FUTURE PROGRAMMING LANGUAGES

STEFANO MARKIDIS

KTH ROYAL INSTITUTE OF TECHNOLOGY, STOCKHOLM

August 29 2014 – PDC Summer School

OUTLINE

- **Future Supercomputers and the Exascale Era**
- **Motivation for new Programming Approaches:**
  - Programming Models for Exascale Hardware
    - GPU and the MPI + X problem
  - Productivity (and still High Performance)
    - PGAS for Distributed Memory Machines
    - OpenACC and OpenMP for accelerators
  - Dynamic Load balancing
    - Task-based approaches
- **Conclusions**

FUTURE PROGRAMMING MODELS

HPC programming models are tailored to a given hardware/ architecture. The goal of programming models is to exploit efficiently a given HW without the application developers focusing on lower level details.

Future programming models need to address modern and future trends in supercomputer architectures.

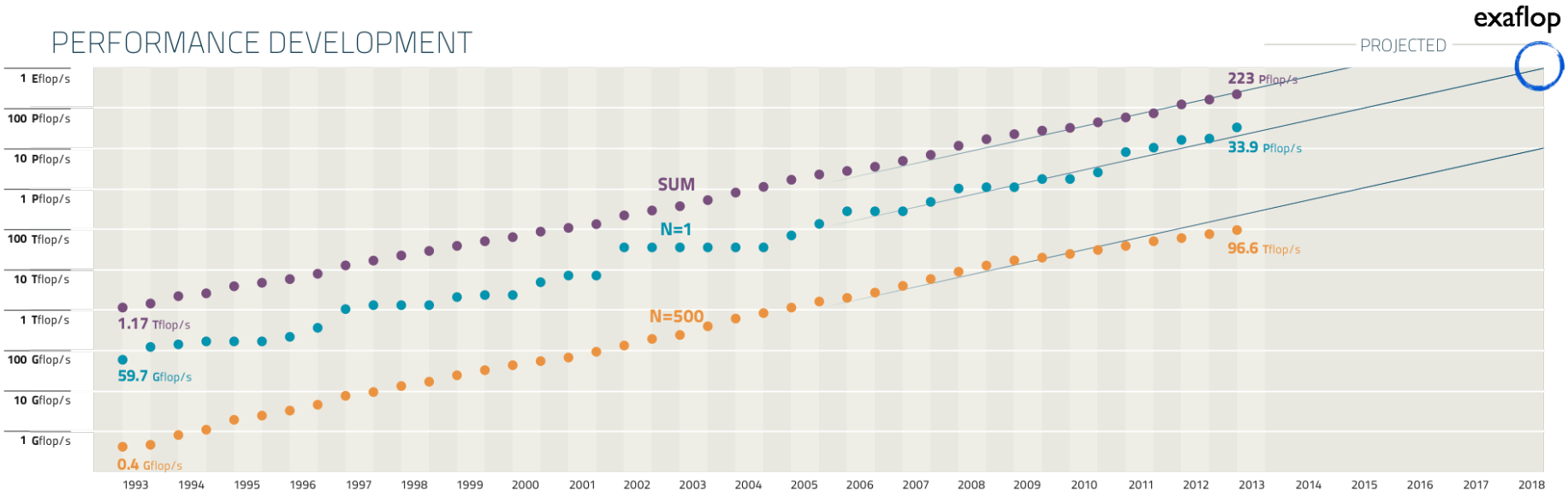
How the next generation supercomputer will be and what programming models will be effective on that machine?

WHERE WE ARE TODAY

Rank	Site	Computer	Country	Cores	Rmax [Pflops]	% of Peak	Power [MW]	MFlops /Watt
1	National Super Computer Center in Guangzhou	Tianhe-2 NUDT, Xeon 12C 2.2GHz + IntelXeon Phi (57c) + Custom	China	3,120,000	33.9	62	17.8	1905
2	DOE / OS Oak Ridge Nat Lab	Titan, Cray XK7 (16C) + Nvidia Kepler GPU (14c) + Custom	USA	560,640	17.6	65	8.3	2120
3	DOE / NNSA L Livermore Nat Lab	Sequoia, BlueGene/Q (16c) + custom	USA	1,572,864	17.2	85	7.9	2063
4	RIKEN Advanced Inst for Comp Sci	K computer Fujitsu SPARC64 VIIIfx (8c) + Custom	Japan	705,024	10.5	93	12.7	827
5	DOE / OS Argonne Nat Lab	Mira, BlueGene/Q (16c) + Custom	USA	786,432	8.16	85	3.95	2066
6	Swiss CSCS	Piz Daint, Cray XC30, Xeon 8C + Nvidia Kepler (14c) + Custom	Swiss	115,984	6.27	81	2.3	2726
7	Texas Advanced Computing Center	Stampede, Dell Intel (8c) + Intel Xeon Phi (61c) + IB	USA	204,900	5.17	61	4.5	1489
8	Forschungszentrum Juelich (FZJ)	JuQUEEN, BlueGene/Q, Power BQC 16C 1.6GHz+Custom	Germany	458,752	5.01	85	2.30	2178
9	DOE / NNSA L Livermore Nat Lab	Vulcan, BlueGene/Q, Power BQC 16C 1.6GHz+Custom	USA	393,216	4.29	85	1.97	2177
10	Government	Cray XC30, Xeon E5 12C 2.7GHz, Custom	USA	225,984	3.14	64		

From top500.org

WHEN EXASCALE ?



From top500.org

First exascale machine arriving in the time range 2018-2020

DARPA STUDY IDENTIFIES 3 EXASCALE CHALLENGES

ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems

Peter Kogge, Editor & Study Lead
Keren Bergman
Shekhar Borkar
Dan Campbell
William Carlson
William Dally
Monty Denneau
Paul Franzon
William Harrod
Kerry Hill
Jon Hiller
Sherman Karp
Stephen Keckler
Dean Klein
Robert Lucas
Mark Richards
Al Scarpelli
Steven Scott
Allan Snavely
Thomas Sterling
R. Stanley Williams
Katherine Yelick

September 28, 2008

This work was sponsored by DARPA IPTO in the ExaScale Computing Study with Dr. William Harrod as Program Manager; AFRL contract number FA8650-07-C-7724. This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings

NOTICE

Report published September 2008.

They concluded the three key challenges were:

- Energy and power consumption.
- Memory and storage.
- Fault-tolerance.

Most important issue is **power consumption**. Linear extrapolation of current architectures indicates over 500 MW for 1 exaFLOP.

