - - - - - - - - - - - - - - - -
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[1] here!

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

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

Can compute Q[2] here!

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

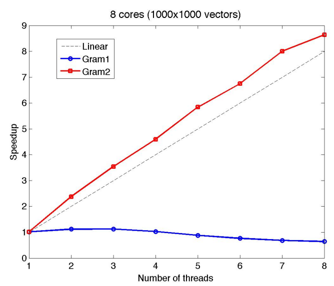
Solution 2:

```
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);
```



```
Gram:

/* Lock all vectors and unlock first vector */
for (j=thrid;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)*NUM_THREADS;
   for (j=start+thrid;j<n;j+=NUM_THREADS){
       s = scalarProd(Q[i-1],V[j]);
       V[j]=V[j]-s*Q[i-1];

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


Performance results:



Note: The same technique can be used in other algorithms as well, e.g., LU-factorization.

Summary:

Thread programming provide

- *Software portability*, code runs unmodified on serial and parallel machines (shared memory).
- *Latency hiding*, can overlap threads waiting for memory, I/O, or communication with other tasks.
- *Scheduling and load balancing*, specify concurrent tasks dynamically and use system level mapping of tasks to cores. (Irregular load, e.g., games, web server)
- *Easy of programming*, compared to local name space models using message passing (MPI).
- *Effiency,* have detailed control of threads and data.

Summary:

To get good performance on Multi-core using Pthreads

• Find and assign large tasks for the threads
• Avoid frequent synchronization of threads
• Keep good cache locality on threads
• Keep good load balance between threads.
  E.g., let master participate as peer.

• Other: Number of threads, thread attributes… ?

Hardware to run on:

• Develop and debug on your computer, use gcc
• Run on IT-servers (geijer, berling celsius, linne etc)
  2 quad core => 8 cores shared memory
• Log on to gullviva (xrlogin gullviva) from a SunRay
  2*16 core => 32 cores shared memory

UPPMAX Systems:
• Kalkyl, 8 core nodes
• Tintin, 16 core nodes
• Halvan, one 64 core node (2048 GB RAM)