Introduction to Parallel Programming

Daniel Lucio

May 15th, NCCU, Durham, NC

Outline

1. Parallelism vs Concurrency?
2. Parallel paradigms
   - *2.1.Shared-Memory*
   - *2.2.Distributed-Memory*
3. Design Models
   - *3.1.Task Decomposition*
   - *3.2.Data Decomposition*
4. Methodologies
5. Design Factor Scorecard
6. What's Not Parallel?
7. Programing Paradigms
   - *7.1.Message Passing Interface*
   - *7.2.OpenMP*
8. Anything else?

Outline

1. Parallelism vs Concurrency?
2. Parallel paradigms
   - *2.1.Shared-Memory*
   - *2.2.Distributed-Memory*
3. Design Models
   - *3.1.Task Decomposition*
   - *3.2.Data Decomposition*
4. Methodologies
5. Design Factor Scorecard
6. What's Not Parallel?
7. Programing Paradigms
   - *7.1.Message Passing Interface*
   - *7.2.OpenMP*
8. Anything else?

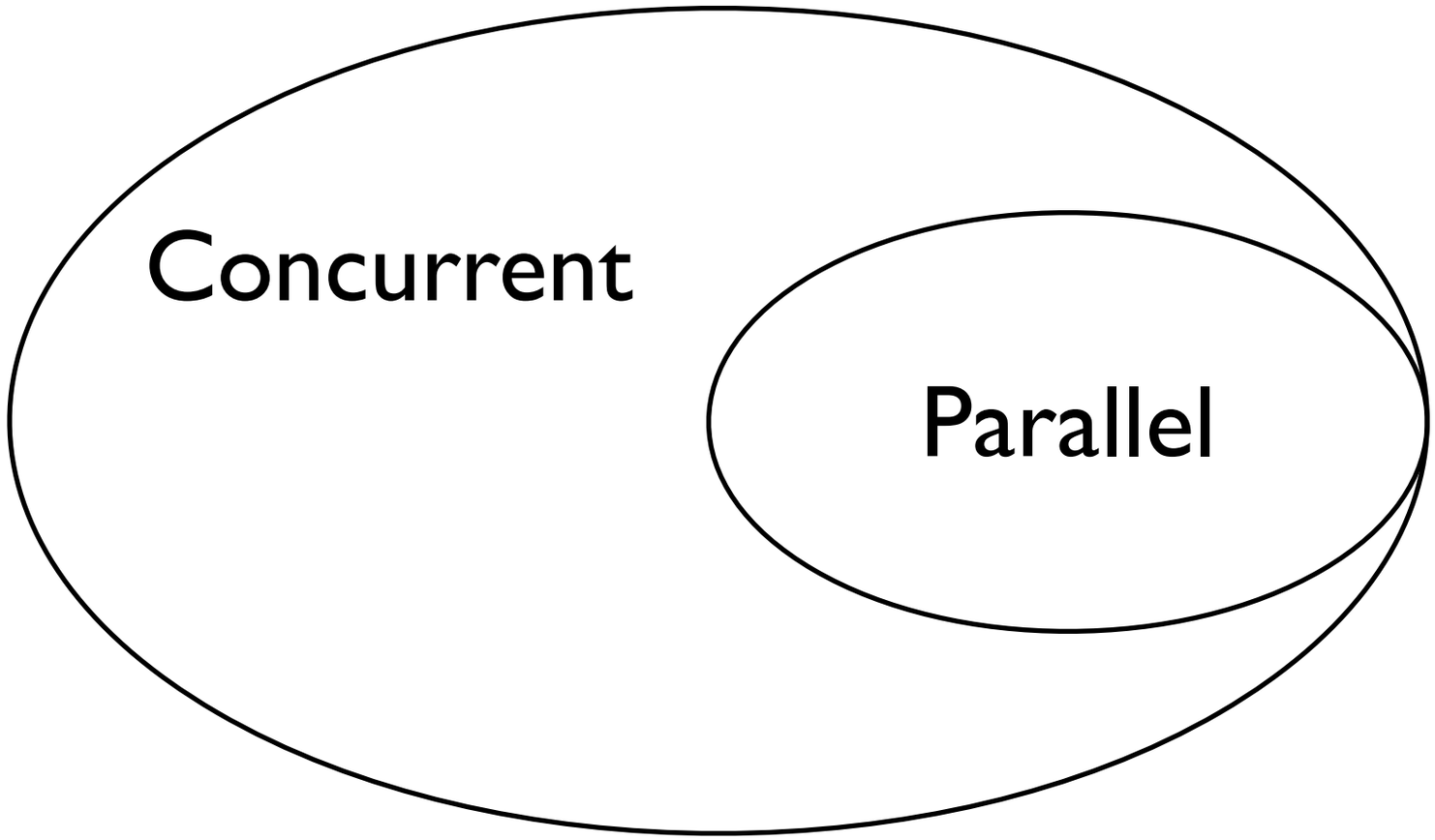
Concurrency or Parallelism?

A concurrent program or algorithm is one where operations can occur at the same time. For instance, a simple integration, where numbers are summed over an interval. The interval can be broken into many concurrent sums of smaller sub-intervals. *Concurrency is a property of the program.*

Parallel execution is when the concurrent parts are **executed** at the same time on separate processors. The distinction is subtle, but important. And, *parallel execution is a property of the machine, not the program.*

Concurrency or Parallelism?

A system is said to be concurrent if it can support two or more actions **in progress** at the same time. A system is said to be parallel if it can support two or more actions **executing** simultaneously.



Quiz: Does Iphone 4 supports parallelism?

Quiz: Does Iphone 4 supports parallelism?

One Core processor!

Quiz: Does Iphone 4 supports parallelism?

One Core processor!

Quiz: Does Iphone 4 supports parallelism?

One Core processor!

Quiz: Does Iphone 4 supports parallelism?

NO

One Core processor!

Quiz: Does Iphone 4s supports parallelism?

Quiz: Does Iphone 4s supports parallelism?

Quiz: Does Iphone 4s supports parallelism?

Dual Core processor!

Quiz: Does Iphone 4s supports parallelism?

Dual Core processor!

Quiz: Does Iphone 4s supports parallelism?

YES

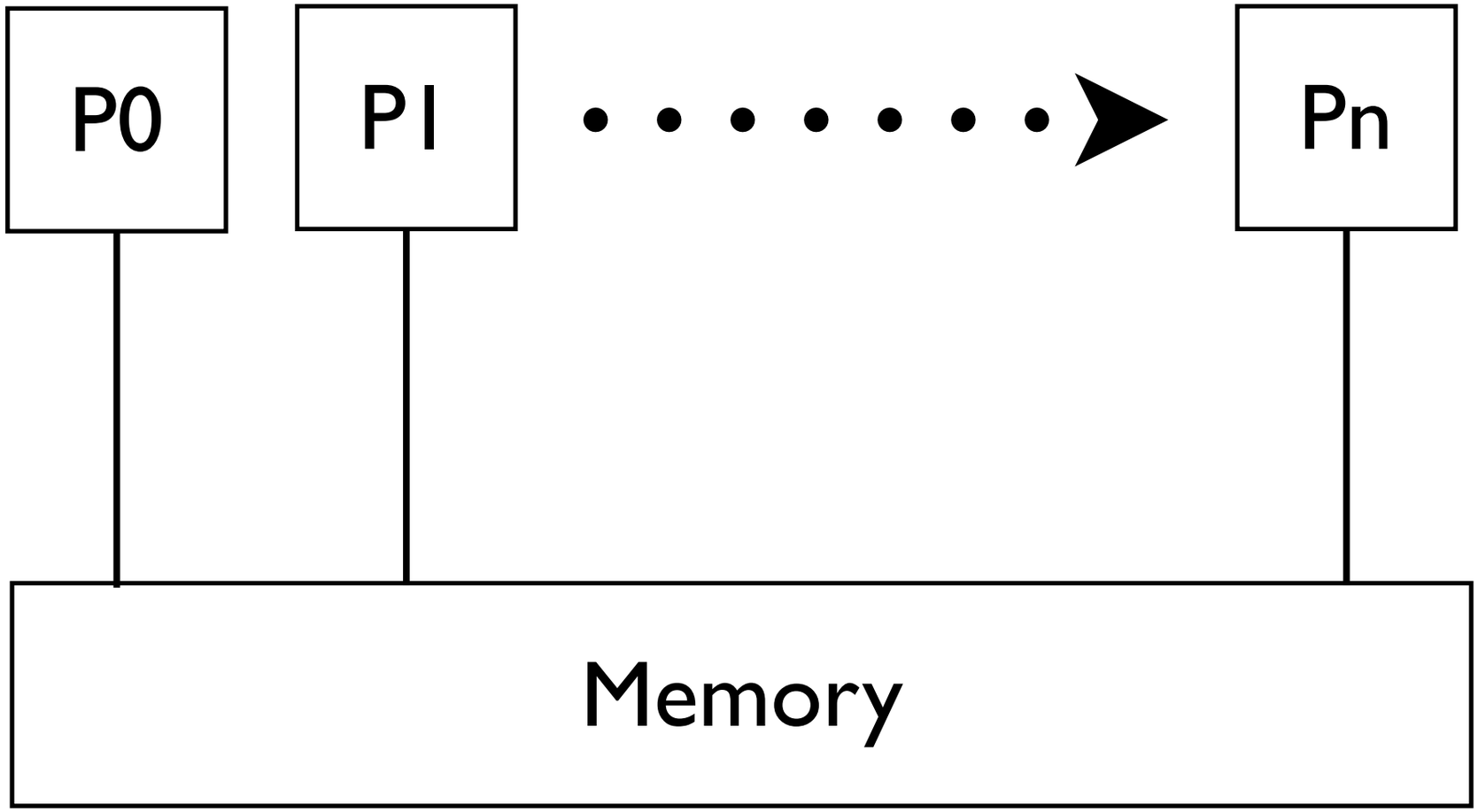
Dual Core processor!

Outline

1. Parallelism vs Concurrency?
2. Parallel Models
   - *2.1. Shared-Memory*
   - *2.2. Distributed-Memory*
3. Design Models
   - *3.1. Task Decomposition*
   - *3.2. Data Decomposition*
4. Methodologies
5. Design Factor Scorecard
6. What's Not Parallel?
7. Programing Paradigms
   - *7.1. Message Passing Interface*
   - *7.2. OpenMP*
8. Anything else?

Shared Memory

Shared Memory has been used to designate a symmetric multiprocessor system (SMP) whose individual processors share memory in such a way that each of them can access any memory location with relative same speed.



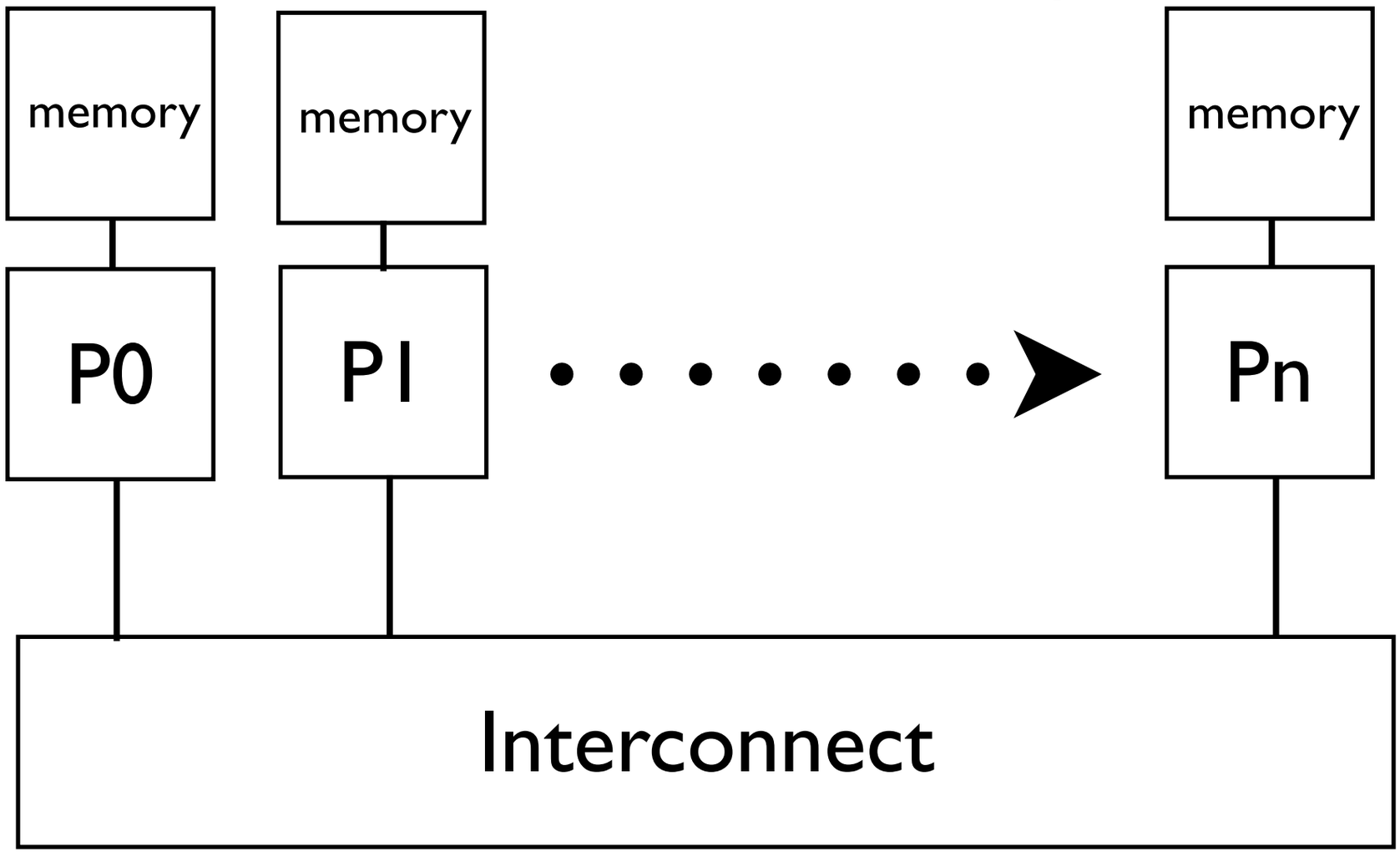
Shared Memory

- **Blacklight** is an SGI UV 1000 cc-NUMA shared-memory system comprising 256 blades.
- Each blade holds 2 Intel Xeon X7560 (Nehalem) eight-core processors, for a total of 4096 cores across the whole machine.
- Each core has a clock rate of 2.27 GHz, supports two hardware threads and can perform 9 Gflops.
- Thus, the total floating point capability of the machine is 37 Tflops.

