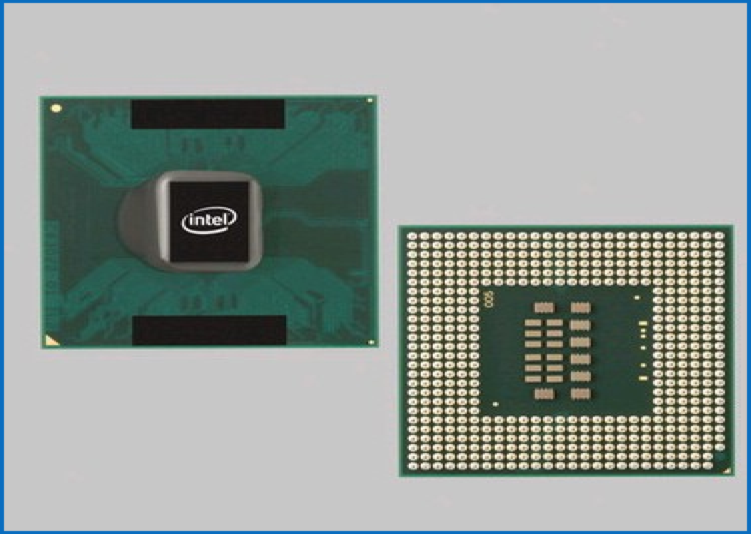


Optimization Techniques For Maximizing Application Performance on Multi-Core Processors

Kittur Ganesh

Agenda

- Multi-core processors – Overview
- Parallelism – impact on multi-core?
- Optimization techniques
- OS Support / SW tools
- Summary

Multi-Core Processors – Overview

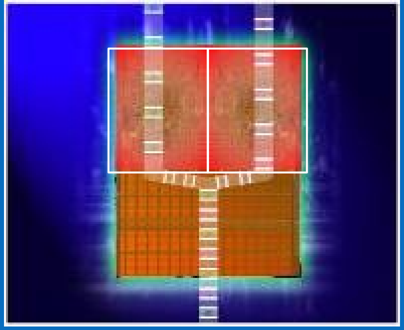
- What are multi-core processors?
  - Integrated circuit (IC) chips containing more than one identical physical processor (core) in the same IC package. OS perceives each core as a discrete processor.
  - Each core has its own complete set of resources, and may share the on-die cache layers
  - Cores may have on-die communication path to front-side bus (FSB)
  - What is a multi processor?

Multi-Core Processors – Overview

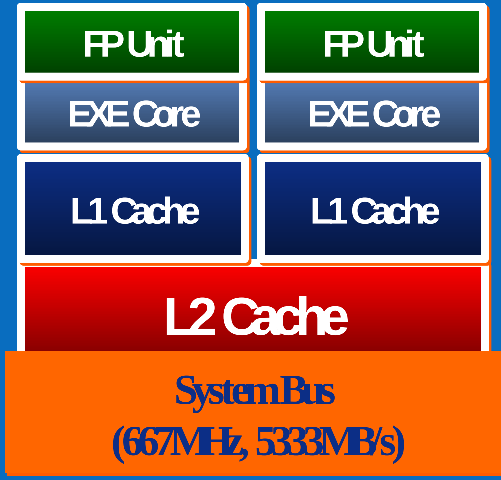
- Multi-core architecture enables divide-and-conquer” strategy to perform more work in a given clock cycle.
- Cores enable thread-level parallelism (multiple instructions / threads per clock cycle)
- Minimizes performance stalls, with a dramatic increase in overall effective system performance
- Greater EEP (energy efficient performance) and scalability

A Dual-core Intel Processor (example)

- Two physical cores in a package
- Each with its own L1 cache
- Each with its own execution resources
- Both cores share the L2 cache
- Truly parallel multi-tasking and threaded execution. Increased throughput.



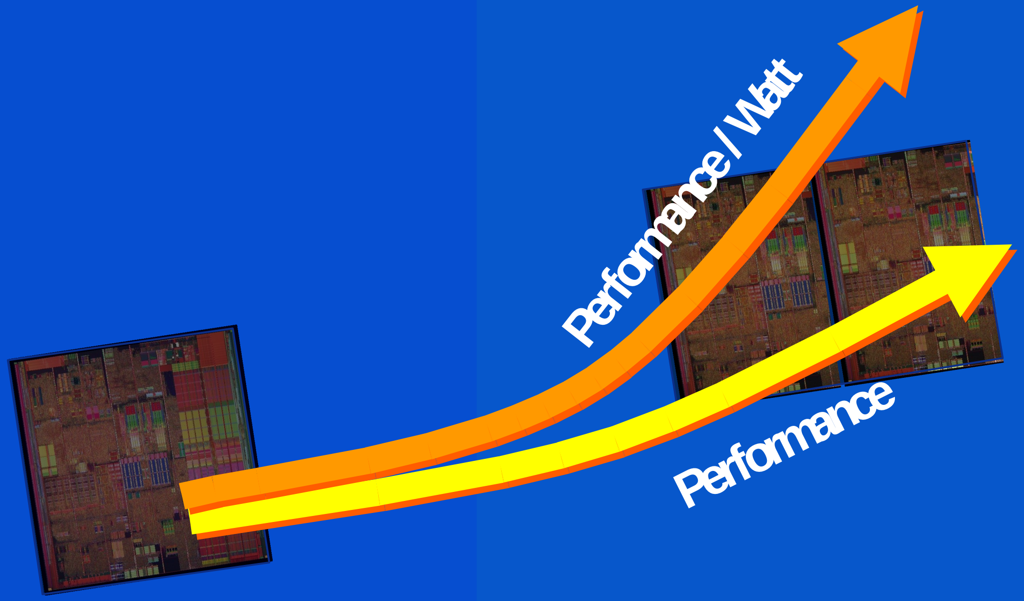
Two Actual Processor Cores



Multi-core Processors – Overview

TODAY

2H'06



Performance / Watt

Performance

Great EEP!
(Energy Efficient Performance)