Available at http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf

THE EXASCALE MACHINE

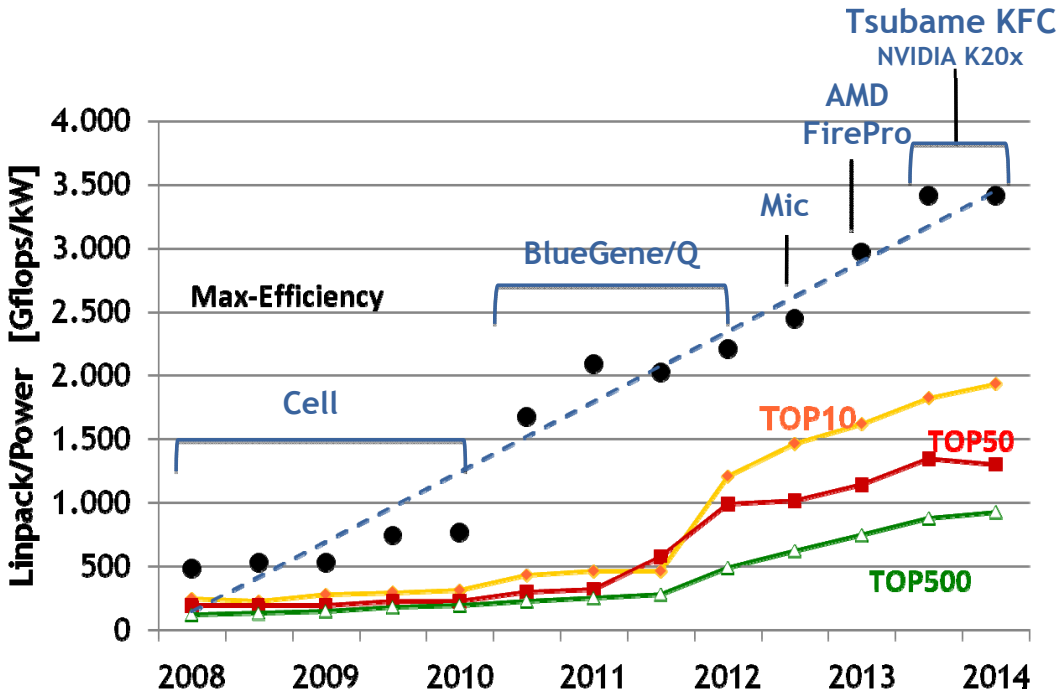
System	KTH Lindgren (# 131)	Tianhe-2 (#1)	Exascale Machine (Estimate)	Difference Tianhe-2 and Exascale Machine
System peak	0.237 PFlops	33.87 PFlops	1 **EFlops**	29.5
Power	0.640 MW	17.8 MW	**goal 20-40 MW**	/
Node concurrency	24	148	**1000-10000**	7-67
Total concurrency	36,384	3,120,000	**10^9**	320
Total memory	0.047 PB	1 PB	**32-64 PB**	32/64
Failure rate	weeks	4 days	**1 hour**	/

THE POWER ISSUE FOR EXASCALE

The only certainty about Exascale machine is that power is the main design constraint: there will be a 20 MW cap (Thianhe-2 is already 17.8 MW).

At the moment, **accelerators and supercomputer with accelerators** are the most power efficient.

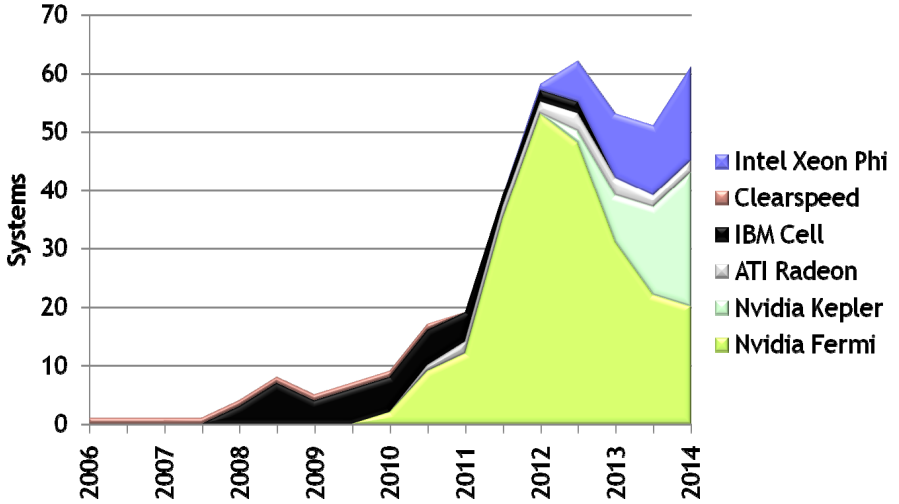
Check green500 list.



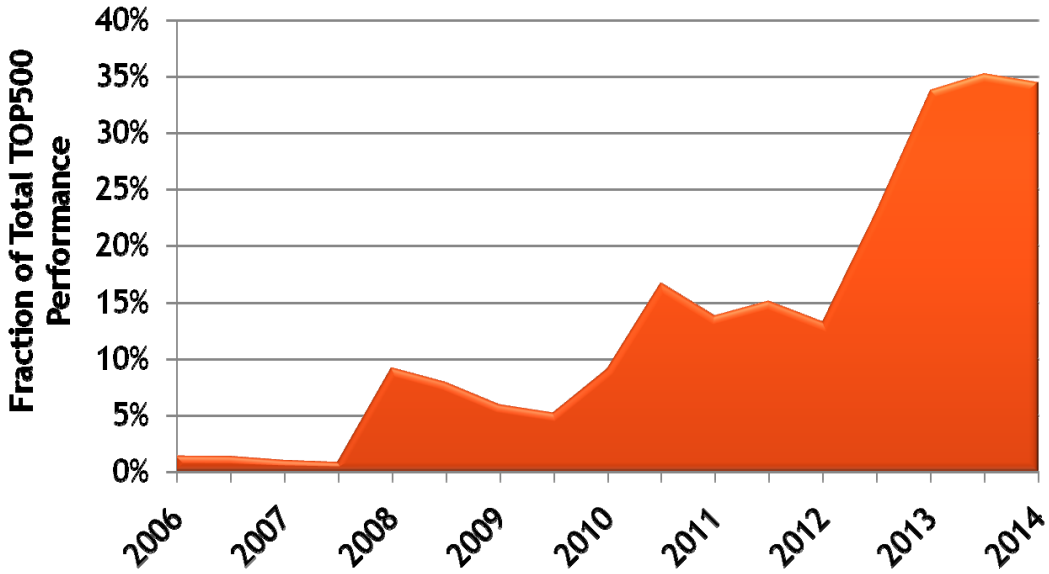
From gren500.org

ACCELERATORS IN HPC

12.2% of Top500 machines use accelerators:. 4 supercomputers in the top 10 use accelerators.