The central hub of RDAV's capabilities is an SGI Altix 1000 UltraViolet system named **Nautilus**. The UltraViolet architecture is a shared memory system with a single system image, allowing both exploitation of a large number of processors for distributed processing and the execution of legacy serial analysis algorithms for very large data processing by large numbers of users simultaneously. Nautilus has 1024 cores (Intel Nehalem EX processors) and 4 terabytes of global shared memory.

Distributed Memory

Distributed memory refers to a multiple-processor computer system in which each processor has its own private memory. Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors.

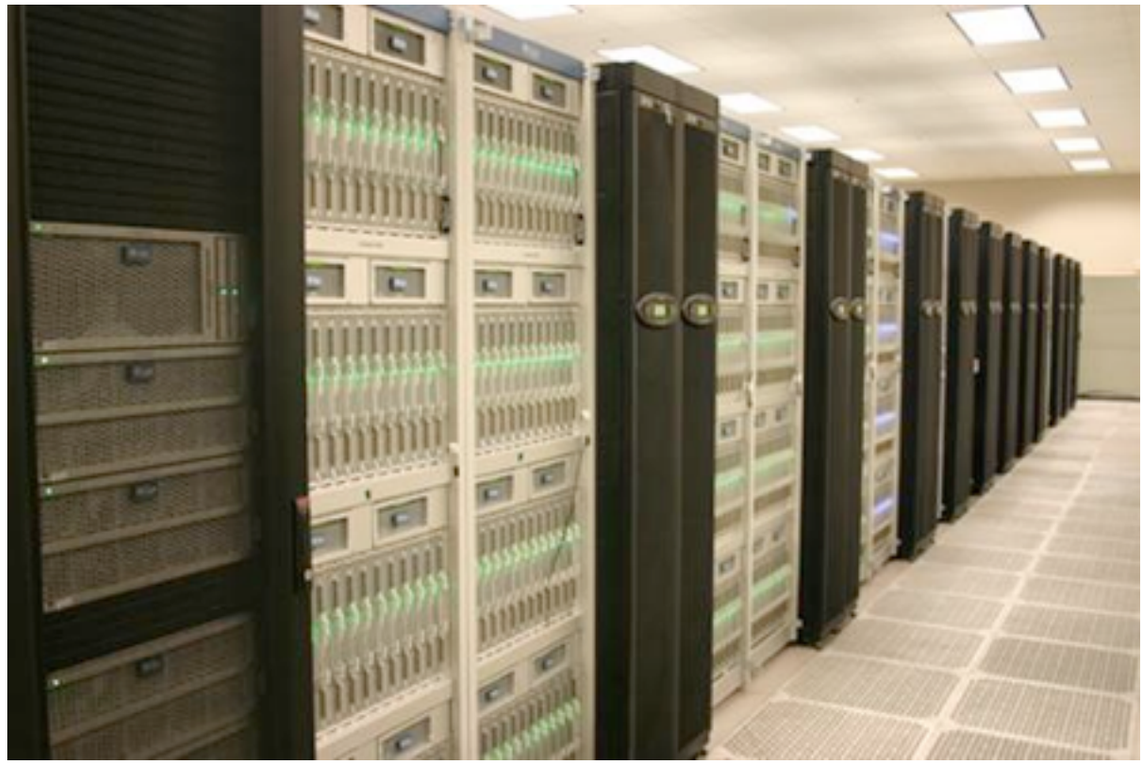


Distributed Memory

Kraken Cray XT5 system specifications:

- Cray Linux Environment (CLE) 3.1
- A peak performance of 1.17 PetaFLOP
- 112,896 compute cores
- 147 TB of compute memory
- A 3.3 PB raw parallel file system of disk storage for scratch space (2.4 PB available)
- 9,408 compute nodes

The **Ranger** system is comprised of 3,936 16-way SMP compute nodes providing 15,744 AMD Opteron processors for a total of 62,976 compute cores, 123 TB of total memory and 1.7 PB of raw global disk space. It has a theoretical peak performance of 579 TFLOPS. All Ranger nodes are interconnected using InfiniBand technology in a full-CLOS topology providing a 1GB/sec point-to-point bandwidth.



Common features

- Extra code/logic/work

  Extra code and logic is needed to make the code work: synchronize, communicate,...

- Dividing work

  There has to be a plan on how to distribute the data or work to be done.

- Sharing data

  There will be times when processes need to share something

- Static/dynamic allocation of work or data

  Work/data can be assigned at one time or over time

Features unique to shared memory

- Local declarations/storage

There are times it will be useful to have a private or local variable that is accessed only by one thread.

- Memory effect

Race conditions can be a problem.

- Communication in memory

Threads share data by reading/writing to/from same memory location.

Features unique to shared memory

- Producer/consumer technique

  Shared queues that contain capsulated tasks become a common technique to distribute work safely among threads (consumer/worker).

- Reader/Writer locks

  There is a need for locks that allow multiple reader threads to enter protected area of code accessing shared variable.

- Mutual exclusion

  There is a need for a mechanism to allow access to one and only one thread at a time, specially when writing to memory.

Outline

1. Parallelism vs Concurrency?
2. Parallel paradigms
   - *2.1.Shared-Memory*
   - *2.2.Distributed-Memory*
3. Design Models
   - *3.1.Task Decomposition*
   - *3.2.Data Decomposition*
4. Methodologies
5. Design Factor Scorecard
6. What's Not Parallel?
7. Programing Paradigms
   - *7.1.Message Passing Interface*
   - *7.2.OpenMP*
8. Anything else?

Task decomposition

Example: *"Let's prepare a meal"*

- *Beverages*
- *Salad*
- *Soup*
- *Entree*
- *Dessert*

All these items can be prepared independently one from the other!

Task decomposition

- The goal of task decomposition is to identify computations/tasks that are completely independent.
- The tasks should be able to execute in any order.
- The most basic framework for doing work in parallel, is to have the main process define and prepare the tasks, launch the workers to execute their taks, and then wait until all the spawned processes have completed.
- There are three key elements needed to consider:
  - *What are the tasks and how are they defined?*
  - *What are the dependencies between tasks and how can they be satisfied?*
  - *How are the tasks assigned to threads*

Task decomposition

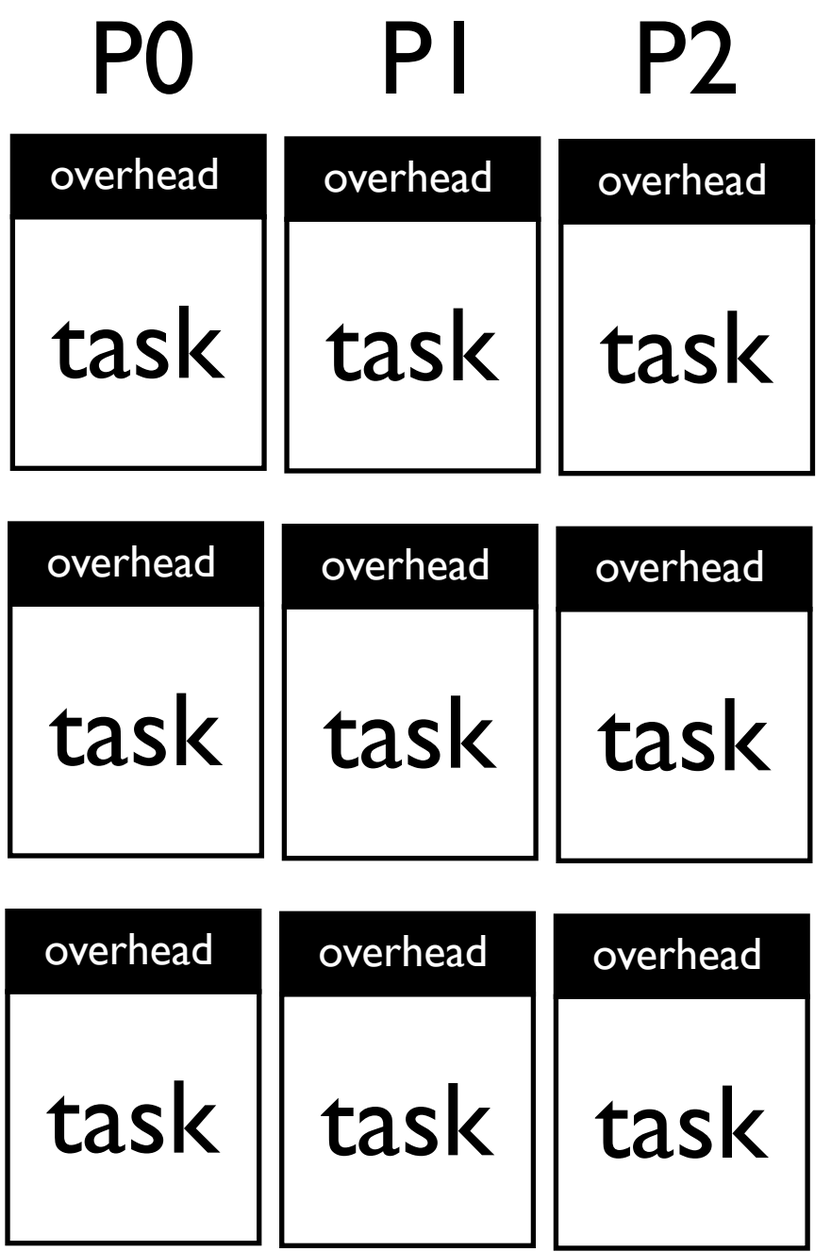
What are the tasks and how are they defined?

- To get the biggest return on investment, you should initially focus on computationally intense portions of the application.
- The following aspects are important when doing the decomposition:
  - There should be at least as many tasks as there will be threads/processes (workers).
  - The amount of computation within each task must be large enough to offset overhead from supporting the parallel system.

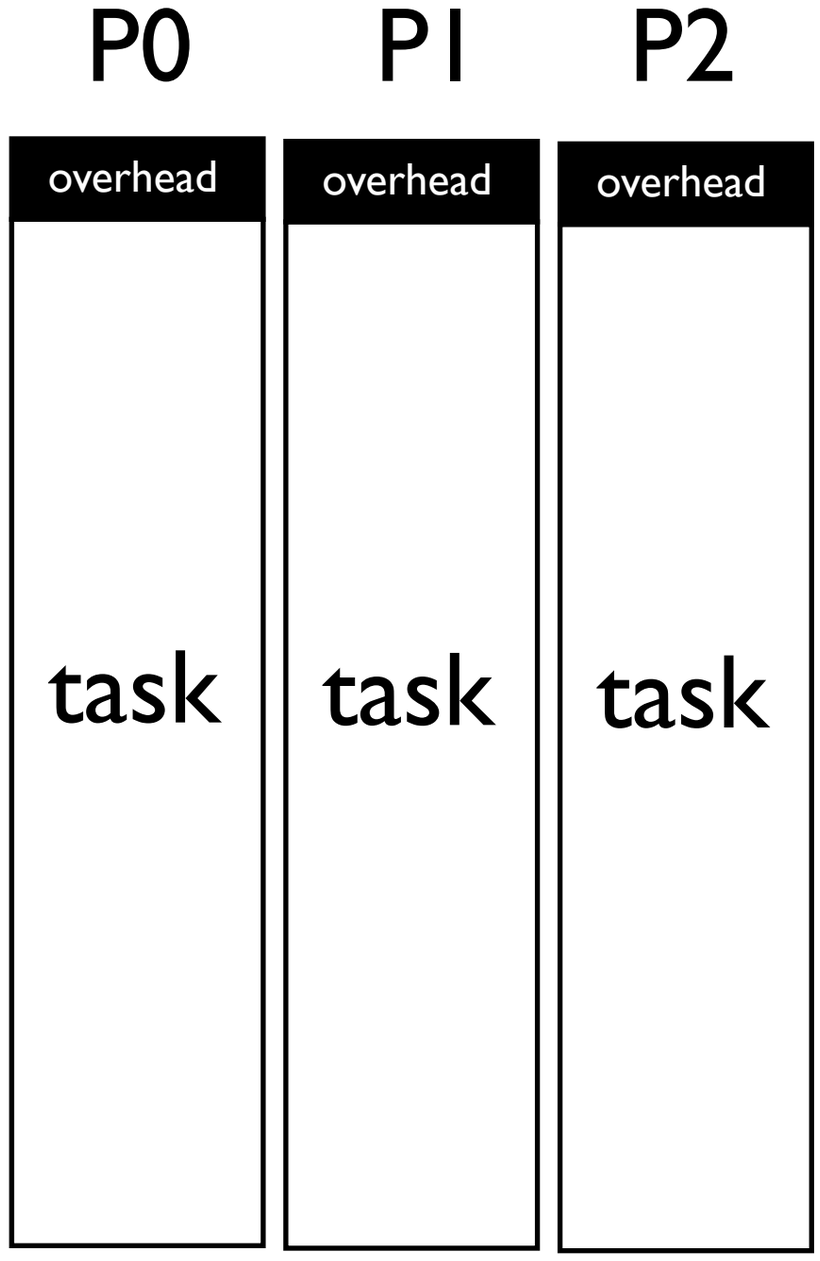
Task decomposition

- The amount of computation/work within a task is called the granularity. The more computation there is within a task, the higher the granularity; the less computation there is, the lower the granularity. The terms *coarse-grained* and *fine-grained* are used to describe instances of high granularity and low granularity.
- Granularity can also be defined as the amount of computation done before synchronization is needed. The longer the time between synchronizations, the coarser the granularity will be. Fine-grained concurrency can suffer from not having enough work to do in order to overcome the overhead (cost).
- Larger tasks can yield better benefits from cache reuse, efficient memory patterns, ...
- You will need to find a way to balance these two criteria.