Over 2X performance*

Driven By Dual Core, Balanced Platform Performance and Lower Power Cores

CPU dies not to scale

* vs. 64bit Intel® Xeon™ Processor based platform (as of May '05)

Parallelism

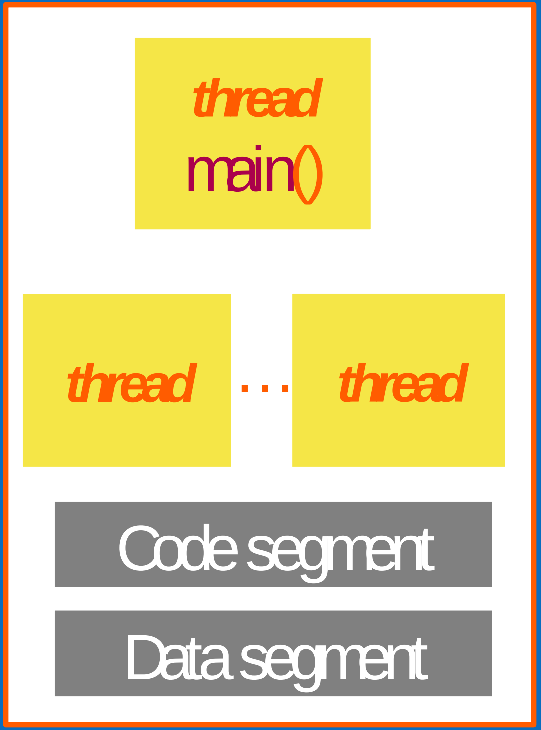
- Power / impact on Multi-core

- Key concepts
  - Processes / Threads
  - Threading – when, why and how?
  - Functional Decomposition
  - Data Decomposition
  - Shared Memory Parallelism
  - Keys to parallelism

Parallelism

- Power / Impact on Multi-core
  - Parallelism is the ability to process multiple instructions, threads or jobs simultaneously per clock cycle, dramatically improving overall performance
  - Multi cores allow full potential for parallelism. An analyst likened this to designing autos with multiple cylinders, each running at optimal power efficiency.
  - Great Energy Efficient Performance, and scalability.

Parallelism – Processes/Threads

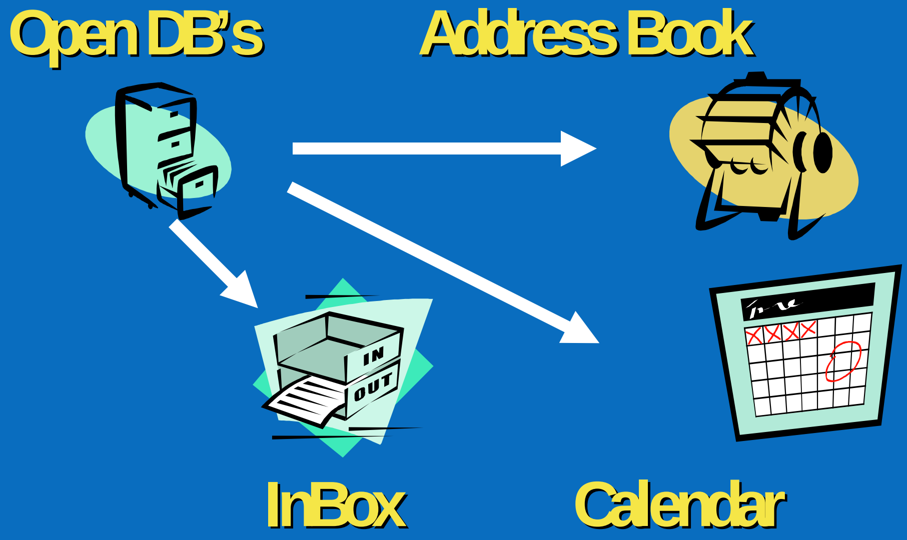


- Modern operating systems load programs as processes
  - Resource holder
  - Execution
- A process starts executing at its entry point as a thread
- Threads can create other threads within the process
- All threads within a process share code & data segments

Parallelism – Threading: When, Why, How

- When to thread?
  - Independent tasks that can execute concurrently
- Why thread?
  - Turnaround or Throughput
- How to thread?
  - Functionality or Performance
- How to define independent tasks?
  - Task or Data decomposition