Accelerators produces 35% of the total computer power in the top 500 supercomputers



From top500.org

PROGRAMMING MODELS FOR ACCELERATORS

- CUDA targets NVDIA GPU
- OpenCL targets all the accelerators.
- OpenACC targets only NVDIA GPU
- OpenMP 4.0 targets all the accelerators. Accelerators not supported yet by compilers.
- MPIVACH supports now Message Passing from GPU memory to GPU memory.

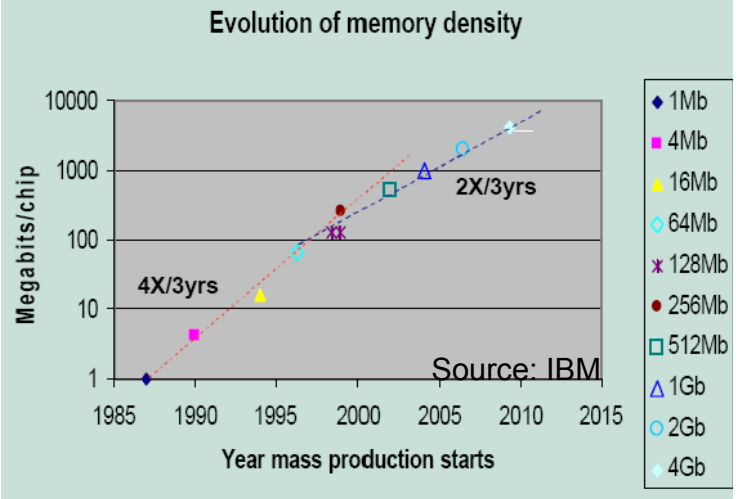
C extension

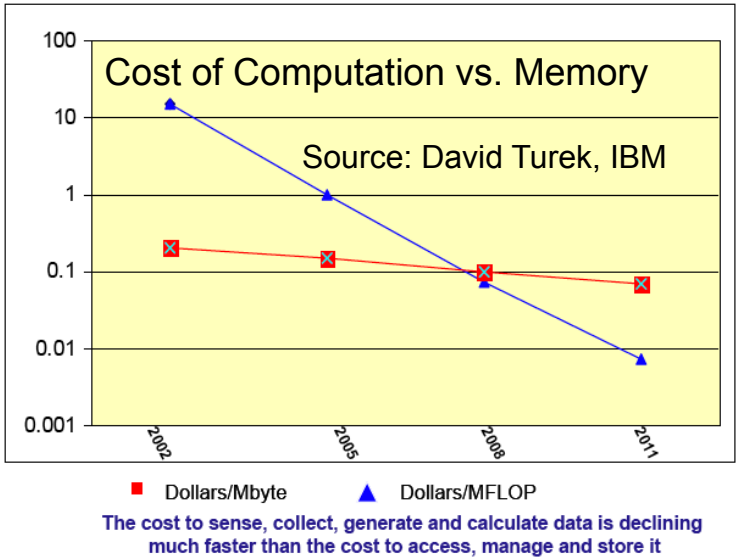
COMPILER DIRECTIVES

MPI Library

THE MEMORY ISSUE

- We know that memory per core will be less as a technological (and economical) trend: memory density 2x every 3 years, processor logic 2x every 2 years
- To move data becomes more important (expensive €) than to compute.





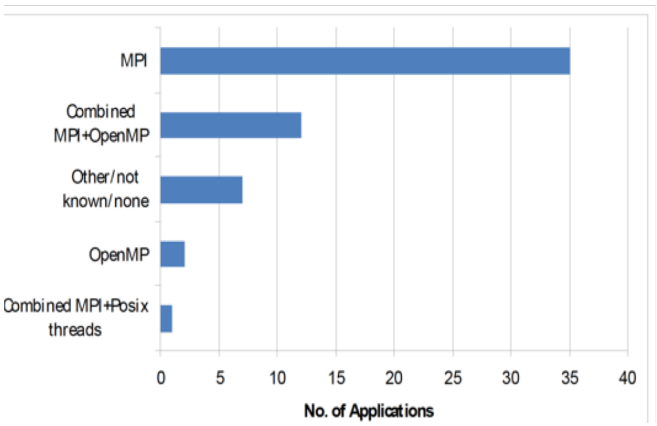
MEMORY ISSUE AND THE MPI + X

MPI will be likely at exascale (vast majority of applications use it).

MPI memory consumption on single node is an important issue (and will be even more serious at exascale). Use MPI for inter-node communication and use X for intra-node parallelism

How we deal with intra-node parallelism? (less memory per core)

- + OpenMP (performance issues)
- + PGAS (see later in the lecture)
- + MPI. MPI 3.0 and 3.1 provide shared memory mechanisms (MPI Win allocate shared and MPI endpoints).



From a PRACE Survey about 57 applications

FAILURES AND FAULT TOLERANCE

At exascale, the number of faults is expected to increase. On million of components, there will always be one component that is not working or not working properly.

The current approach to save snapshot of the simulation data (check-pointing) and in case failures recover from snapshot.

At exascale, the time to save all the simulatio data is larger than average failure times.

Programming models should provide mechanisms to identify processes/group of processes undergoing failures, destroy them and spawn new processes and communicator.

In particular we need dynamic process management.

FUTURE PM: FOCUS ON PRODUCTIVITY

The development of parallel code takes lot of time. How long? In my experience, all large production codes have **+10** years of development.

Part of this problem is that many developers start the code and learn HPC during the development of the code. (Trial/Error development)

Many initiatives were born to provide programming languages that allow fast development. Most famous is the PERCS project funded by DARPA.

WHAT IS PRODUCTIVITY IN HPC?

Make easier the development of a parallel code:

- Write less line of codes for achieving same functionalities of codes written in lower level languages, i.e. C + MPI
- Have a language that makes you avoid bugs, i.e. no pointers, no point-to-point communication, no goto, …
- Don’t deal with low-level stuff (everything can be done by compiler/ runtime). What is low level is error-prone and typically done much better by the compiler.

WHY PRODUCTIVE PROGRAMMING APPROACHES ?

HPC is reaching now new communities other than the usual CFD, Biochemistry, Material Science, Astrophysics communities.

For new HPC communities, good knowledge of Matlab, R, and SQL:

- Biology
- Medicine
- Humanities, i.e. archeologists
- …

WHY PRODUCTIVE LANGUAGES ? LEGACY CODES

Legacy code is code that relates to a no-longer supported or manufactured operating system or other computer technology.