Task decomposition



Fine-grained



Coarse-grained

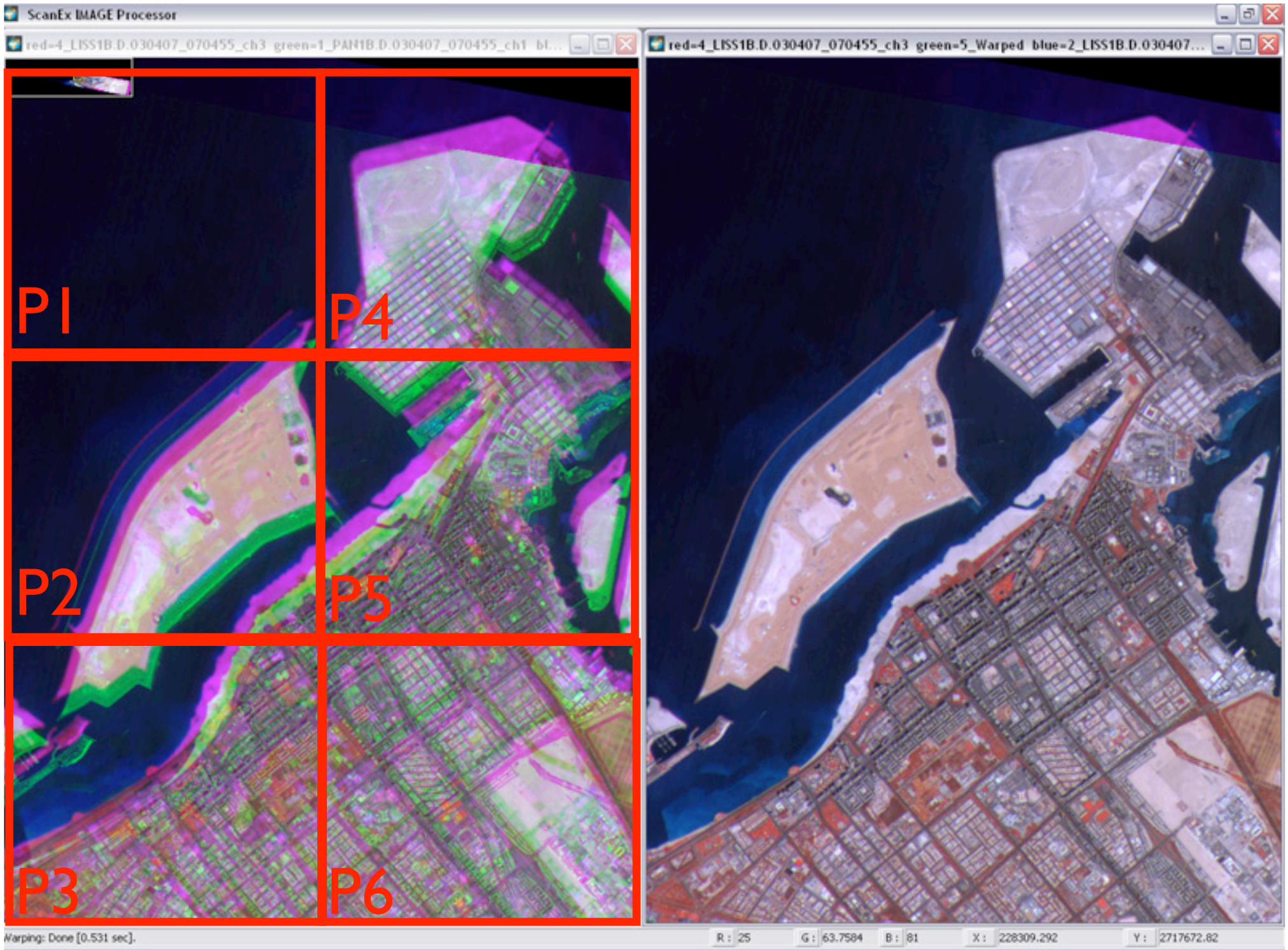
Task decomposition

What are the dependencies between tasks and how can they be satisfied?

- There are two types of dependencies that can occur between tasks:
  - Order dependency: Some tasks depend on the completion or results from previous tasks. Like when building a house: *you do no start with the roof!*
- Data dependency: Shared variables can be updated in the wrong order (non-deterministically) which can yield to wrong ending results. The easiest solution can be achieved by using local variables and/or using mutual exclusion techniques.

Data decomposition

Example: Image processing

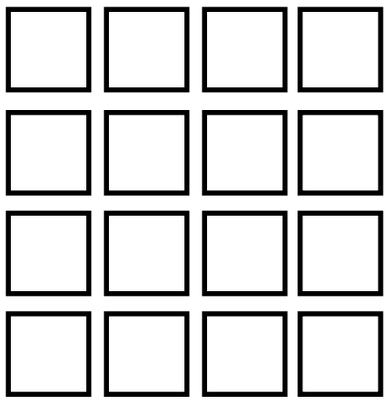


Data decomposition

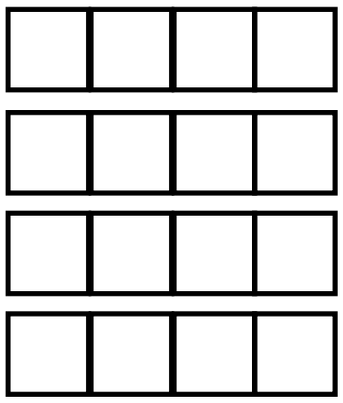
- Data decomposition is the method of defining tasks based on assigning blocks of large data structures to threads/processes. The most common structure used to decompose are arrays.
- There are three key elements to consider:
  - How to divide the data into chunks?
  - How to ensure that the tasks for each chunk have access to all data required for updates?
  - How are the data chunks assigned to threads?

Data decomposition

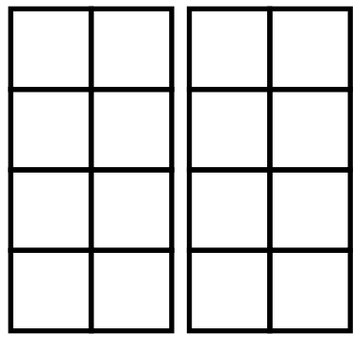
How to divide the data into chunks?



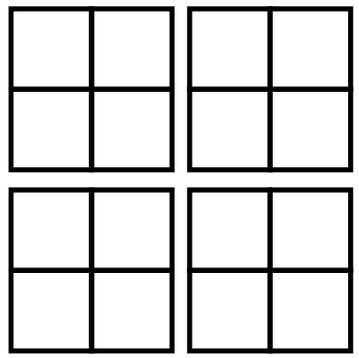
a) Individual elements



b) by row



c) by groups of columns

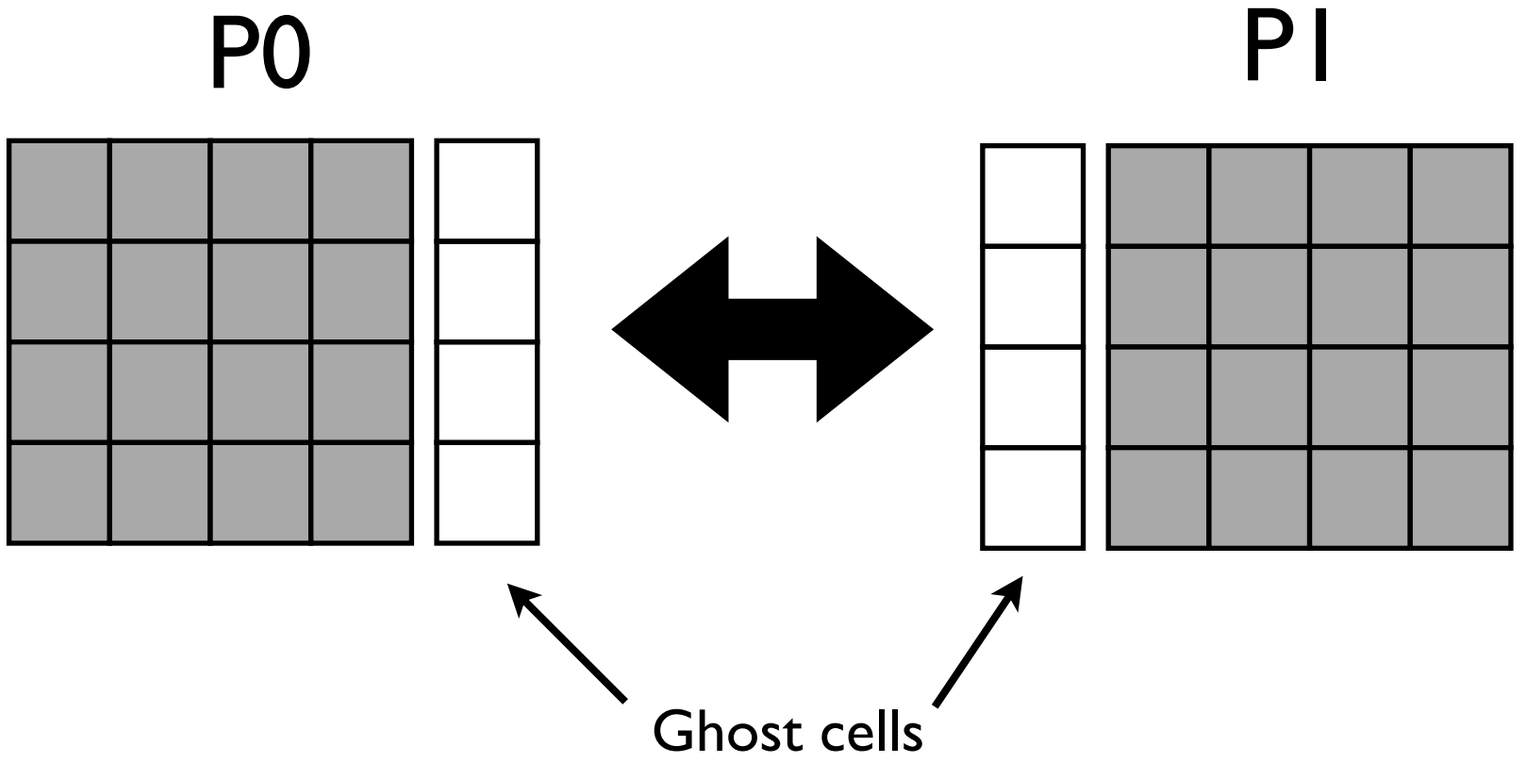


d) by blocks

When doing data decomposition, the shape of chunks is another dimension to consider!

Data decomposition

How to ensure that the tasks for each chunk have access to all data required for updates?



Extra local memory resources can be allocated to hold copied data from a neighboring chunk is often called "ghost cells"

Data decomposition

How are the data chunks assigned to threads?

Similar to task decomposition, data chunks can be assigned to threads/processes statically (one time only), or dynamically (during the execution of the program).

Outline

1. Parallelism vs Concurrency?
2. Parallel paradigms
   - *2.1. Shared-Memory*
   - *2.2. Distributed-Memory*
3. Design Models
   - *3.1. Task Decomposition*
   - *3.2. Data Decomposition*
4. Methodologies
5. Design Factor Scorecard
6. What's Not Parallel?
7. Programing Paradigms
   - *7.1. Message Passing Interface*
   - *7.2. OpenMP*
8. Anything else?

Parallelization methodology

developed by Intel

- Analysis

  Find the parts of the code that can be parallelized. A profile of the serial execution can identify hot spots.

- Design and Implementation
- Test for correctness

  Sometimes the code will work fine in one system, but produce wrong results in another. No data races or deadlocks.

- Tune for performance

  After removing all errors introduced by the parallelization process, you can start tuning the code for performance. Recommended to start from a tuned serial code. Tuning parallel code typically comes down to identifying situations like contention, imbalance, excessive overhead, inefficient memory access. Always check correctness!

Outline

1. Parallelism vs Concurrency?
2. Parallel paradigms
   - *2.1.Shared-Memory*
   - *2.2.Distributed-Memory*
3. Design Models
   - *3.1.Task Decomposition*
   - *3.2.Data Decomposition*
4. Methodologies
5. Design Factor Scorecard
6. What's Not Parallel?
7. Programing Paradigms
   - *7.1.Message Passing Interface*
   - *7.2.OpenMP*
8. Anything else?

Design Factor Scorecard

from Mattson' Patterns for parallel Programming

Criteria used when developing parallel algorithms:

- Efficiency

  Overhead should be minimal. Application should make good use of resources.

- Simplicity

  The simpler your parallel algorithm is, the easier it will be to code it, debug it, verify it and maintain it.

- Portability

  Algorithm should not be platform dependent.

- Scalability

  Refers to the behavior of the program as the number of processors and the problem increases.

  Scalability and then Efficiency could be the most important.

Outline

1. Parallelism vs Concurrency?
2. Parallel paradigms
   - *2.1.Shared-Memory*
   - *2.2.Distributed-Memory*
3. Design Models
   - *3.1.Task Decomposition*
   - *3.2.Data Decomposition*
4. Methodologies
5. Design Factor Scorecard
6. What's Not Parallel?
7. Programing Paradigms
   - *7.1.Message Passing Interface*
   - *7.2.OpenMP*
8. Anything else?

What's not parallel?

Not everything can be executed in parallel. For example:

- Algorithms with state
- Recurrences
- Induction variables
- Reduction
- Loop carried dependencies

Outline

1. Parallelism vs Concurrency?
2. Parallel paradigms
   - *2.1.Shared-Memory*
   - *2.2.Distributed-Memory*
3. Design Models
   - *3.1.Task Decomposition*
   - *3.2.Data Decomposition*
4. Methodologies
5. Design Factor Scorecard
6. What's Not Parallel?
7. Programing Paradigms
   - *7.1.Message Passing Interface*
   - *7.2.OpenMP*
8. Anything else?

Message Passing Interface

- The Message-Passing Interface (MPI) is a standard for expressing distributed parallelism via message passing.
- MPI consists of a header file, a library of routines and a runtime environment.
- When you compile a program that has MPI calls in it, your compiler links to a local implementation of MPI, and then you get parallelism; if the MPI library isn't available, then the compile will fail.
- MPI can be used in Fortran, C and C++.

Message Passing Interface

MPI calls in Fortran look like this:

```
CALL MPI_Funcname(..., errcode)
```

In C, MPI calls look like:

```
errcode = MPI_Funcname(...);
```

In C++, MPI calls look like:

```
errcode = MPI::Funcname(...);
```

Notice that `errcode` is returned by the MPI routine `MPI_Funcname`, with a value of `MPI_SUCCESS` indicating that `MPI_Funcname` has worked correctly.