Functional/Data Decomposition



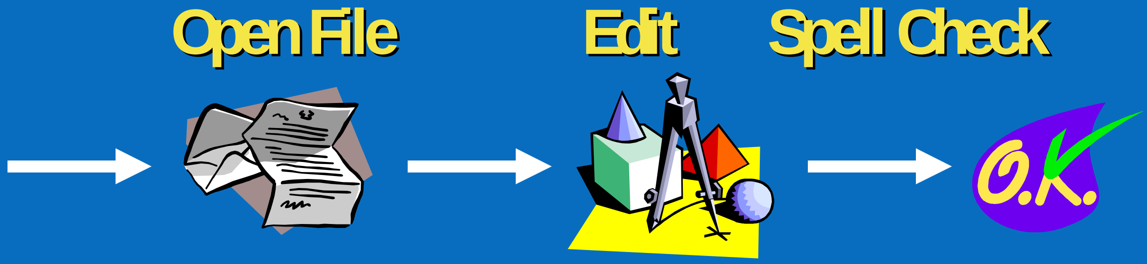
Open DB's

Address Book

InBox

Calendar

Concurrent Tasks



Open File

Edit

Spell Check

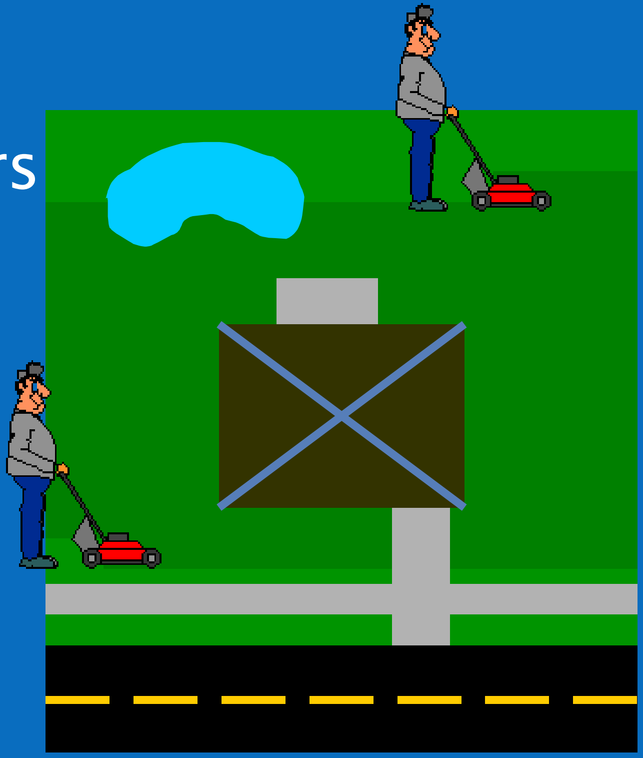
Sequential Tasks

Shared Memory Parallelism

- Multiple threads:
  - Executing concurrently
  - Sharing a single address space
  - Sharing work in coordinated fashion
  - Scheduling handled by OS
- Requires a system that provides shared memory and multiple CPUs

Keys to Parallelism

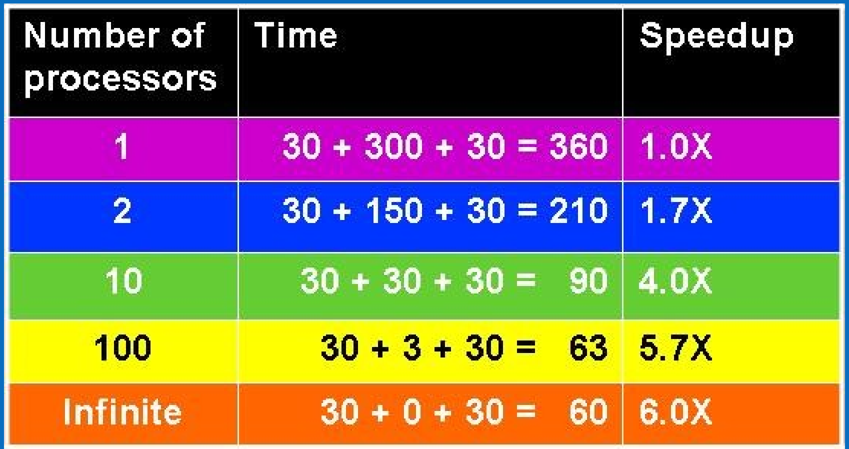
- Identify concurrent work.
- Spread work evenly among workers
- Create private copies of commonly used resources.
- Synchronize access to costly or unique shared resources.



Amdahl's Law – Theoretical Maximum

Speedup of parallel execution

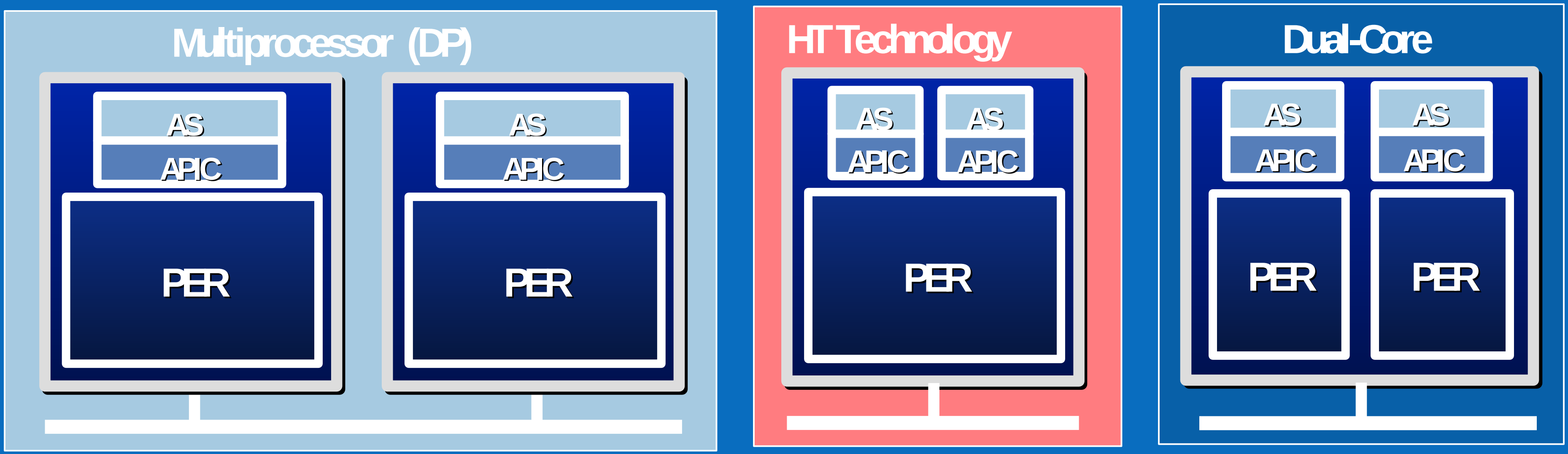
- speedup = 1/(P/N + S)
  - P (parallel code fraction) S (serial code fraction) N (processors)
- Example: Image processing
  - 30 minutes of preparation (serial)
  - One minute to scan a region
  - 30 minutes of cleanup (serial)



Number of processors	Time	Speedup
1	30 + 300 + 30 = 360	1.0X
2	30 + 150 + 30 = 210	1.7X
10	30 + 30 + 30 = 90	4.0X
100	30 + 3 + 30 = 63	5.7X
Infinite	30 + 0 + 30 = 60	6.0X

- Speedup is restricted by serial portion. And, speedup increases with greater number of cores!