Typical examples are code written in Fortran77 and still in use today.

Very common situation in University groups working on codes coming from the seventies (CFD, Weather forecast, …)

Many of these codes were not designed for running on distributed memory machines with accelerators

PRODUCTIVITY IDEAS IN HPC

DISTRIBUTED MEMORY MACHINE

MPI  **PGAS**

GPU

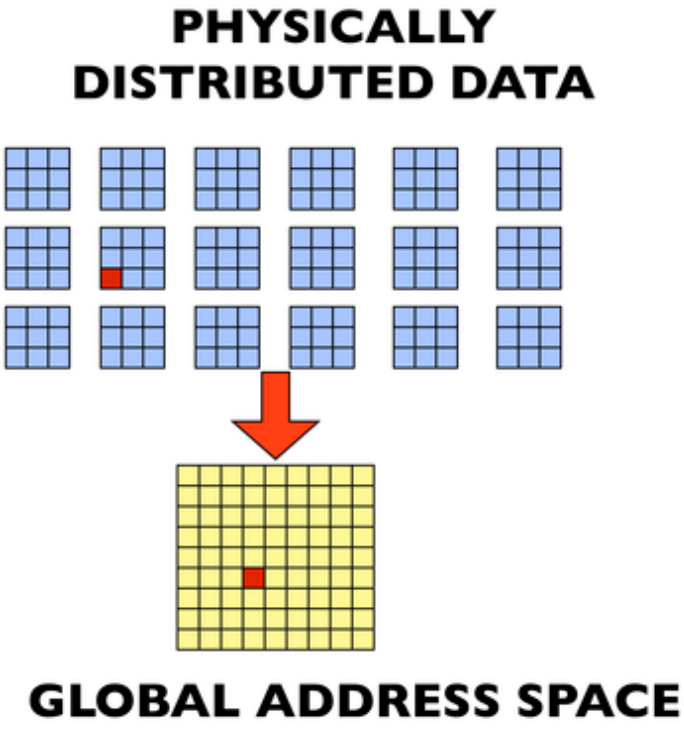
CUDA,
OpenCL  **OpenACC,**
OpenMP 4

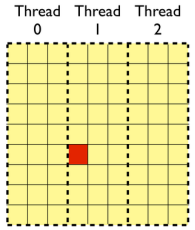
PARTITIONED GLOBAL ADDRESS SPACE (PGAS)

The PGAS programming model provides a global shared (among processes) memory space, that is physically divided on different nodes.

Each Process (Thread) can remotely and directly access this global shared memory regardless this data is physically located.

In PGAS programming models, the communication is implicit (PGAS is taking care of this for you); however problems by accessing concurrently to shared data might occur (race conditions)





PGAS APPROACHES

Many of the PGAS languages originates from 2002 DARPA HPCS program. All the major languages have a PGAS "extension":

- C → UPC (Universal)
- C++ → Co-array C++
- Fortran → Co-array Fortran
- Java → Titanum

USEFUL IF YOU HAVE SERIAL CODE

New parallel languages:

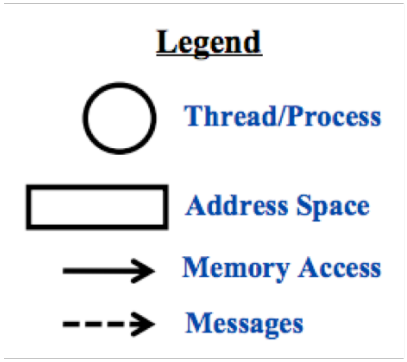
- Chapel (Cray)
- X10 (IBM)
- Frotress (Sun)

BOLDEST APPROACH, USEFUL IF START NEW CODE

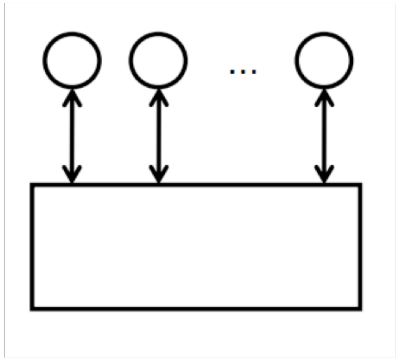
Libraries:

- Global Arrays
- OpenSHMEM
- GPI

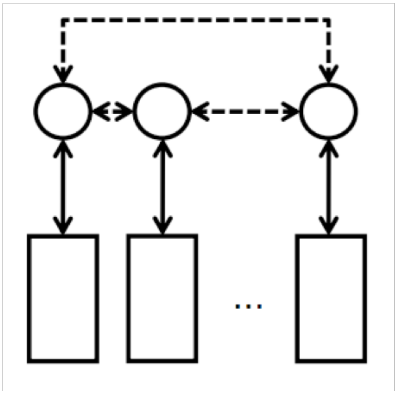
PGAS AS A "COMBINATION" OF OPENMP AND MPI



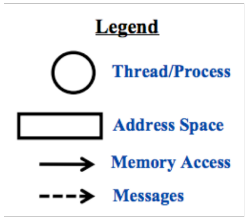
Shared Memory Model
(last week)

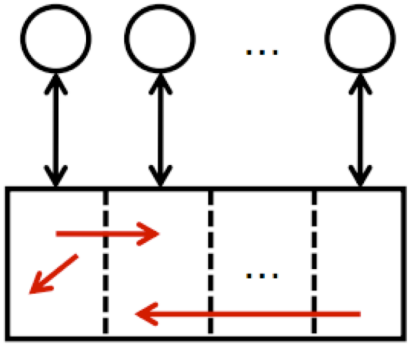


Message Passing Model
(this week)