Message Passing Interface

- MPI is actually just an Application Programming Interface (API).
- An API specifies what a call to each routine should look like, and how each routine should behave.
- An API does not specify how each routine should be implemented, and sometimes is intentionally vague about certain aspects of a routine's behavior.
- Each platform has its own MPI implementation.
- MPI uses the term "rank" to identify a process.

Message Passing Interface

Most common routines

- `MPI_Init` starts up the MPI runtime environment at the beginning of a run.
- `MPI_Finalize` shuts down the MPI runtime environment at the end of a run.
- `MPI_Comm_size` gets the number of processes in a run, `Np` (typically called just after `MPI_Init`).
- `MPI_Comm_rank` gets the process `ID` that the current process uses, which is between `0` and `Np-1` inclusive (typically called just after `MPI_Init`).

Message Passing Interface

Most common routines

- `MPI_Send` sends a message from the current process to some other process (the destination).
- `MPI_Recv` receives a message on the current process from some other process (the source).
- `MPI_Bcast` broadcasts a message from one process to all of the others.
- `MPI_Reduce` performs a reduction (e.g., sum, maximum) of a variable on all processes, sending the result to a single process.

Message Passing Interface

Anatomy of a Fortran MPI program

```
PROGRAM my_mpi_program
  IMPLICIT NONE
  INCLUDE "mpif.h"
     [other includes]
  INTEGER :: my_rank, num_procs, mpi_error_code
     [other declarations]
  CALL MPI_Init(mpi_error_code)      !! Start up MPI
  CALL MPI_Comm_Rank(my_rank,   mpi_error_code)
  CALL MPI_Comm_size(num_procs, mpi_error_code)
     [actual work goes here]
  CALL MPI_Finalize(mpi_error_code) !! Shut down MPI
END PROGRAM my_mpi_program
```

Message Passing Interface

Anatomy of a C MPI program

```
#include <stdio.h>
#include "mpi.h"
 [other includes]

int main (int argc, char* argv[])
{ /* main */
  int my_rank, num_procs, mpi_error;
     [other declarations]
  mpi_error = MPI_Init(&argc, &argv); /* Start up MPI  */
  mpi_error = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  mpi_error = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
     [actual work goes here]
  mpi_error = MPI_Finalize();          /* Shut down MPI */
} /* main */
```

Message Passing Interface

- MPI uses kind of parallelism known as Single Program, Multiple Data (SPMD).
- This means that you have one MPI program – a single executable – that is executed by all of the processes in an MPI run.

- So, to differentiate the roles of various processes in the MPI run, you have to have `if` statements:

```
if (my_rank == server_rank) {
    …
}
```

Message Passing Interface

How it works

- Every process gets a copy of the executable: Single Program, Multiple Data (SPMD).
- They all start executing it.
- Each looks at its own rank to determine which part of the problem to work on.
- Each process works completely independently of the other processes, except when communicating.

Message Passing Interface

Hello world example in C:

```
#include <stdio.h>
#include "mpi.h"

int main ( int argc, char *argv[])
{
  int rank, size;

  MPI_Init (&argc, &argv);      /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);        /* get current process id */
  MPI_Comm_size (MPI_COMM_WORLD, &size);        /* get number of processes */
  printf( "Hello world from rank %d of %d\n", rank, size );
  MPI_Finalize();

  return 0;
}
```

Message Passing Interface

Compiling and running example in Kraken:

```
$ cc -o hello_mpi hello_mpi.c

$ qsub -I -lsize=12,walltime=00:05:00
$ cd $SCRATCHDIR
$ aprun -n 3 ~/suraTraining/MPI/hello_mpi

Hello world from rank 2 of 3
Hello world from rank 1 of 3
Hello world from rank 0 of 3
Application 679385 resources: utime ~0s, stime ~0s
```

In other systems you might need to use '`mpicc`', or link the mpi library with '`-lmpi`'.

Message Passing Interface

Construction of a message

```
MPI_Send(message, strlen(message) + 1,
     MPI_CHAR, destination, tag,
     MPI_COMM_WORLD);
```

- When MPI sends a message, it doesn't just send the contents; it also sends an "envelope" describing the contents:
  - Size (number of elements of data type)
  - Data type
  - Source: rank of sending process
  - Destination: rank of process to receive
  - Tag (message `ID`)
  - Communicator (e.g., `MPI_COMM_WORLD`)

Message Passing Interface

Data types

C		Fortran	
char	MPI_char	CHARACTER	MPI_CHARACTER
int	MPI_INT	INTEGER	MPI_INTEGER
float	MPI_FLOAT	REAL	MPI_REAL
double	MPI_DOUBLE	DOUBLE PRECISION	MPI_DOUBLE_PRECISION

MPI supports several other data types, but most are variations of these, and probably these are all you'll use.

Message Passing Interface

MPI Communicators

- An MPI communicator is a collection of processes that can send messages to each other.
- `MPI_COMM_WORLD` is the default communicator; it contains all of the processes. It's probably the only one you'll need.
- Some libraries create special library-only communicators, which can simplify keeping track of message tags.

Message Passing Interface

- What happens if one process has data that everyone else needs to know?
- For example, what if the server process needs to send an input value to the others?

```
MPI_Bcast(length, 1, MPI_INTEGER,
          source, MPI_COMM_WORLD);
```

- Note that `MPI_Bcast` doesn't use a tag, and that the call is the same for both the sender and all of the receivers.
- All processes have to call `MPI_Bcast` at the same time; everyone waits until everyone is done.

Message Passing Interface

MPI Reductions

- A reduction converts an array to a scalar: for example, sum, product, minimum value, maximum value, Boolean AND, Boolean OR, etc.
- Reductions are so common, and so important, that MPI has two routines to handle them:
- `MPI_Reduce`: sends result to a single specified process
- `MPI_Allreduce`: sends result to all processes (and therefore takes longer)

Message Passing Interface

Non-blocking communications

- MPI allows a process to start a send, then go on and do work while the message is in transit.
- This is called non-blocking or immediate communication.
- Here, "immediate" refers to the fact that the call to the MPI routine returns immediately rather than waiting for the communication to complete.

Message Passing Interface

Immediate send

```
mpi_error_code =
    MPI_Isend(array, size, MPI_FLOAT,
          destination, tag, communicator,
          request);
```

Likewise:

```
mpi_error_code =
    MPI_Irecv(array, size, MPI_FLOAT,
          source, tag, communicator,
          request);
```

This call starts the send/receive, but the send/receive won't be complete until:

```
MPI_Wait(request, status);
```

What's the advantage of this?

Message Passing Interface

Communication hiding

In between the call to `MPI_Isend/Irecv` and the call to `MPI_Wait`, both processes can **do work**!

If that work takes at least as much time as the communication, then the cost of the communication is effectively zero, since the communication won't affect how much work gets done.

This is called communication hiding.

When you want to hide communication:

- as soon as you calculate the data, send it;
- don't receive it until you need it.

That way, the communication has the maximal amount of time to happen in background (behind the scenes).

Calculating π

f(x)

4
3.2
2.4
1.6
0.8
0
y

$f(x_i) = 4/(1 + x_i^2)$

f(x)=4/(1+x^2)

$\Delta = 1/N$

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2} \Delta \simeq \pi$$

0
0.25
0.5
0.75
1
x

0

$x_i = (i + 0.5)\Delta$

1

Serial PI example

```
#include <stdio.h>
static long num_steps = 100000;
double pi, sum = 0.0;

int main () {
  int i;
  double step, x;
  step = 1.0/(double)num_steps;
  for (i=1;i<= num_steps; i++) {
    x = step*((double)i-0.5);
    sum += 4.0/(1.0+x*x);
  }
  pi = step * sum;
  printf("Pi is %4.11f\n",pi);

  return 0;
}
```

Symbolically we know

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

which can be discretized

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2} \Delta \simeq \pi$$

$$x_i = (i+0.5)\Delta$$

$$\Delta = 1/N$$

```
$ pgcc -o pi pi.c
$ ./pi
Pi is 3.14159265360
```

Message Passing Interface

Computing the value of 'π' with MPI

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int n=1000, myid, numprocs, i;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
            h   = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i += numprocs) {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
            mypi = h * sum;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                       MPI_COMM_WORLD);
            if (myid == 0)
                printf("pi is approximately %.16f\n", pi);
    MPI_Finalize();
    return 0;
}
```

Broadcast

Reduce op

Only root

Message Passing Interface

Compiling and running example in Kraken:

```
$ cc -o pi_mpi pi_mpi.c

$ qsub -I -lsize=12,walltime=00:05:00
$ cd $SCRATCHDIR
$ aprun -n 3 ./pi_mpi

pi is approximately 3.1415927369231262
Application 679597 resources: utime ~0s, stime ~0s
```

In other systems you might need to use '`mpicc`', or link the mpi library with '`-lmpi`'.

Outline

1. Parallelism vs Concurrency?
2. Parallel paradigms
   - *2.1. Shared-Memory*
   - *2.2. Distributed-Memory*
3. Design Models
   - *3.1. Task Decomposition*
   - *3.2. Data Decomposition*
4. Methodologies
5. Design Factor Scorecard
6. What's Not Parallel?
7. Programing Paradigms
   - *7.1. Message Passing Interface*
   - *7.2. OpenMP*
8. Anything else?

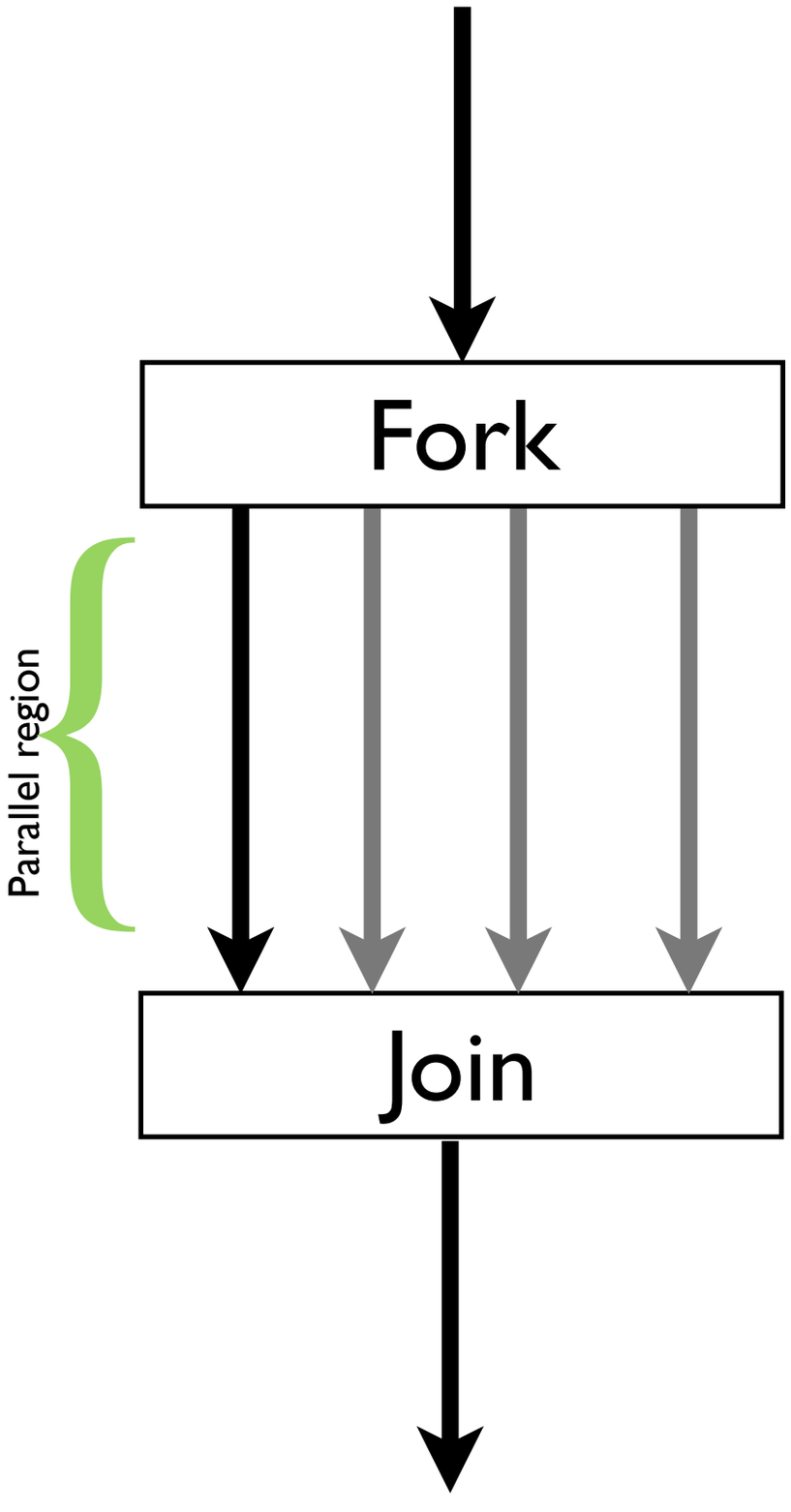
OpenMP

- OpenMP was defined by the OpenMP Architecture Review Board (ARB), a group of vendors who joined forces during the latter half of the 1990s to provide a common means for programming a broad range of SMP architectures.
- OpenMP is not a new programming language! It can be considered as a "notation" that can be added to a sequential program in Fortran, C or C++ to describe how the work is to be shared among threads that will execute on different processors/cores, and to order accesses to shared data as needed.
- The OpenMP Application Programming Interface (API) was developed to enable portable shared memory parallel programming.

OpenMP

- An OpenMP directive is an instruction in a special format that is only understood by OpenMP-aware compilers.
- The API is designed to permit incremental approach to parallelizing an existing code, possibly in successive steps. (Different to a all-or-nothing conversion as MPI).
- The impact of OpenMP parallelization is frequently localized, i.e. modifications are often needed in few places.
- The main approach is based in identifying the parallelism in the program, and not in reprogramming the code to implement the parallelism.
- It is possible to write the code such that the original sequential version is preserved.

Fork/Join model



The program starts as a single thread just like a sequential program. The thread that executes this code is referred to as the *initial thread*.

Whenever an OpenMP *parallel* construct is encountered, a team of threads is created (fork). The initial thread becomes the master of the team. They all execute the code enclosed in the construct.

At the end of the construct, only the original thread continues and all other terminate (join).

OpenMP

The OpenMP API provides directives, library functions and environment variables to create and control the execution of parallel programs.

Basic constructs

- Parallel construct
- Work-Sharing constructs
  - Loop constructs
  - Sections constructs
  - Single constructs
  - Workshare constructs (Fortran only)
- Data-sharing, No-wait, and, Schedule classes

Synchronization

- Barrier construct
- Critical construct
- Atomic construct
- Locks
- Master construct
- Locks

Parallel Construct

This construct is used to specify the code that should be executed in parallel. The code not enclosed by a parallel construct will NOT be executed in parallel.

C/C++ syntax