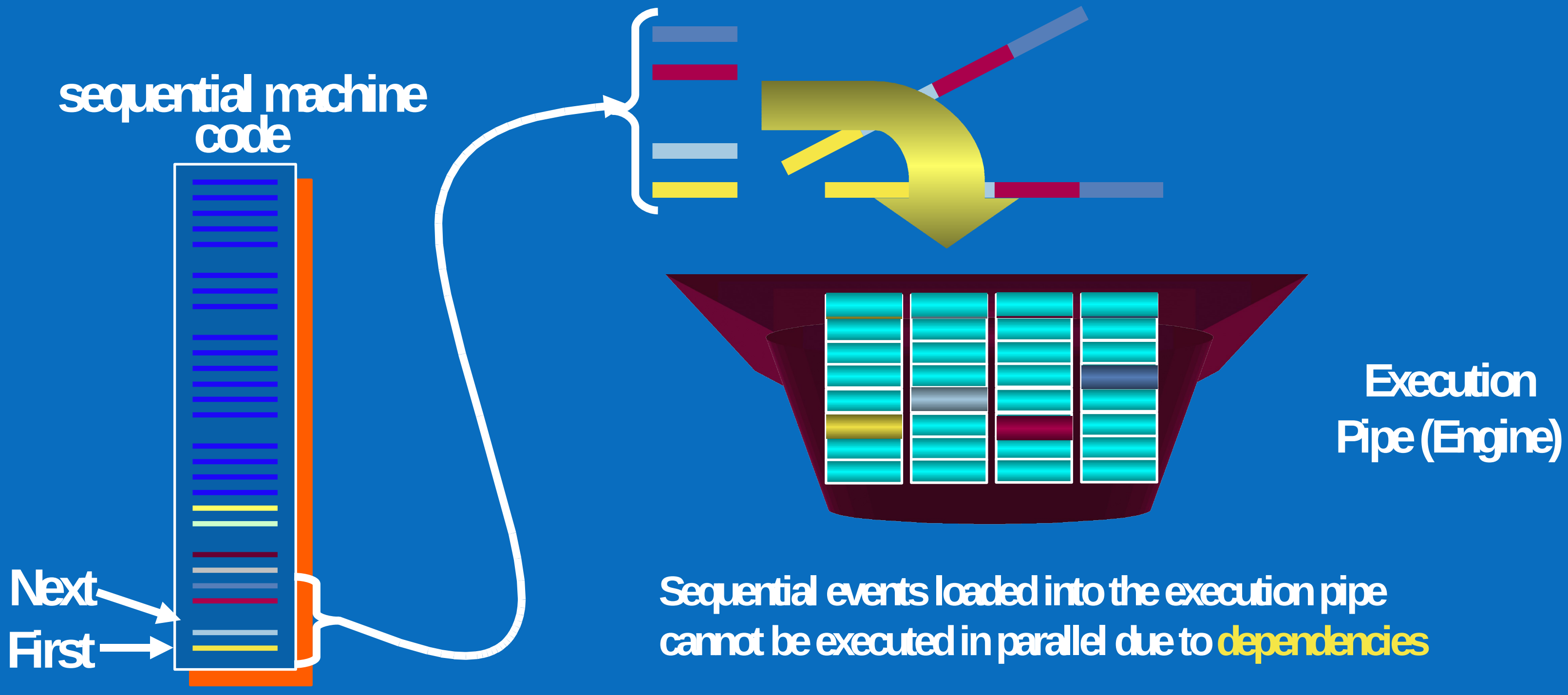
Power of parallelism – seen in Intel Processors



AS = Architecture State (registers, flags, timestamp counter, etc.)
APIC = Advanced Programmable Interrupt Controller
PER = Processor Execution Resources (execution units, instruction decode, etc.)

- Software optimized for DP will perform well on HT Technology and Dual-Core
- Multithreading is required for maximizing application performance
- Single threaded apps will not run faster but benefit while multitasking (running multiple single threaded apps)

Parallelism: machine code execution [Out-of-order execution engine in Duo Core]



Sequential events loaded into the execution pipe cannot be executed in parallel due to dependencies

Instructions are sequential, most instructions depend on completion of the previous instructions

Run time reordering can overcome the dependencies by changing the execution sequence and enable more parallelism

Optimization Techniques

- Multi-core processor implementation (inherent parallelism) has significant impact on software applications
  - Full potential harnessed by programs that migrate to a threaded software model
  - Efficient use of threads (kernel or system / user threads) is KEY to dramatically increase effective system performance

Threaded Software Model

- Explicit Threads
  - Thread Libraries
    - POSIX* threads
    - Win32* API
  - Message Passing Interface (MPI)

- Compiler Directed Threads
  - OpenMP* (portable shared memory parallelism)
  - Auto-parallelization

POSIX* threads

- POSIX.1c standard
- C Language Interface
- Threads exist within the same process
- All threads are peers
  - No explicit parent-child model
  - Exception: main()

Creating POSIX* Threads

```
int pthread_create (
        pthread_t* handle,
        const pthread_attr_t* attributes,
        void *(*function) (void *),
       void* arg );
```

- Function(s) are explicitly mapped to created thread
- Thread handle – holds all related data on created thread.