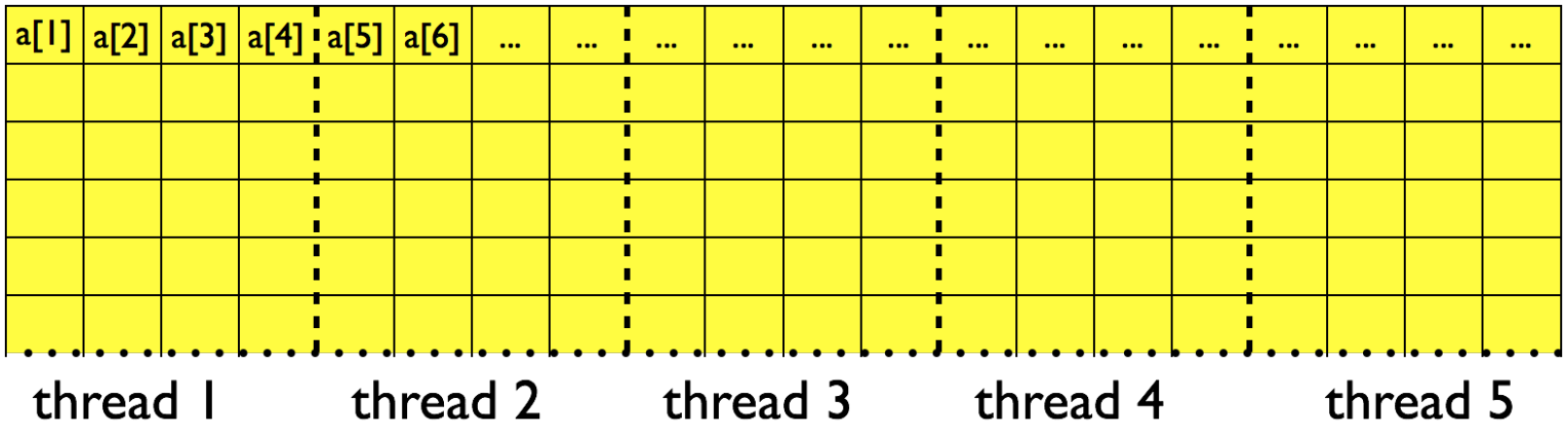


OpenMP → Productivity
MPI → Performance





EXAMPLE: PGAS SHARED ARRAY a



What happens if we want to calculate
a[1] = a[4] + a[5]; ?
the value a[5] is transferred to thread 1 by implicit communication and summed to a[4] and set in thread 1 as a[1]

THE “SHARED MEMORY” SIDE OF PGAS

From OpenMP and shared memory programming approaches, PGAS is taking:

- the **shared** and **private** variable scope. Shared accessible by all the threads, and while each process has its own private variables not accessible by other processes. This partition of memory space between shared and private gives the P in PGAS
- Work sharing (distribution of work among threads) similar to #pragma omp parallel for, i.e in UPC upc_forall
- Concept of affinity: association of one thread to one core (to minimize thread migration and context switching)
- Unfortunately for us, even challenge of detecting race-conditions and checking correctness of the code.

THE “MPI” SIDE OF PGAS

PGAS programs operate in Single Program, Multiple Data (like MPI) fashion: multiple processes execute the same program, but the execution paths can depending on the process ID.

This gives you a local view of the processes, making you thinking more often about locality (=performance)

```
#include <upc.h>  /* needed for UPC extensions */
#include <stdio.h>

main() {
  printf("Thread %d of %d: hello UPC world\n",
         MYTHREAD, THREADS);
}
```

UPC example

WHY TO USE PGAS IN MPI CODES ?

- Typically PGAS is used with MPI because one-sided communication is easier to use in PGAS than in MPI (this especially true for MPI 2.0)

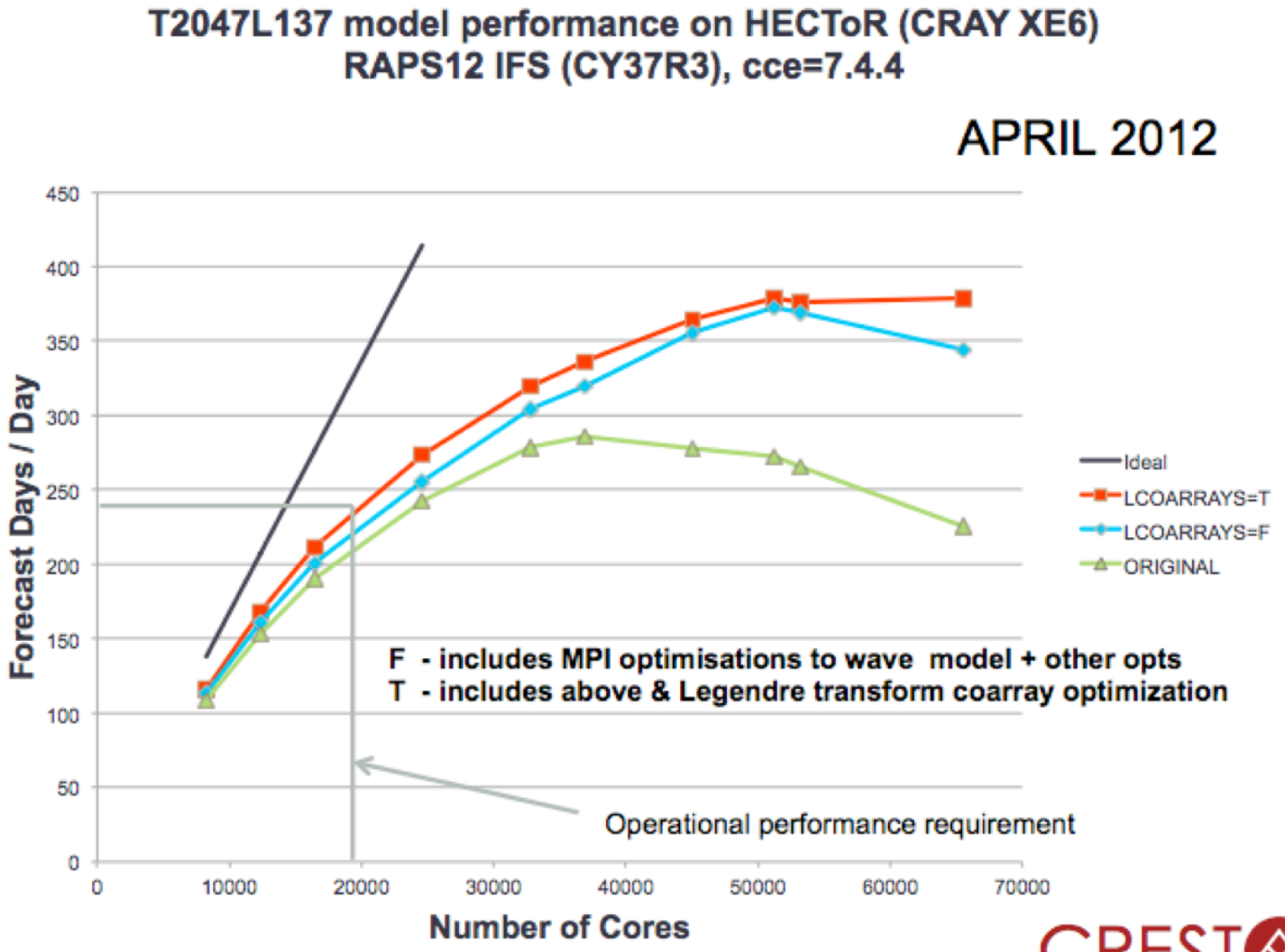
- One-sided communication was faster in PGAS than MPI (Hoefler' group implementation much faster than previous)

- On a single node, PGAS doesn't use extra memory MPI uses (buffers, …) – saw memory issue at exascale earlier.