```
#pragma omp parallel [clause[[,] clause]...]
{
   /* parallel section */
}
```

Fortran syntax

```
!$omp parallel [clause[[,] clause]...]
   /* parallel section */
!$omp end parallel
```

General form of an OpenMP directive

C/C++ syntax

```
#pragma omp directive-name [clause[[,] clause]...] new-line
```

Note: In C/C++ directives are case sensitive!

Fortran syntax

```
!$omp directive-name [clause[[,] clause]...] new-line
c$omp directive-name [clause[[,] clause]...] new-line
*$omp directive-name [clause[[,] clause]...] new-line
```

Note: In Fortran all directives must begin with a directive sentinel. In fixed-source format Fortran there are three choices, but, the sentinel must start in column one. In free-source format only the first sentinel is supported.

Example: Hello world

```
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
        int tid;
        printf("Hello world from threads:\n");
        #pragma omp parallel private(tid)
        {
                tid = omp_get_thread_num();
                printf("<%d>\n", tid);
        }
        printf("I am sequential now\n");

        return 0;
}
```

OpenMP

Compiling and running example in Kraken:

```
$ cc -mp -o hello_openmp hello_openmp.c

$ qsub -I -lsize=12,walltime=00:05:00
$ cd $SCRATCHDIR
$ export OMP_NUM_THREADS=3
$ aprun -n1 -d3 ~/suraTraining/OPENMP/hello_openmp
Hello world from threads:
<0>
<2>
<1>
I am sequential now
Application 679908 resources: utime ~0s, stime ~0s
```

First attempt to compute π

```
#include <stdio.h>
#include <omp.h>
#define NBIN 100000

int main() {
 double step,sum=0.0,pi;
 step = 1.0/NBIN;
 # pragma omp parallel
 {
  int nthreads,tid,i;
  double x;
  nthreads = omp_get_num_threads();
  tid = omp_get_thread_num();
  for (i=tid; i<NBIN; i+=nthreads) {
   x = (i+0.5)*step;
   sum += 4.0/(1.0+x*x);
  }
 }
 pi = sum*step;
 printf("PI = %f\n",pi);
 return 0;
}
```

First attempt to compute π

OpenMP

```
$ cc -mp -o pi1_omp pi1.c

$ qsub -I -lsize=12,walltime=00:10:00
$ export OMP_NUM_THREADS=3
$ cd $PBS_O_WORKDIR

$ aprun -n 1 -d3 ./pi1_omp

PI = 1.054395
```

Second attempt to compute π

OpenMP

```
#include <stdio.h>
#include <omp.h>
#define NBIN 100000

int main() {
 double step,sum=0.0,pi;
 step = 1.0/NBIN;
 # pragma omp parallel
 {
  int nthreads,tid,i;
  double x;
  nthreads = omp_get_num_threads();
  tid = omp_get_thread_num();
  for (i=tid; i<NBIN; i+=nthreads) {
   x = (i+0.5)*step;
   #pragma omp critical
   sum += 4.0/(1.0+x*x);
  }
 }
 pi = sum*step;
 printf("PI = %f\n",pi);
 return 0;
}
```

Second attempt to compute π


```
$ cc -mp -o pi2_omp pi2.c $ qsub -I -lsize=12,walltime=00:10:00
$ export OMP_NUM_THREADS=3
$ cd $PBS_O_WORKDIR
$ aprun -n 1 -d3 ./pi2_omp
PI = 3.141593
```

Third attempt to compute π

OpenMP

```
#include <stdio.h>
#include <omp.h>
#define NBIN 100000
#define MAX_THREADS 12