POSIX* threads – example

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

#define NTHREADS 4

 void test(void *arg) {printf (“Hello, world\n”);}

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

    pthread_t  h[NTHREADS];

    for (int i=0; i<NTHREADS; i++)
        pthread_create (&h[i], NULL, (void *)test, NULL);
 }
```

Message Passing Interface (MPI)

- Message Passing Interface (MPI) is a message passing library standard (based on MPI Forum)
- All parallelism is explicit.
- Supports SMP/Workstation Clusters / heterogeneous networks

MPI – example

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

int main(argc,argv)
int argc;
char *argv[]; {
  int numtasks, rank, rc;
  rc = MPI_Init(&argc,&argv);
  if (rc != MPI_SUCCESS) {
     printf ("Error starting MPI program. Terminating.\n");
     MPI_Abort(MPI_COMM_WORLD, rc);
  }
  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  printf ("Number of tasks= %d My rank= %d\n", numtasks,rank);
  /******* do some work *******/
  MPI_Finalize();
}
```

OpenMP* [www.openmp.org]

An Application Program Interface (API) for multi-threaded, shared memory parallelism

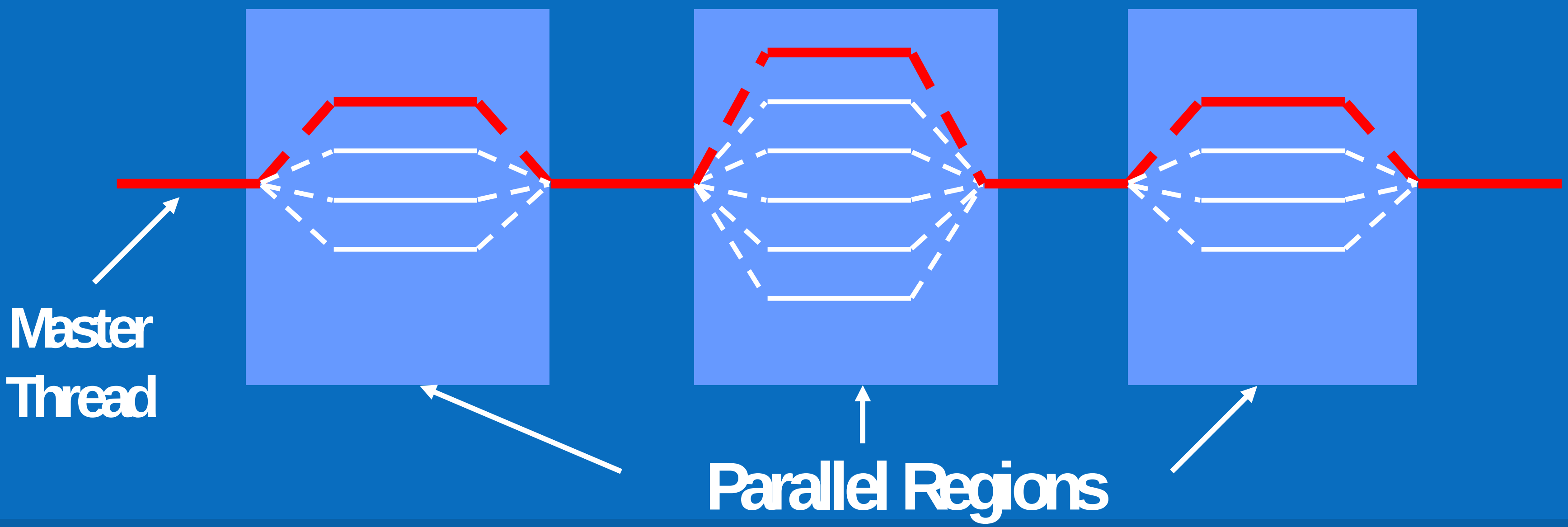
- Portable
  - API for Fortran 77, Fortran 90, C, and C++, on all architectures, including Unix* and Windows*
- Standardized
  - Jointly developed by major SW/HW vendors.
  - Standardizes the last 15 years of symmetric multi-processing (SMP) experience
- Major API components
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables

OpenMP* Programming Model

- Thread Based Parallelism
  - A multi-threaded shared memory process
- Explicit Parallelism
  - OpenMP is an Explicit (not automatic) programming model
  - Programmer has full control over parallelization
- Fork-Join Model
  - Uses fork-join model of parallel execution
- Compiler Directive Based
  - All of OpenMP parallelism is specified through compiler directives imbedded in code.
- Nested Parallelism Support
- Dynamic Threads

Fork – Join Parallelism

- Master thread spawns a team of threads as needed
- Parallelism is added incrementally: i.e., the sequential program evolves into a parallel program



OpenMP* Pragma Syntax

Most constructs in OpenMP* are compiler directives or pragmas.

- For C and C++, the pragmas take the form:

  #pragma omp *construct [clause [clause]…]*

- For Fortran, the directives take the form:

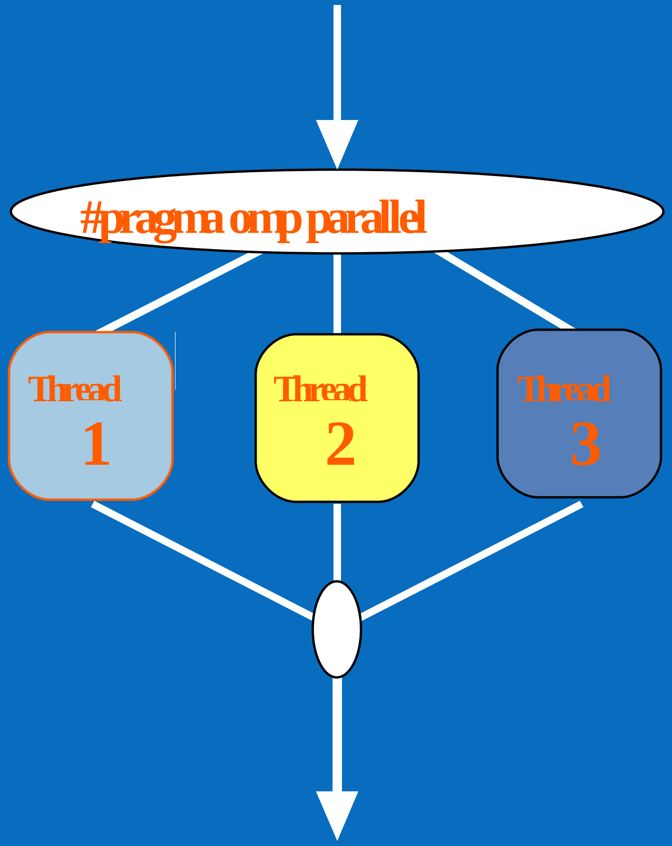
  !$OMP *CONSTRUCT [CLAUSE [CLAUSE]…]*

OpenMP* – parallel region specification

- Defines parallel region over structured block of code
- Threads are created as "parallel" pragma is crossed
- Threads block at end of region
- Data is shared among threads unless specified otherwise

C/C++ :

```
#pragma omp parallel
    {
            block
    }
```

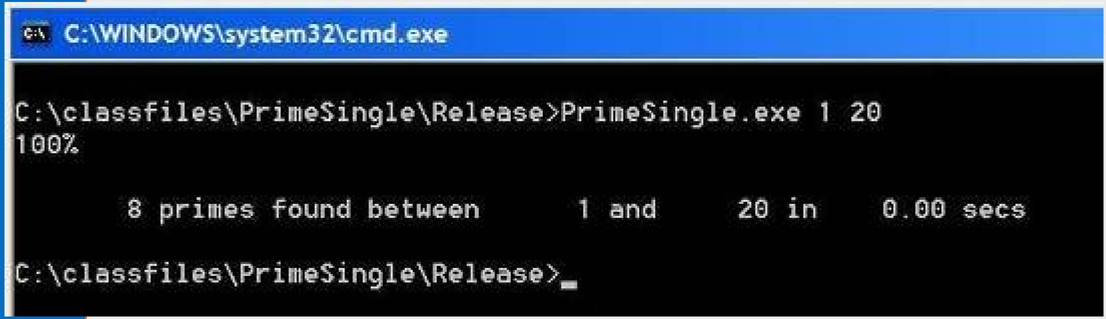


OpenMP* – Example [Prime Number Gen.]

- *Serial Exec.*

i	factor
3	2
5	2
7	23
9	23
11	23
13	234
15	23
17	234
19	234

```
bool TestForPrime(int val)
{   // let's start checking from 3
    int limit, factor = 3;
5f);
factor) )
```



```
C:\classfiles\PrimeSingle\Release>PrimeSingle.exe 1 20
100%

       8 primes found between       1 and      20 in     0.00 secs

C:\classfiles\PrimeSingle\Release>_
```

```
void FindPrimes(int start, int end)
{
```

```
for( int i = start; i <= end; i+= 2 ){
     if( TestForPrime(i) )
          globalPrimes[gPrimesFound++] = i;
     ShowProgress(i, range);
}
```

```
}
```

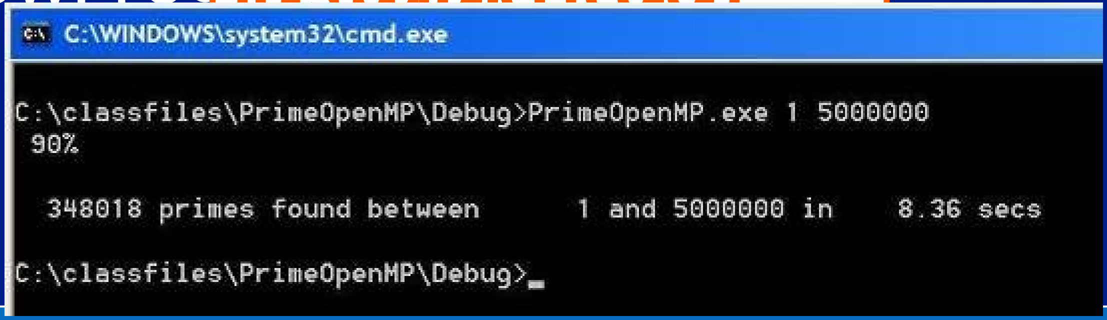
OpenMP* – Example [Prime Number Gen.] -> With OpenMP*

```
#pragma omp parallel for
    for( int i = start; i <= end; i+= 2 ){
                    orPrime(i) )
            globalPrimes[gPrimesFound++] = i;
        ShowProc
    }
```