SUITABILITY OF PGAS FOR EXASCALE

- PGAS reduces the need of temporary buffers and allows for reduced synchronization. PGAS is well suited for applications with irregular communication pattern.
- PGAS has good potential for exascale, but it requires disruptive changes in data layout in the current codes running on peta-scale supercomputers.
- Codes need to use asynchronous algorithms to fully exploit PGAS features. This requires the re-design of communication pattern of applications

SCALING OF CAF + MPI IN A LEGACY CODE

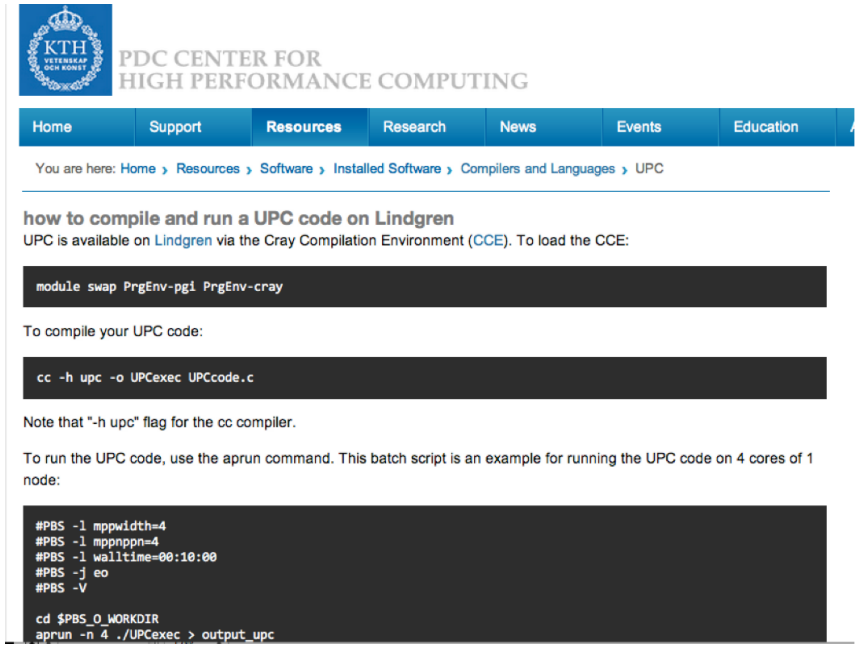


CREST

PGAS AT PDC

UPC/CAF compilers are available on PDC Cray supercomputers. Info on how to compile and run:

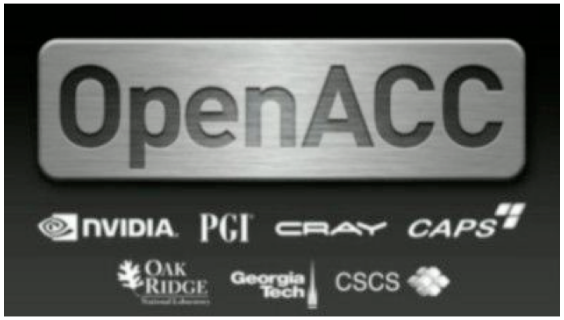
- **UPC** (http://www.pdc.kth.se/resources/software/installed-software/compilers-and-languages/upc)
- **CAF** (http://www.pdc.kth.se/resources/software/installed-software/compilers-and-languages/coarray-fortran)



Example codes are available too.

COMPILER DIRECTIVES FOR ACCELERATOR PROGRAMMING

- The most productive approach is to instruct the compiler where to run part of the code (CPU/GPU) and let the compiler handle the **memory transfer** and code translation for accelerator.
- OpenACC, and openMP4, provide a collection of compiler directives to use accelerators..



OPENMP FOR ACCELERATORS AND OPENACC

There are currently two standardization efforts ongoing:

- In the OpenMP Architecture Review Board (ARB) standards committee, a subcommittee was established to develop an extension to the existing OpenMP 3.0 standard that would **target a wide class of possible accelerators**. This would include GPUs, but also address other accelerators e.g. digital signal processors (DSPs).

- However, there was a need for a minimal, interim standard to serve early adopters of the directive programming model for GPU programming model. To this end, the OpenACC standard launched in November 2011, with support from NVIDIA and compiler developers Cray, PGI and CAPS.

OPENACC: AN OPENMP FOR GPU

- A set of compiler directives (#pragma)
- Offload specific loops or parallelizable sections in code onto accelerators

```
#pragma acc parallel {
    for(i = 0; i < size; i++) {
        A[i] = B[i] + C[i];
    }
}
```

- Routines to allocate/free memory on accelerators

```
buffer = acc_malloc(MYBUFSIZE);
acc_free(buffer);
```

- Supported for **C, C++ and Fortran**
- Huge list of modifiers – **copy, copyout, private, independent, etc.**

EXAMPLE OF OPENACC IN FORTRAN (AXB = C)

```
double precision a(n1, n2, m), b(n2, n3, m), c(n1, n3, m)
double precision tmp
!$acc data copyin(a,b) copyout(c)
!$acc kernels loop independent
do imat = 1, m
!$acc loop independent
do j = 1, n3
!$acc loop independent
do i = 1, n1
tmp = 0.0
!$acc loop
do k = 1, n2
tmp = tmp + a(i,k,imat)*b(k,j,imat)
end do
c(i, j, imat) = tmp
end do
end do
!$acc end kernels
end do
!$acc end data
```

The outermost 'data' region ensures that the input matrices a and b are copied to the GPU, and that the result c is copied back to the host. The outermost loop over 'imat' is marked as being the place to start parallelization using the 'kernels' directive. All of the four loops are marked as parallelizable

OPENMP FOR ACCELERATORS

OpenMP 4.0 allows host and device memory to be shared. New pragmas:

omp target [map] → marks a region to execute on device

omp teams → creates a league of thread teams

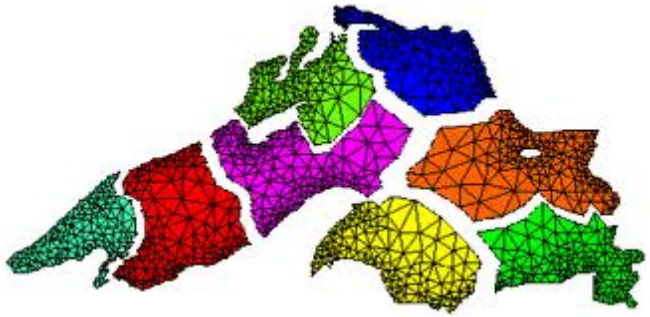
omp distribute → distributes a loop over the teams in the league

omp declare target / omp end declare target → marks function(s) that can be called on the device

Not “real” support from compilers, i.e. target pragma always offloads to host in gcc.