int main() {
 int nthreads,tid;
 double step,sum[MAX_THREADS]={0.0},pi=0.0;
 step = 1.0/NBIN;
 #pragma omp parallel private(tid)
 {
  int i;
  double x;
  nthreads = omp_get_num_threads();
  tid = omp_get_thread_num();
  for (i=tid; i<NBIN; i+=nthreads) {
   x = (i+0.5)*step;
   sum[tid] += 4.0/(1.0+x*x);
  }
 }
 for(tid=0; tid<nthreads; tid++) pi += sum[tid]*step;
 printf("PI = %f\n",pi);
 return 0;
}
```

Outline

1. Parallelism vs Concurrency?
2. Parallel paradigms
   - *2.1.Shared-Memory*
   - *2.2.Distributed-Memory*
3. Design Models
   - *3.1.Task Decomposition*
   - *3.2.Data Decomposition*
4. Methodologies
5. Design Factor Scorecard
6. What's Not Parallel?
7. Programing Paradigms
   - *7.1.Message Passing Interface*
   - *7.2.OpenMP*
8. Anything else?

Anything else?

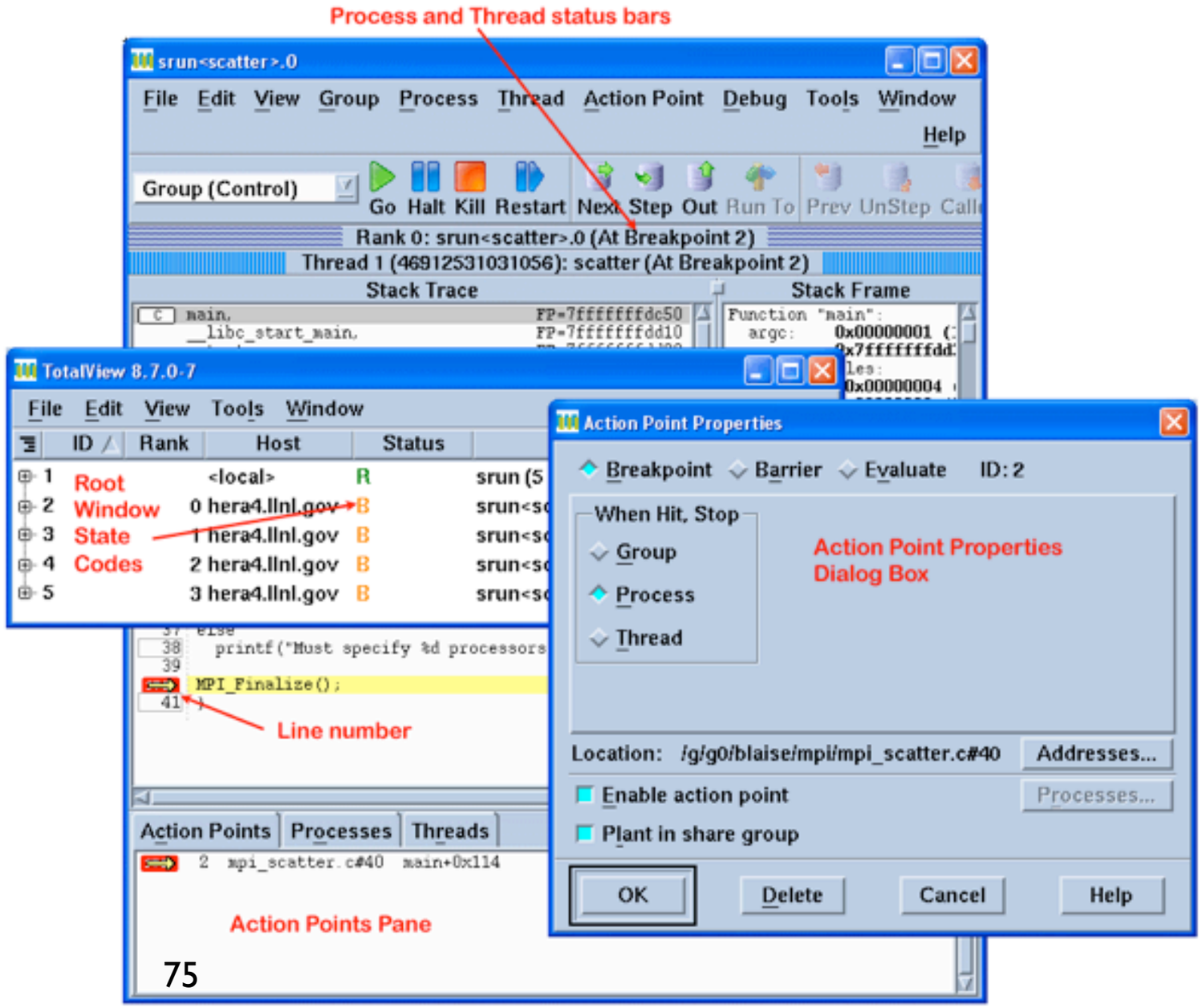
- Debuggers
- Profilers
- Libraries
- I/O
- Power
- Future languages
- Exascale Computing?

Anything else?

- Debuggers
- Profilers
- Libraries
- I/O
- Power
- Future languages
- Exascale Computing?

- TotalView from Rogue Wave Software is a parallel debugging tool that can be run with up to 512 processors.

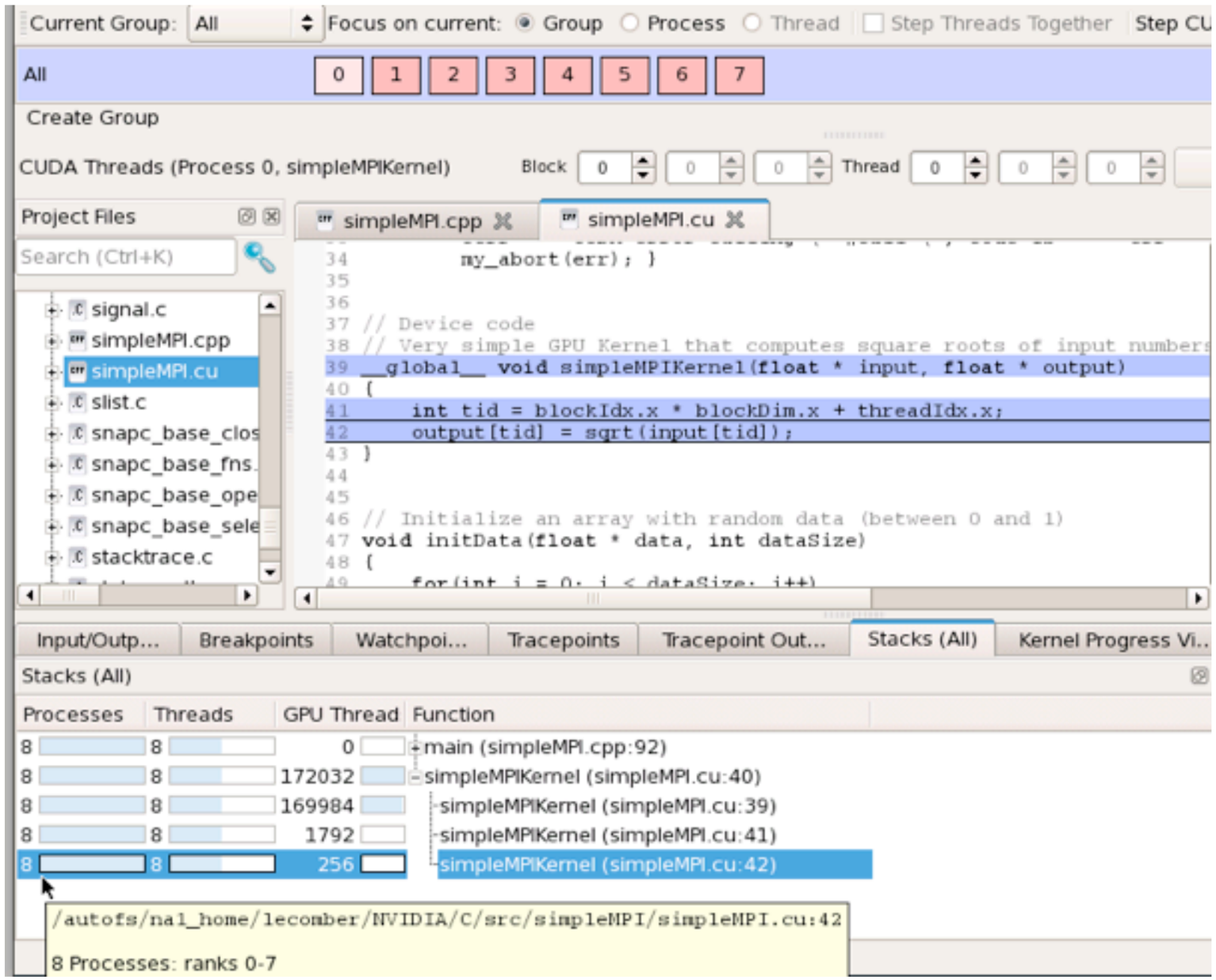
- It provides both X Windows-based Graphical User Interface (GUI) and command line interface (CLI) environments for debugging.



- Distributed Debugging Tool (DDT) from Allinea Software is a parallel debugger

- DDT is a parallel debugger which can be run with up to 8192 processors. It can be used to debug serial, OpenMP, MPI, Coarray Fortran (CAF), UPC (Unified Parallel C) codes.

allinea ddt
the distributed debugging tool



Anything else?

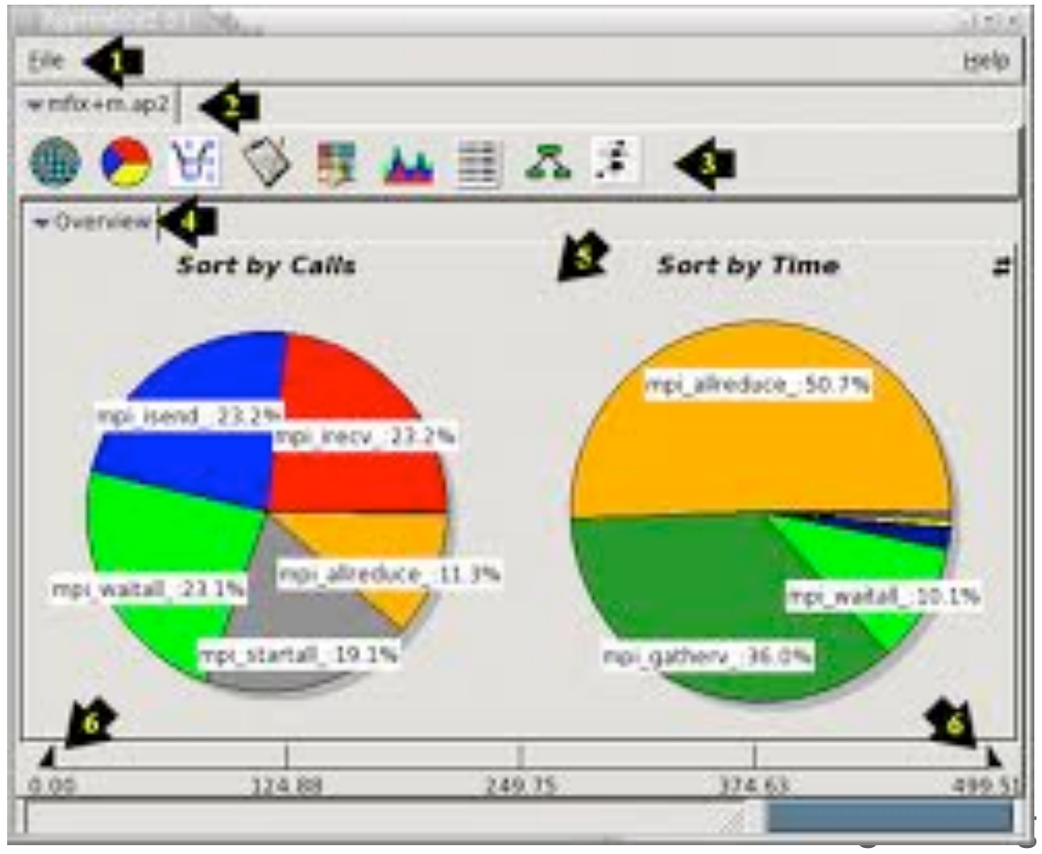
- Debuggers
- Profilers
- Libraries
- I/O
- Power
- Future languages
- Exascale Computing?

- CrayPat is a performance analysis tool offered by Cray for the XT and XE platforms

- Cray Apprentice2 is a tool used to visualize performance data instrumented with the CrayPat tool.

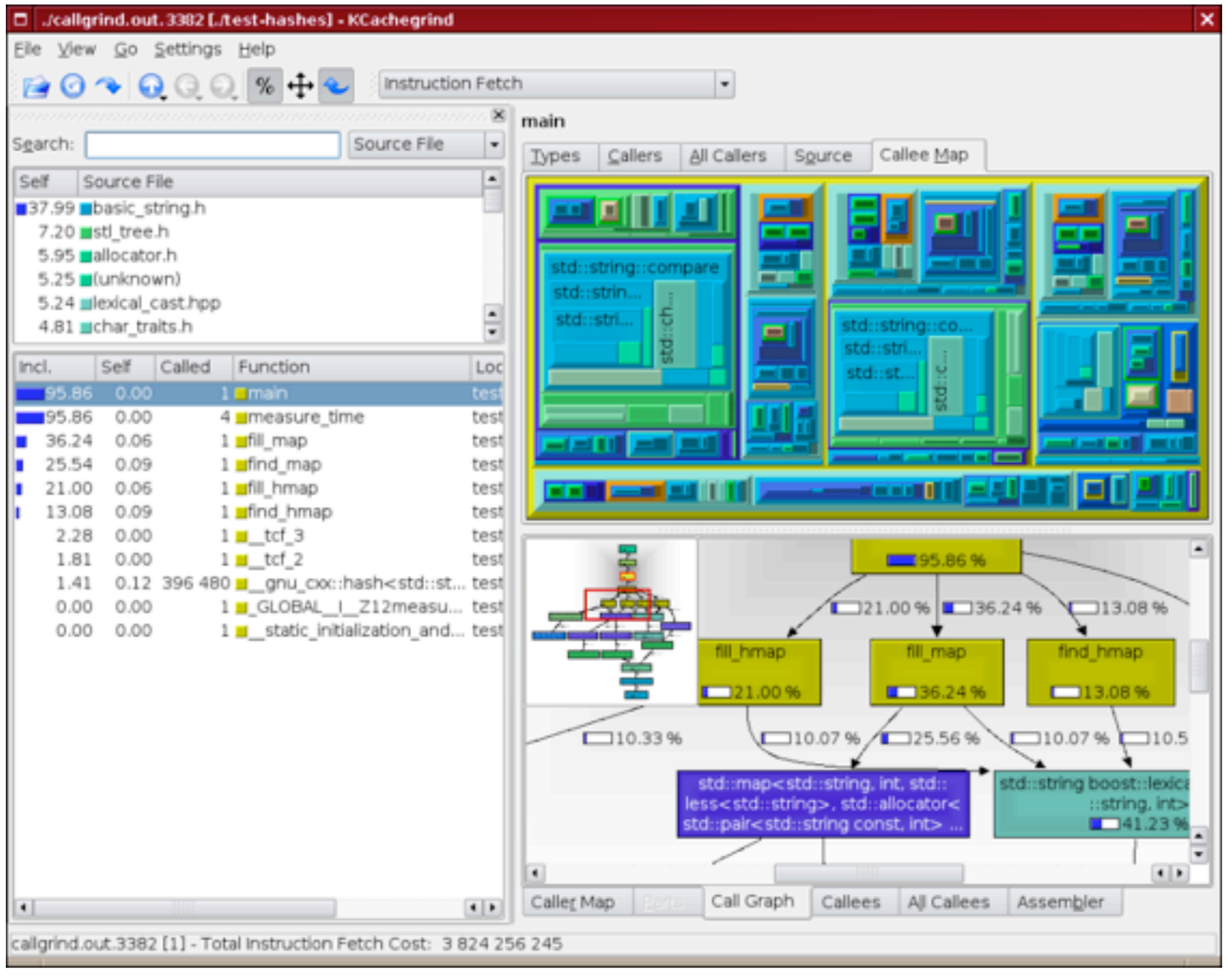
GM3 Cray PAT tracing report with API information from 8 PE run when investigating enhancement of UKCA_COAGWITHNUCL

Time %	Time	Imb. Time	Imb. Time %	Calls	Group / Function / PE='HIDE'
100.0%	2058.854843	--	--	6731515	Total
94.5%	1945.732550	--	--	6557264	USER
36.9%	759.508205	6.805455	1.0%	144	chimie_
6.9%	142.135909	8.451937	6.4%	1440	#11.11 loop init mtrac
6.8%	140.708321	41.589230	26.1%	1	main
6.6%	136.060100	2.901422	2.4%	288	advy2_
4.0%	82.400176	1.758459	2.4%	144	consom_
3.9%	79.572205	0.500504	0.7%	144	advx2_
3.1%	64.781270	2.547494	4.3%	288	advx2_
2.4%	50.124924	1.249039	2.8%	3456	ukca_water_content_v_
2.4%	48.938390	2.616689	5.8%	1440	ukca_conden_
2.3%	47.166706	1.214859	2.9%	4032	ukca_coag_coff_v_
2.1%	43.071936	3.410142	8.4%	1440	#5.5 imode loop
2.0%	41.345881	0.211065	0.6%	864	ukca_volume_mode_
1.9%	39.174169	2.602136	7.1%	7200	ukca_solvecoagnucl_v_
1.8%	38.046784	1.764512	5.1%	14400	ukca_cond_coff_v_
1.8%	37.836103	2.580603	7.3%	1440	#2.imode loop
1.8%	37.507454	1.191105	3.5%	25920	#10.10 ncp loop
1.8%	36.644806	1.473848	4.4%	1440	#7.7 imode loop
1.7%	34.149579	1.831906	5.8%	2617	prls_
1.4%	28.488018	1.145093	4.4%	2617	jac_
3.6%	73.941021	--	--	155216	MPI
2.1%	42.940114	10.573601	22.6%	29385	mpi_sendrecv_
1.0%	20.905757	8.770574	33.8%	35596	mpi_recv_
1.9%	39.181272	--	--	19035	MPI_SYNC
1.4%	28.908872	4.654048	15.8%	1123	mpi_bcast_(sync)

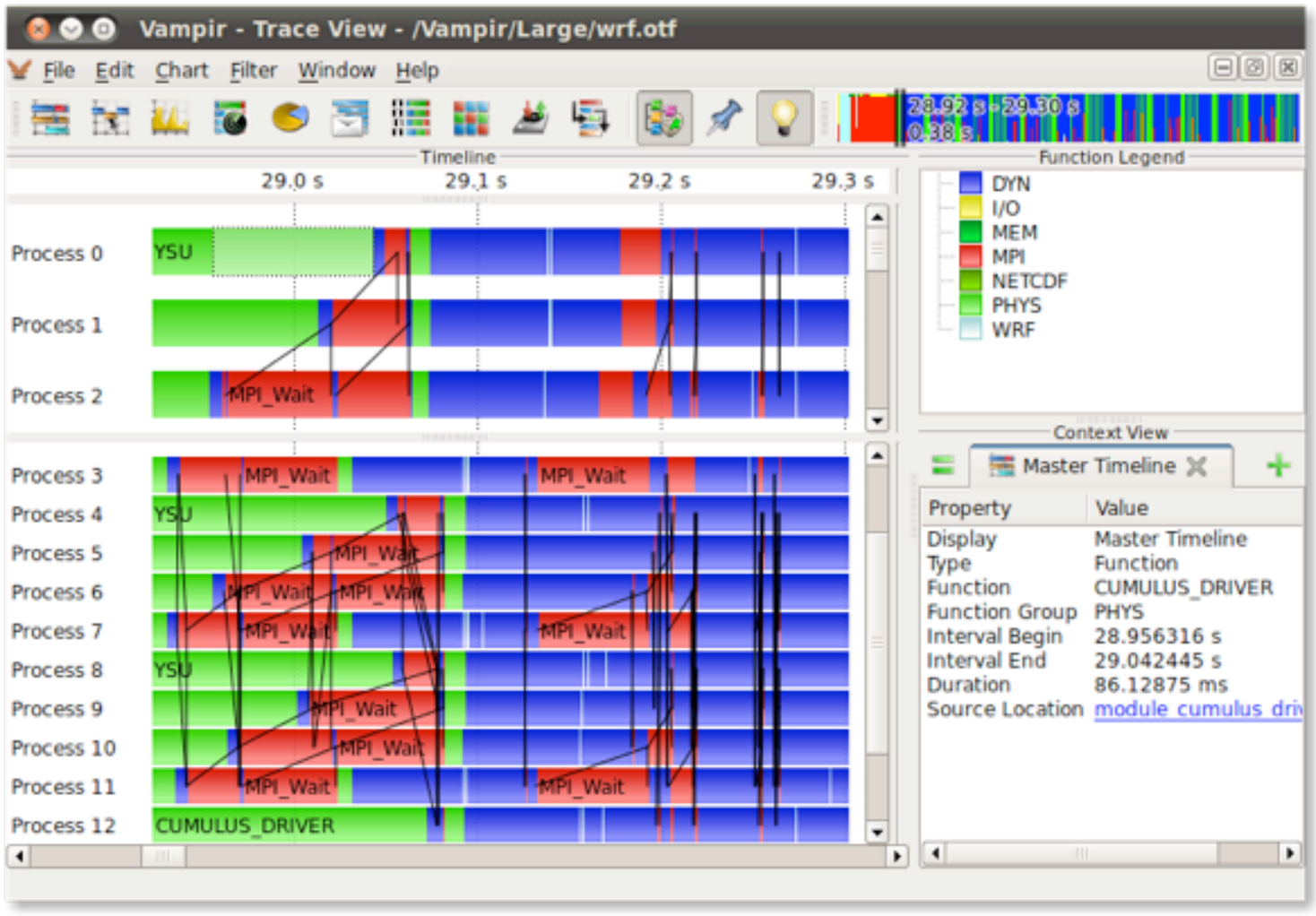


Valgrind is well-known as a tool for finding errors of work with memory. But except this, it also contain number of additional utilities for performance profiling, finding synchronization errors in multi-threading programs and analysis of memory consumption.

Valgrind



Vampir provides an easy to use analysis framework which enables developers to quickly display program behavior at any level of detail.



Anything else?

- Debuggers
- Profilers
- Libraries
- I/O
- Power
- Future languages
- Exascale Computing?

The Cray Scientific Libraries package, LibSci, is a collection of numerical routines optimized for best performance on Cray systems. When possible, you should use calls to the Cray LibSci routines in your code in place of calls to public-domain or user-written versions.

The Cray LibSci collection contains the following libraries:

- BLAS (Basic Linear Algebra Subroutines, including routines from the University of Texas 64-bit libGoto library);
- BLACS (Basic Linear Algebra Communication Subprograms);
- LAPACK (Linear Algebra Routines, including routines from the University of Texas 64-bit libGoto library);
- ScaLAPACK (Scalable LAPACK);
- IRT (Iterative Refinement Toolkit), linear solvers using 32-bit factorizations that preserve accuracy through mixed-precision iterative refinement;
- CRAFFT (Cray Adaptive Fast Fourier Transform Routines);
- FFT (Fast Fourier Transform Routines);
- FFTW2 (the Fastest Fourier Transforms in the West, release 2); and
- FFTW3 (the Fastest Fourier Transforms in the West, release 3).