OpenMP

Defined by the **for** loop

Create threads here for this parallel region



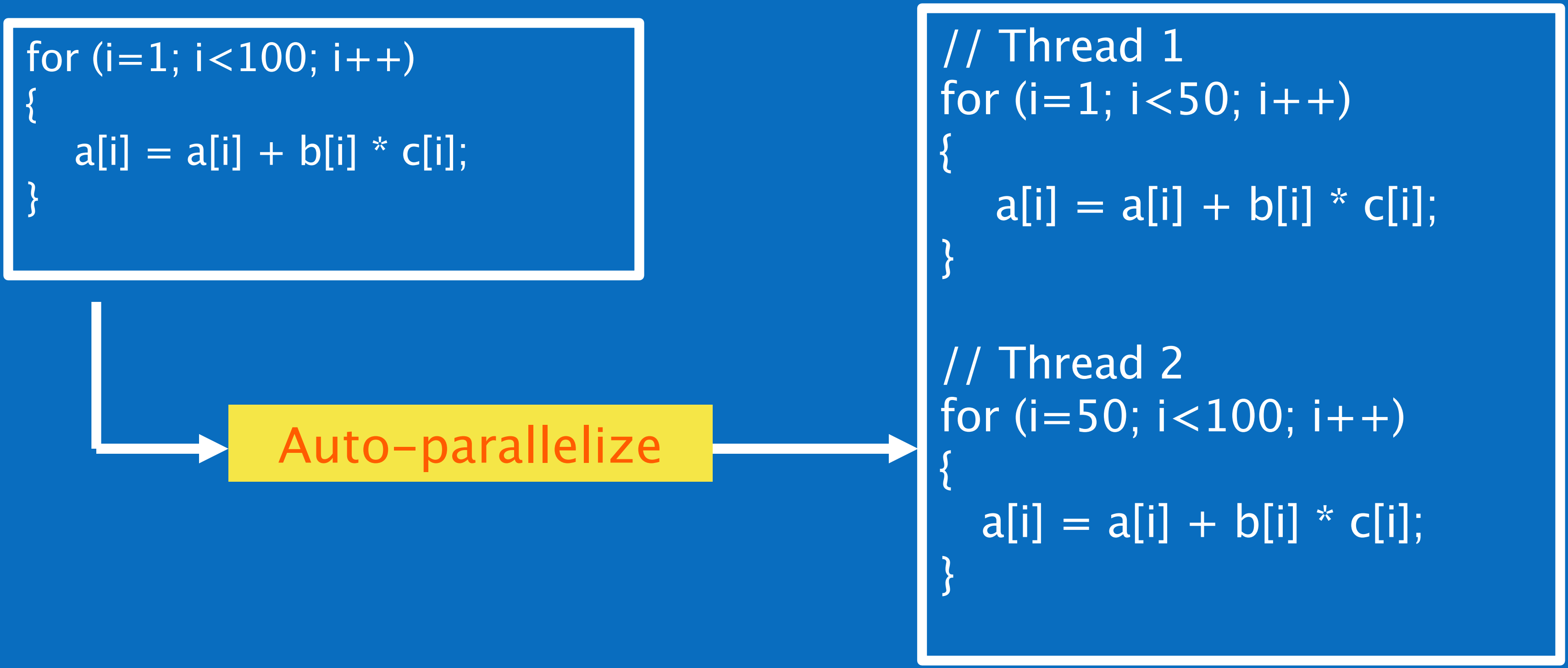
```
C:\classfiles\PrimeOpenMP\Debug>PrimeOpenMP.exe 1 5000000
 90%

  348018 primes found between      1 and 5000000 in    8.36 secs

C:\classfiles\PrimeOpenMP\Debug>_
```

Auto-parallelism

- Auto-parallelism is implicit parallelism.
- The compiler will do automatic threading of loops and other structures, without having to manually insert OpenMP* directives.
- Focus is on loop unrolling and splitting. Loops whose trip counts are known can be parallelized, and no loop carried dependencies exists (read after write, write after read).

NOTE: A loop carried dependence occurs when same memory location is referenced in different iterations of the loop.

Auto-parallelism – Example



```
for (i=1; i<100; i++)
{
    a[i] = a[i] + b[i] * c[i];
}
```

Auto-parallelize

```
// Thread 1
for (i=1; i<50; i++)
{
    a[i] = a[i] + b[i] * c[i];
}

// Thread 2
for (i=50; i<100; i++)
{
   a[i] = a[i] + b[i] * c[i];
}
```

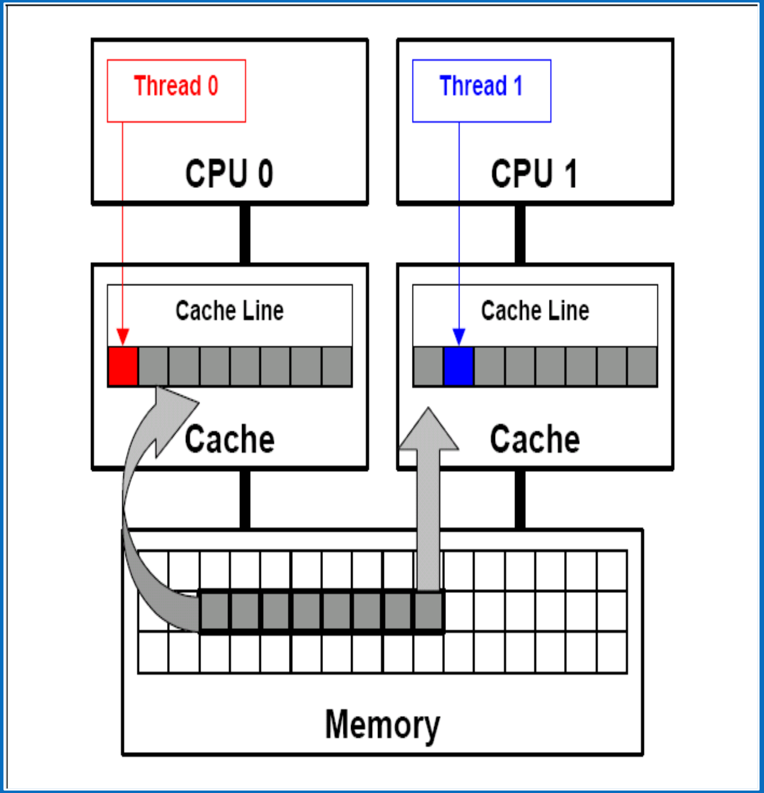
Threading Issues To Deal With

- Data Races
  - Concurrent access of same variable by multiple threads
- Synchronization
  - Share data access must be coordinated
- Thread Stalls
  - Threads wait indefinitely due to dangling locks
- Dead Locks
  - Indefinite wait for resources, caused by locking hierarchy in threads
- False Sharing
  - Threads writing different data on the same cache line