MOTIVATION: LOAD BALANCING

In many problems, a problem could be perfectly divided over processors, i.e. PDE each process take part of the grid. In this case, a processor would always be performing useful work, and only be idle because of communication.

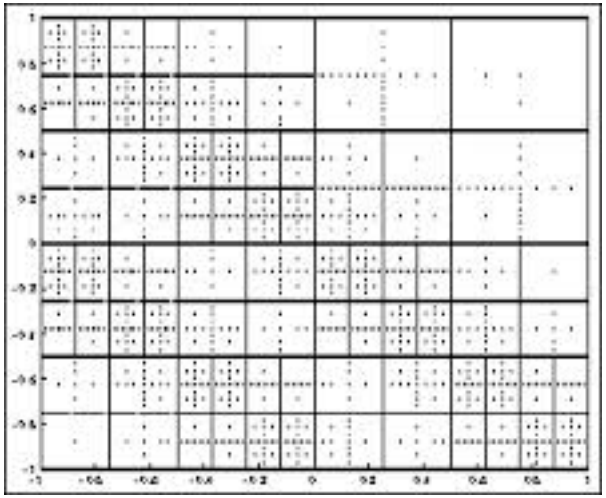


A processor may be idle because it is waiting for a message, and the sending processor has not even reached the send instruction in its code. This situation, **where one processor is working and another is idle**, is called **load unbalance** → a process have been working if we had distributed the work load differently

LOAD BALANCING

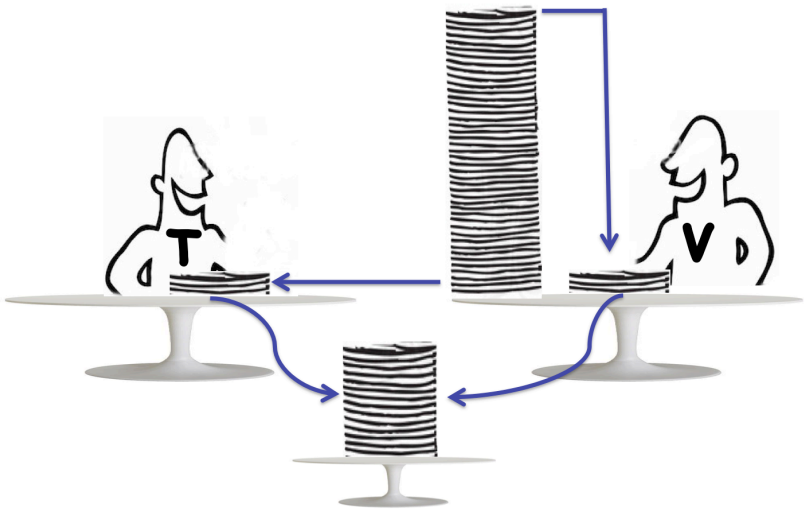
Dynamic load balancing issues become even more important at exascale. They arise from:

- Algorithm and applications: adaptive grids, particle codes,…
- Hardware: computing resources have different computing speed (and memory access speed too).
- Hardware failures become more likely at exascale. Need to give work to other processes



DYNAMIC LOAD BALACING WITH TASKS APPROACH

The task-based programming approach is the best fit to solv the dynamic load balancing problem.



It is based on **the over-decomposition of the work**, we divide the work in more tasks than processors. Tasks are assigned to a work-pool, and available processors take next task from the pool whenever they finish the job..

WORK-POOL AND SCHEDULER

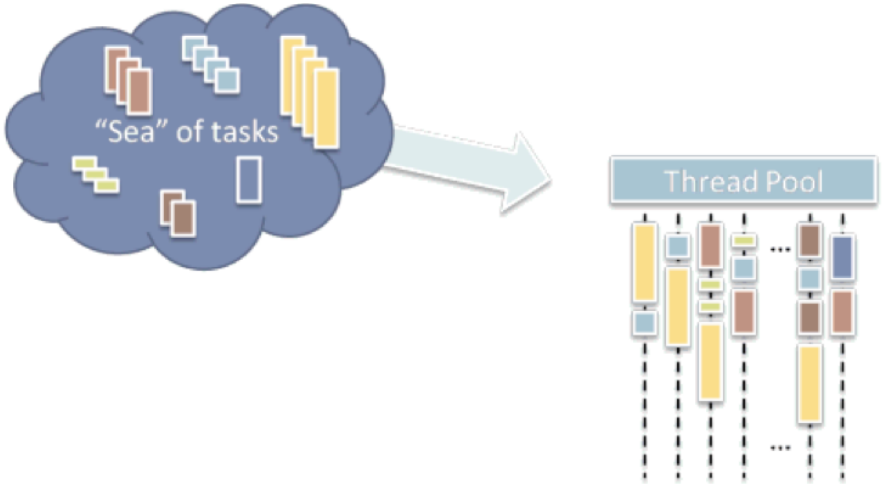
How Many threads ?

Too few threads will undersubscribe the system → waste some of the available hardware resources.

Too many threads → oversubscribe the system, causing the operating system to have overhead as it must time-slice access to the hardware resources.

One common way to perform the balancing act is **to create a pool of threads.**

The application then dynamically schedules computations (tasks) on to threads in the thread pool.



TASK-BASED APPROACHES

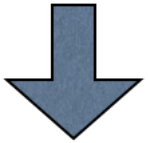
- **CILK** → extension to C including keywords to handle parallel computing. Developed at MIT and then bought by Intel. For shared memory.
- Intel Thread Bulilding Block (**TBB**) → Intel template library for C++. A typical code with TBB creates, synchronizes and destroy graphs of depend tasks. For shared memory.
- **OpenMP** Tasks (saw last week)

CILK – NESTED PARALLELISM

Parallelism is expressed using a spawn task statement while a sync statement forces a parent task to wait until all its children are finished. Tasks may be **nested** up to arbitrary depth.