ACML provides a free set of thoroughly optimized and threaded math routines for HPC, scientific, engineering and related compute-intensive applications. ACML is ideal for weather modeling, computational fluid dynamics, financial analysis, oil and gas applications and more.

ACML consists of the following main components:

- A full implementation of Level 1, 2 and 3 Basic Linear Algebra Subroutines (BLAS), with key routines optimized for high performance on AMD Opteron™ processors.
- A full suite of Linear Algebra (LAPACK) routines. As well as taking advantage of the highly-tuned BLAS kernels, a key set of LAPACK routines has been further optimized to achieve considerably higher performance than standard LAPACK implementations.
- A comprehensive suite of Fast Fourier Transforms (FFTs) in both single-, double-, single-complex and double-complex data types.
- Random Number Generators in both single- and double-precision.

Intel® Math Kernel Library (Intel® MKL) is a computing math library of highly optimized, extensively threaded math routines for applications that require maximum performance. Core math functions include:

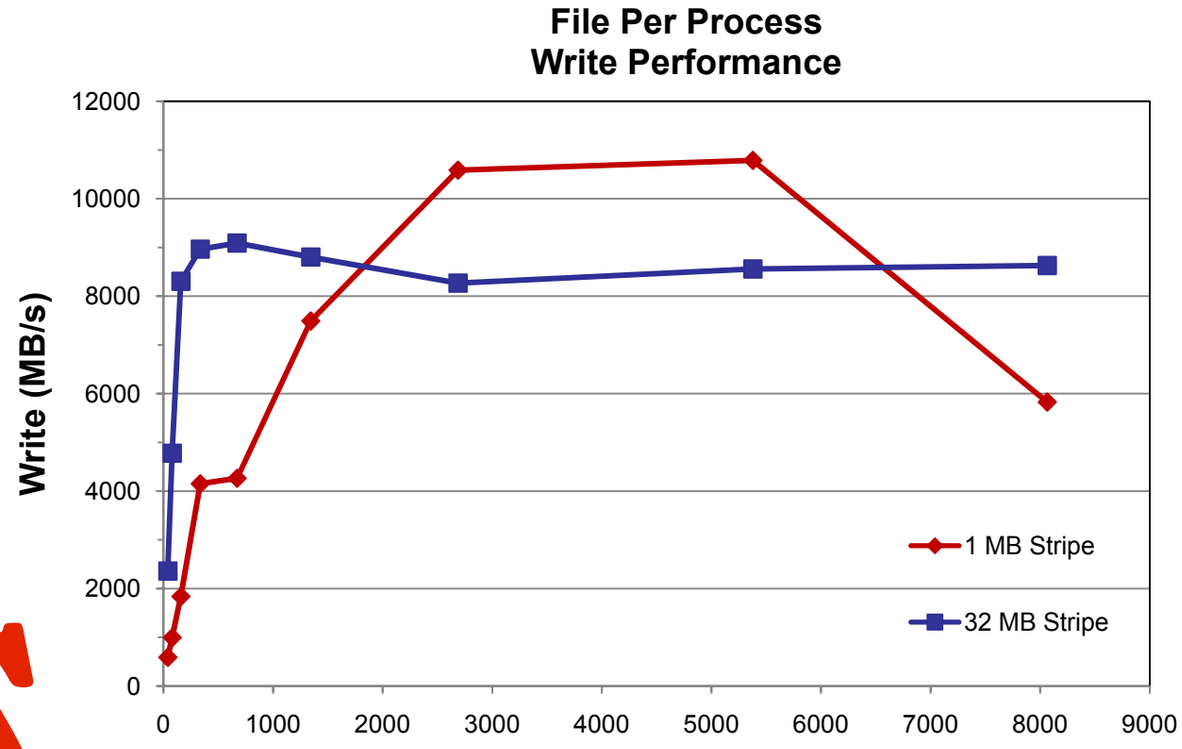
- BLAS,
- LAPACK,
- ScaLAPACK,
- sparse solvers,
- fast Fourier transforms,
- vector math

Anything else?

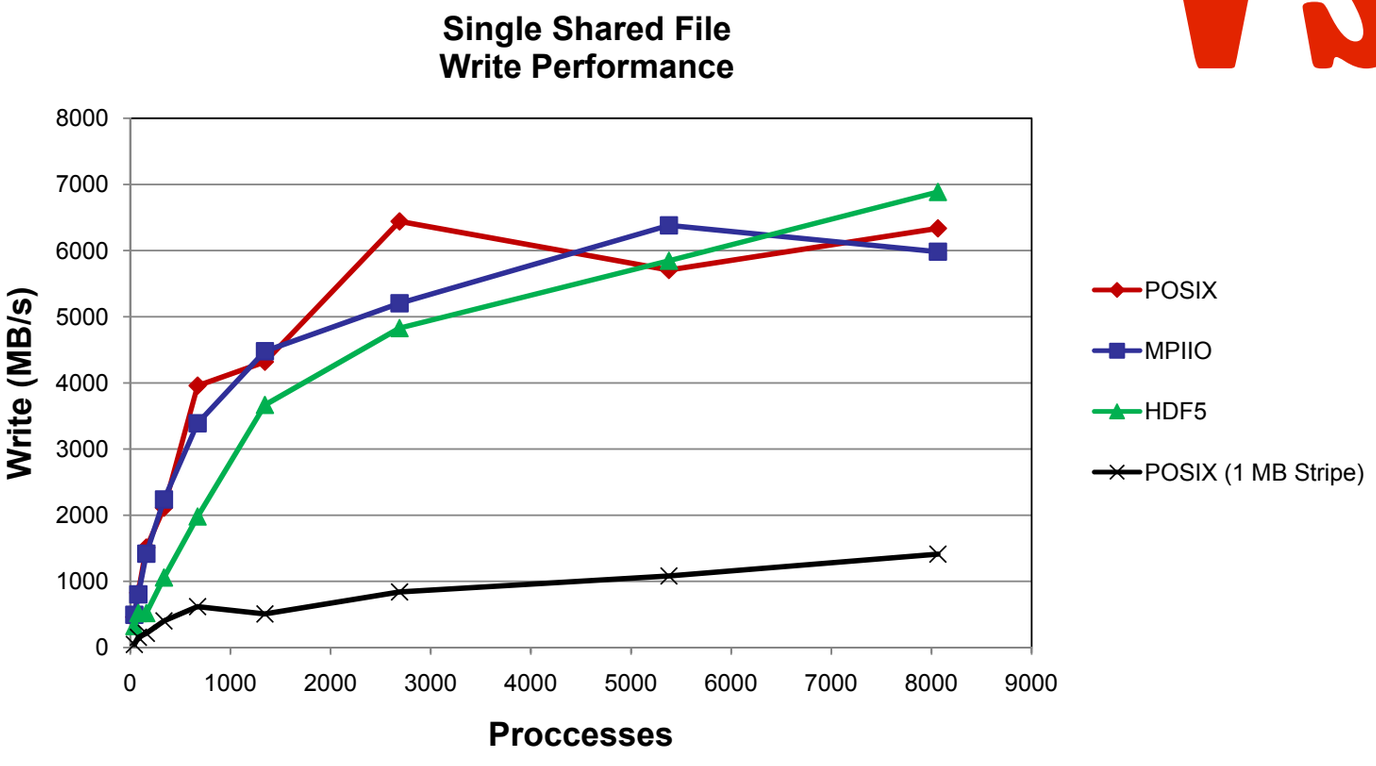
- Debuggers
- Profilers
- Libraries
- I/O
- Power
- Future languages
- Exascale Computing?

File per process

128 MB per file and a 32 MB Transfer size



VS



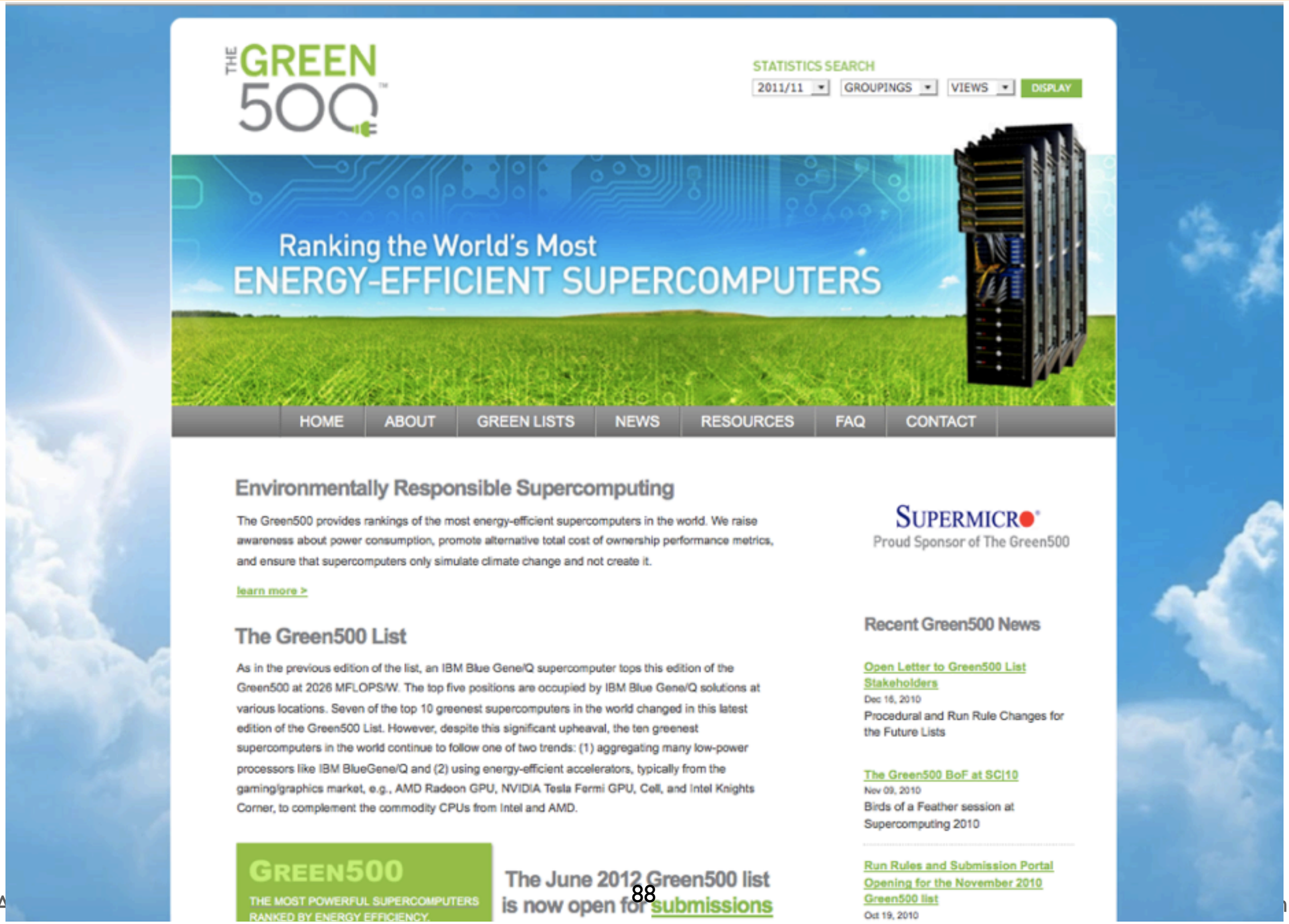
Single shared file

32 MB per process and 32 MB Transfer size

Anything else?

- Debuggers
- Profilers
- Libraries
- I/O
- **Power**
- Future languages
- Exascale Computing?

Power


THE GREEN 500
STATISTICS SEARCH
2011/11
GROUPINGS
VIEWS
DISPLAY
Ranking the World's Most
ENERGY-EFFICIENT SUPERCOMPUTERS
HOME
ABOUT
GREEN LISTS
NEWS
RESOURCES
FAQ
CONTACT
Environmentally Responsible Supercomputing
The Green500 provides rankings of the most energy-efficient supercomputers in the world. We raise awareness about power consumption, promote alternative total cost of ownership performance metrics, and ensure that supercomputers only simulate climate change and not create it.
learn more >
The Green500 List
As in the previous edition of the list, an IBM Blue Gene/Q supercomputer tops this edition of the Green500 at 2026 MFLOPS/W. The top five positions are occupied by IBM Blue Gene/Q solutions at various locations. Seven of the top 10 greenest supercomputers in the world changed in this latest edition of the Green500 List. However, despite this significant upheaval, the ten greenest supercomputers in the world continue to follow one of two trends: (1) aggregating many low-power processors like IBM BlueGene/Q and (2) using energy-efficient accelerators, typically from the gaming/graphics market, e.g., AMD Radeon GPU, NVIDIA Tesla Fermi GPU, Cell, and Intel Knights Corner, to complement the commodity CPUs from Intel and AMD.
SUPERMICRO
Proud Sponsor of The Green500
Recent Green500 News
Open Letter to Green500 List Stakeholders
Dec 16, 2010
Procedural and Run Rule Changes for the Future Lists
The Green500 BoF at SC|10
Nov 09, 2010
Birds of a Feather session at Supercomputing 2010
Run Rules and Submission Portal Opening for the November 2010 Green500 list
Oct 19, 2010
GREEN500
THE MOST POWERFUL SUPERCOMPUTERS RANKED BY ENERGY EFFICIENCY.
The June 2012 Green500 list is now open for submissions

Anything else?

- Debuggers
- Profilers
- Libraries
- I/O
- Power
- **Future languages**
- Exascale Computing?

Global Arrays (GA)

The **Global Arrays** (GA) toolkit provides an efficient and portable "shared-memory" programming interface for distributed-memory computers. Each process in a MIMD parallel program can asynchronously access logical blocks of physically distributed dense multi-dimensional arrays, without need for explicit cooperation by other processes. Unlike other shared-memory environments, the GA model exposes to the programmer the non-uniform memory access (NUMA) characteristics of the high performance computers and acknowledges that access to a remote portion of the shared data is slower than to the local portion. The locality information for the shared data is available, and a direct access to the local portions of shared data is provided.

Global Arrays have been designed to complement rather than substitute for the message-passing programming model. The programmer is free to use both the shared-memory and message-passing paradigms in the same program, and to take advantage of existing message-passing software libraries. Global Arrays are compatible with the Message Passing Interface (MPI).

Unified Parallel C

Unified Parallel C (UPC) is an extension of the C programming language designed for high-performance computing on large-scale parallel machines, including those with a common global address space (SMP and NUMA) and those with distributed memory (e.g. clusters). The programmer is presented with a **single shared, partitioned address space**, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. UPC uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time, typically with a single thread of execution per processor.

UPC example

```
#include <upc.h>
#include <stdio.h>

int main() {
 printf("Hello from thread %d of %d\n", MYTHREAD, THREADS);

 return 0;
}
```

```
$ module swap PrgEnv-pgi PrgEnv-cray
$ cc -o hello-upc hello.upc


$ qsub -I -lsize=12,walltime=00:10:00
$ cd $SCRATCHDIR
$ aprun -n 3 ~/hello-upc
Hello from thread 2 of 3
Hello from thread 0 of 3
Hello from thread 1 of 3
```

Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing unit (CPUs), graphics processing unit (GPUs), and other processors. OpenCL includes a language (based on C99) for writing kernels (functions that execute on OpenCL devices), plus application programming interfaces (APIs) that are used to define and then control the platforms. OpenCL provides parallel computing using task-based and data-based parallelism. OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group. It has been adopted by Intel, Advanced Micro Devices, Nvidia, and ARM Holdings.

Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by Nvidia for graphics processing. CUDA is the computing engine in Nvidia graphics processing units (GPUs) that is accessible to software developers through variants of industry standard programming languages. Programmers use 'C for CUDA' (C with Nvidia extensions and certain restrictions), compiled through a PathScale Open64 C compiler, to code algorithms for execution on the GPU.

OpenACC®

DIRECTIVES FOR ACCELERATORS

The **OpenACC Application Program Interface** describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

The directives and programming model defined in this document allow programmers to create high-level host+accelerator programs without the need to explicitly initialize the accelerator, manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown.

CAPS Innovative software for manycore paradigms

NVIDIA

CRAY THE SUPERCOMPUTER COMPANY

PGI®

- Chapel is a new parallel programming language developed by Cray. It is being developed as part of the Cray Cascade project, a participant in DARPA's High Productivity Computing Systems (HPCS) program, which had the goal of increasing supercomputer productivity by the year 2010.

X10: Performance and Productivity at Scale

IBM Research is developing the open-source X10 programming language to provide a programming model that can address the architectural challenge of multiples cores, hardware accelerators, clusters, and supercomputers in a manner that provides scalable performance in a productive manner. The project leverages over six years of language research funded, in part, by the DARPA/HPCS program.

The X10 programming language is organized around four basic principles of asynchrony, locality, atomicity, and order that are developed on a type-safe, class-based, object-oriented foundation. This foundation is robust enough to support fine-grained concurrency, Cilk-style fork-join programming, GPU programming, SPMD computations, phased computations, active messaging, MPI-style communicators, and cluster programming. X10 implementations are available on Power and x86 clusters, on Linux, AIX, MacOS, Cygwin and Windows.

Fortress

Fortress is a programming language designed for high-performance computing. It was created by Sun Microsystems with funding from DARPA's High Productivity Computing Systems project.

The name "Fortress" is intended to connote a secure Fortran, i.e., "a language for high-performance computation that provides abstraction and type safety on par with modern programming language principles".[1] Its improvements include implicit parallelism, Unicode support and concrete syntax that is similar to mathematical notation. The language is not designed to be similar to Fortran. Syntactically, it most resembles Scala, Standard ML, and Haskell. Fortress is being designed from the outset to have multiple syntactic stylesheets. Source code can be rendered as ASCII text, in Unicode, or as a prettied image. This will allow for support of mathematical symbols and other symbols in the rendered output for easier reading.
Fortress is also designed to be both highly parallel and have rich functionality contained within libraries, drawing from Java but taken to a higher degree. For example, the 'for' loop is a parallel operation, which will not always iterate in a strictly linear manner depending on the underlying software and hardware. However, the 'for' loop is a library function and can be replaced by another 'for' loop of the programmer's liking rather than being built into the language.

Readability, Predictability, Tool-ability, Modularity, Meta-programmability. Runs on JVM	RedHat
Concurrency using Functional programming paradigm. Runs on JVM, CLR, and JavaScript engines	Rich Hickey
A replacement for JavaScript on the browser Faster, easier to maintain, and less susceptible to subtle bugs. Dart VM needs to be compiled – can run on Linux, Mac and Windows	Google

Multi-paradigm: Functional + Imperative + Object-oriented.Runs on CLR and Mono	Microsoft
Portability, support for functional programming and concurrency.Runs on JVM and CLR. Is also compiles to javaScript. Future targets might include Objective-C for the iPhone	Brian Frank, Andy Frank
Compiled with the ease of programming of a dynamic language, concurrency and communication, speed of compilation.Compiler available for Linux, Mac OS X, Windows	Google

Multi-platform support. Compiler for JavaScript, Flash, NekoVM, PHP, C++. C# and Java support is expected	Nicolas Cannasse
Targeted for cloud computing. Client-side UI, server-side logic, and database I/O are all implemented in a single languageRuntime environment own Web server and DBMS. Runs on 64bit Linux and Mac	MLstate
Scalability for multicore and distributed computing. For large team. Multi-paradigm: Functional and O-O. Extensible.Runs on JVM, Android, CLR	EPFL

Aims to be fast, concise, portable, and easy-to-read and support GUI application to an OS kernel.Compiles to ANSI C	Apache
Python is a programming language that lets you work more quickly and integrate your systems more effectively. Python runs on Windows, Linux/Unix, Mac OS X, and has been ported to the Java and .NET virtual machines.	Python Software Foundation
NumPy is the fundamental package for scientific computing with Python. It contains among other things: a powerful N-dimensional array object	

The fundamental library needed for scientific computing with Python is called NumPy. This Open Source library contains:

- a powerful N-dimensional array object
- advanced array slicing methods (to select array elements)
- convenient array reshaping methods

and it even contains 3 libraries with numerical routines:

- basic linear algebra functions
- basic Fourier transforms
- sophisticated random number capabilities

SciPy is an Open Source library of scientific tools for Python. It depends on the NumPy library, and it gathers a variety of high level science and engineering modules together as a single package. SciPy provides modules for

- statistics
- optimization
- numerical integration
- linear algebra
- Fourier transforms
- signal processing
- image processing
- ODE solvers
- special functions

mpi4py

MPI for Python provides bindings of the Message Passing Interface (MPI) standard for the Python programming language, allowing any Python program to exploit multiple processors.

This package is constructed on top of the MPI-1/2 specifications and provides an object oriented interface which closely follows MPI-2 C++ bindings. It supports point-to-point (sends, receives) and collective (broadcasts, scatters, gathers) communications of any picklable Python object, as well as optimized communications of Python object exposing the single-segment buffer interface (NumPy arrays, builtin bytes/string/array objects)

Calculating π

with Python, Mpi4pi and Numpy

Master's code

```
#!/usr/bin/env python
#
# MASTER'S CODE
#
from mpi4py import MPI
import numpy
import sys

comm = MPI.COMM_SELF.Spawn(sys.executable,
                           args=['cpi.py'],
                           maxprocs=5)

N = numpy.array(100, 'i')
comm.Bcast([N, MPI.INT], root=MPI.ROOT)
PI = numpy.array(0.0, 'd')
comm.Reduce(None, [PI, MPI.DOUBLE],
            op=MPI.SUM, root=MPI.ROOT)
print(PI)

comm.Disconnect()
```

Client's code

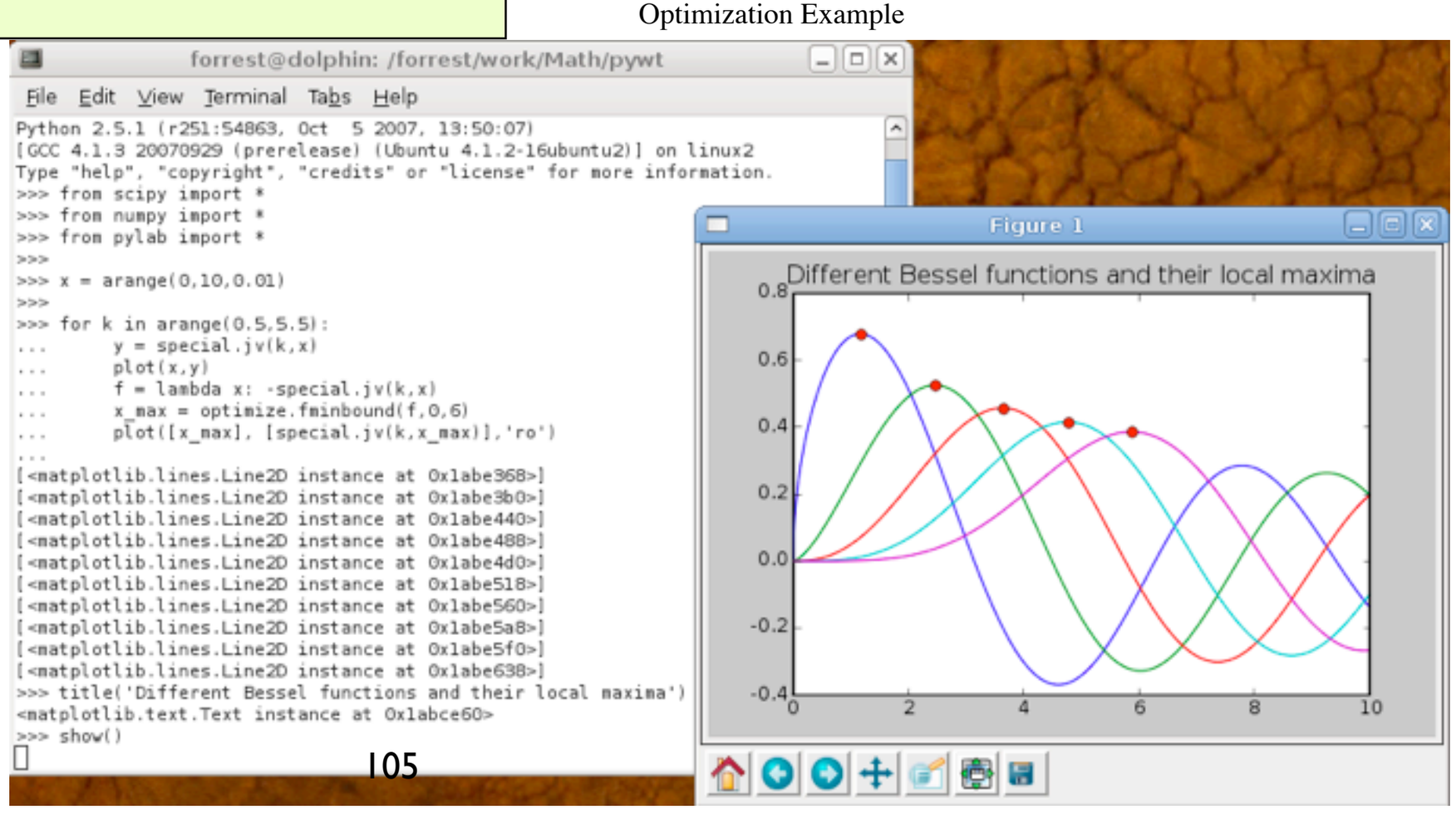
```
#!/usr/bin/env python
#
# CLIENT'S CODE:'cpi.py'
#
from mpi4py import MPI
import numpy

comm = MPI.Comm.Get_parent()
size = comm.Get_size()
rank = comm.Get_rank()

N = numpy.array(0, dtype='i')
comm.Bcast([N, MPI.INT], root=0)
h = 1.0 / N; s = 0.0
for i in range(rank, N, size):
    x = h * (i + 0.5)
    s += 4.0 / (1.0 + x**2)
PI = numpy.array(s * h, dtype='d')
comm.Reduce([PI, MPI.DOUBLE], None,
            op=MPI.SUM, root=0)

comm.Disconnect()
```

Scientific Python

Generating data from several Bessel functions and finding some local maxima using fminbound.

```
 1 from scipy import optimize, special
 2 from numpy import *
 3 from pylab import *
 4
 5 x = arange(0,10,0.01)
 6
 7 for k in arange(0.5,5.5):
 8       y = special.jv(k,x)
 9       plot(x,y)
10       f = lambda x: -special.jv(k,x)
11       x_max = optimize.fminbound(f,0,6)
12       plot([x_max], [special.jv(k,x_max)],'ro')
13
14 title('Different Bessel functions and their local maxima')
15 show()
```

Optimization Example



Anything else?

- Debuggers
- Profilers
- Libraries
- I/O
- Power
- Future languages
- **Exascale Computing?**

Exascale computing

Exascale computing refers to computing capabilities beyond the currently existing petascale. If achieved, it would represent a thousandfold increase over that scale.

The United States has put aside $126 million for exascale computing beginning in 2012.
Three projects aiming at developing technologies and software for Exascale Computing have been started in 2011 within the European Union. The CRESTA project (Collaborative Research into Exascale Systemware, Tools and Applications), the DEEP project (Dynamical ExaScale Entry Platform), and the project Mont-Blanc.

Future challenges?

- **Synchronization-reducing algorithms**
  - ➡ *Break Fork-Join model*
- **Communication-reducing algorithms**
  - ➡ *Use methods which have lower bound on communication*
- **Mixed precision methods**
  - ➡ *2x speed of ops and 2x speed for data movement*
- **Autotuning**
  - ➡ *Today's machines are too complicated, build "smarts" into software to adapt to the hardware*
- **Fault resilient algorithms**
  - ➡ *Implement algorithms that can recover from failures/bit flips*
- **Reproducibility of results**
  - ➡ *Today we can't guarantee this. We understand the issues, but some of our "colleagues" have a hard time with this.*

"Future" Technologies?

- **Most likely a hybrid system?**
  - ➡ *Multicore + Accelerators*
- **Today accelerators are attached via a card with PCI-e**
  - ➡ *Next generation more integrated*
- **Today we have Intel's Knight Ferry**
  - ➡ *Tomorrow Knights corner, 48-80 x86 cores?*

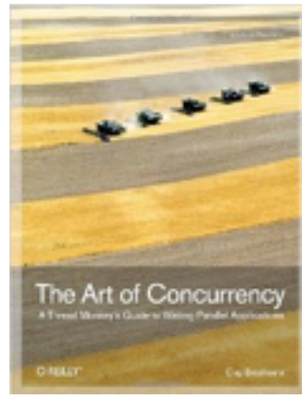
- **AMD's Fusion**
  - ➡ *Multicore with embedded ATI graphics*

- **ARM technologies**
  - ➡ *NVIDIA's Denver project*

Questions?

References

The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications
Clay Breshears



Using OpenMP: Portable Shared Memory Parallel Programming
Barbara Chapman, Gabriele Jost

Using MPI and Using MPI-2
William Gropp , Ewing L. Lusk

Parallel Programming in C with Mpi and Openmp
Michael J. Quinn