C

```
int fib (int n) {
if (n<2) return (n);
  else {
    int x,y;
    x = fib(n-1);
    y = fib(n-2);
    return (x+y);
  }
}
```

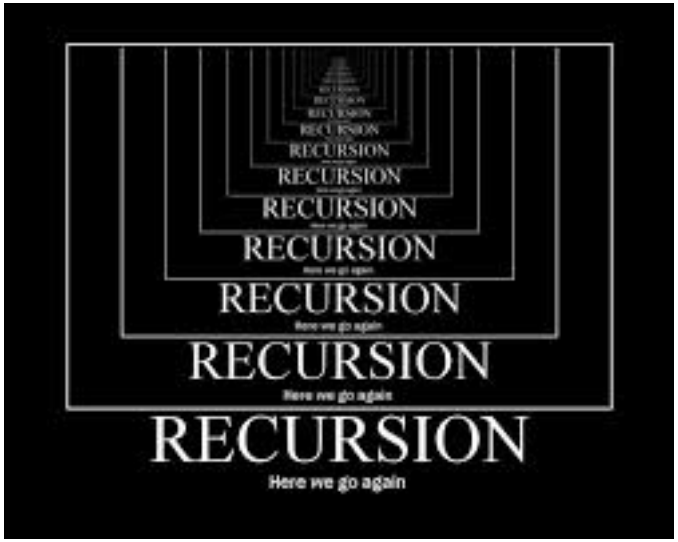


Cilk

```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

RECURSION WITH TASKS

Tasks are effective with recursion. Recursion is often slower than iteration for serial programming, but it turns out that recursive parallelism has some advantages over iterative parallelism with respect to load balancing and cache reuse on multicore processes



CACHE OBLIVIOUS TASK ALLOCATION

Cache oblivious = tailor task allocation to caches size without knowing the size of the caches.

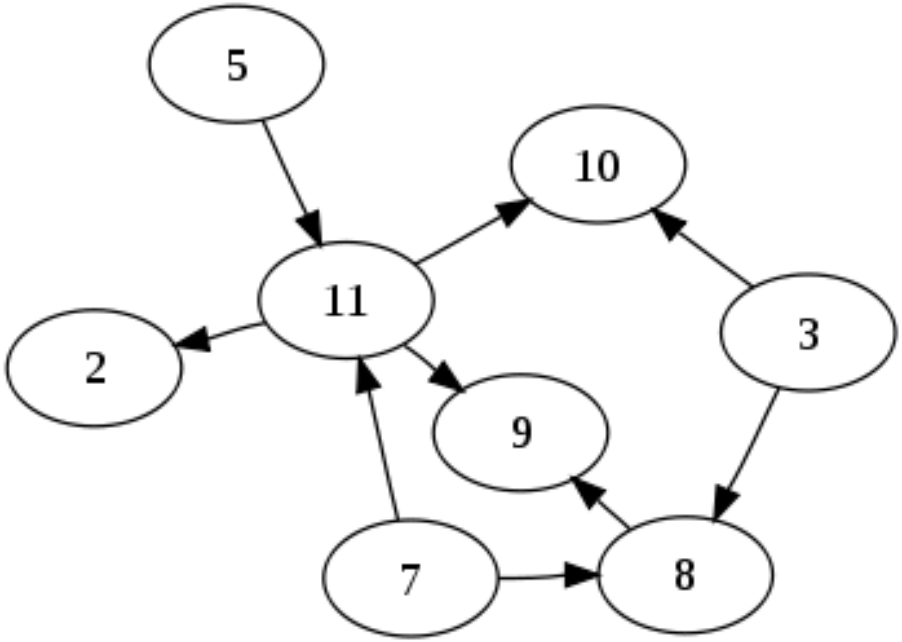
The problem is divided into smaller and smaller sub-problems. Eventually, one reaches a sub-problem size that fits into cache, regardless of the cache size.

FFT algorithms can take advantage of this cache oblivious technique.

CONVENIENCE OF TASK BASED APPROACHES

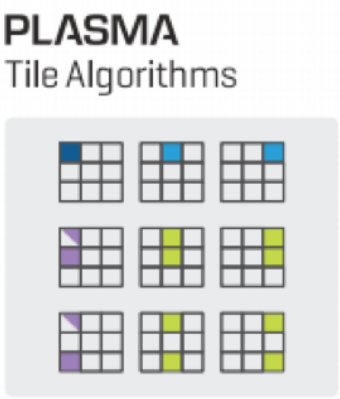
Task approaches solve the problem of dynamic of load balancing but with extra-cost of a scheduler (overhead).

In order for an application to benefit from task approach, your algorithm need to be formulated in terms of tasks in a DAG graph

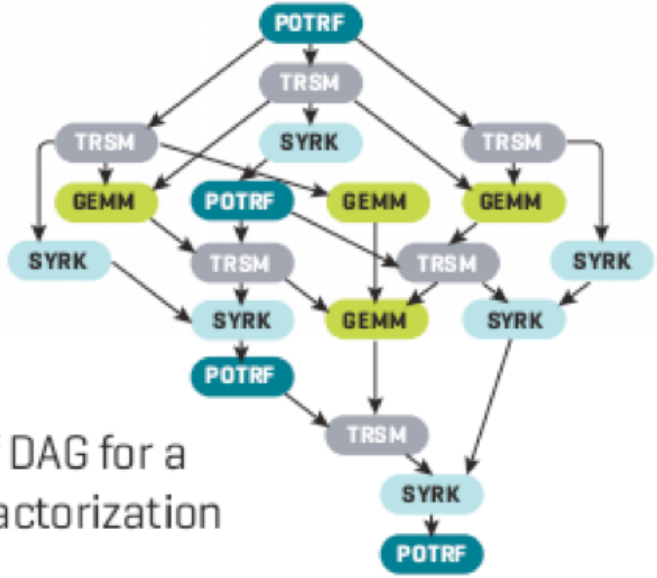


THE PLASMA LIBRARY – RETHING ALGORITHMS IN TERMS OF TASKS

- One of the most famous example of “rethinking” algorithms in terms of tasks is the plasma library developed by ORNL. It comprises linear systems solvers and other linear algebra routines.



- Plasma uses tile algorithms that can be represented as a DAG where nodes represent the tasks in which the operation can be decomposed and the edges represent the dependencies among them.



Example of DAG for a Cholesky Factorization

CONCLUSIONS

Future programming approaches will address new challenges coming from HW:

- power consumption → accelerators?
- less memory per node → which programming model should we use on single node ? → MPI + X issue
- failure and fault tolerance → dynamic processes management

High Productivity is one of the main focuses for future programming models:

- PGAS for distributed memory machines
- OpenACC and OpenMP 4

Dynamic Load Balancing:

- Task-based programming approaches