False Sharing – Memory conflict

- Data elements from multiple threads lie on same cache line

- Could cause problem even if threads are not accessing same memory location



Common Performance Issues

- Parallel Overhead
  - Due to thread creation, scheduling
- Synchronization
  - Excessive use of global data, contention for the same synchronization object
- Load Imbalance
  - Improper distribution of parallel work
- Granularity
  - No sufficient parallel work

Parallel Overhead

- Thread Creation overhead
  - Overhead increases rapidly as the number of active threads increases
- Solution
  - Use of re-usable threads and thread pools
    - Amortizes the cost of thread creation
    - Keeps number of active threads relatively constant

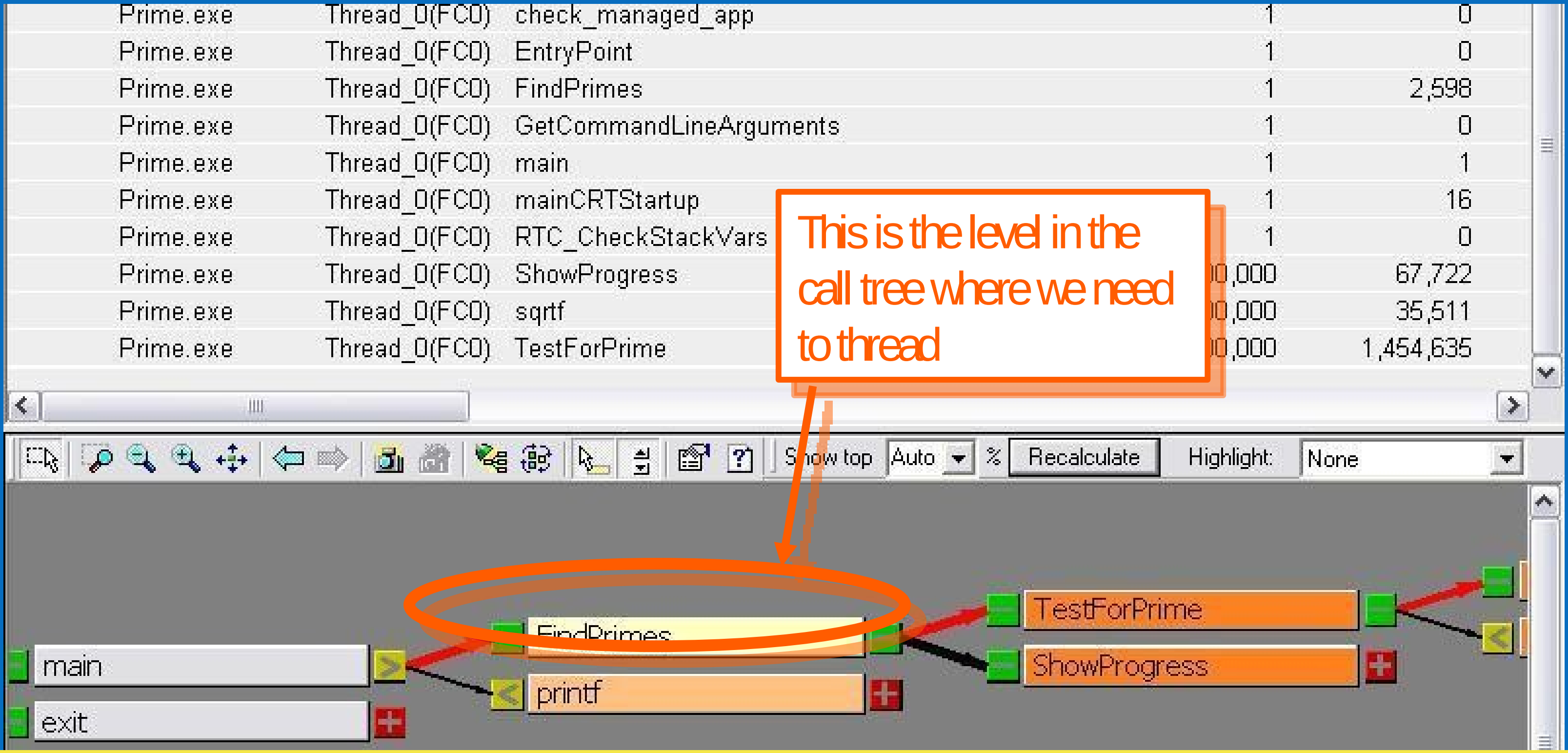
Synchronization

- Heap contention
  - Allocation from heap causes implicit synchronization
  - Allocate on stack or use thread local storage
- Atomic updates versus critical sections
  - Some global data updates can use atomic operations
  - Use atomic updates whenever possible
- Critical Sections vs. Mutual Exclusion API
  - Use CRITICAL SECTION objects when visibility across process boundaries is not required
  - Introduces lesser overhead

Threading tools

- Thread Checker tools
  - Can be used to help debug for correctness of threaded applications
  - Can pin-point notorious threading bugs like data races, thread stalls, deadlocks etc.
- Thread Profiler tools
  - Used for performance tuning to maximize code performance
  - Can pinpoint performance bottlenecks in threaded applications like load imbalance, granularity, load imbalance and synchronization

Example Thread Checker tool: Intel® Thread Checker



Prime.exe	Thread_0(FC0)	check_managed_app	1	0
Prime.exe	Thread_0(FC0)	EntryPoint	1	0
Prime.exe	Thread_0(FC0)	FindPrimes	1	2,598
Prime.exe	Thread_0(FC0)	GetCommandLineArguments	1	0
Prime.exe	Thread_0(FC0)	main	1	1
Prime.exe	Thread_0(FC0)	mainCRTStartup	1	16
Prime.exe	Thread_0(FC0)	RTC_CheckStackVars	1	0
Prime.exe	Thread_0(FC0)	ShowProgress	00,000	67,722
Prime.exe	Thread_0(FC0)	sqrtf	00,000	35,511
Prime.exe	Thread_0(FC0)	TestForPrime	00,000	1,454,635

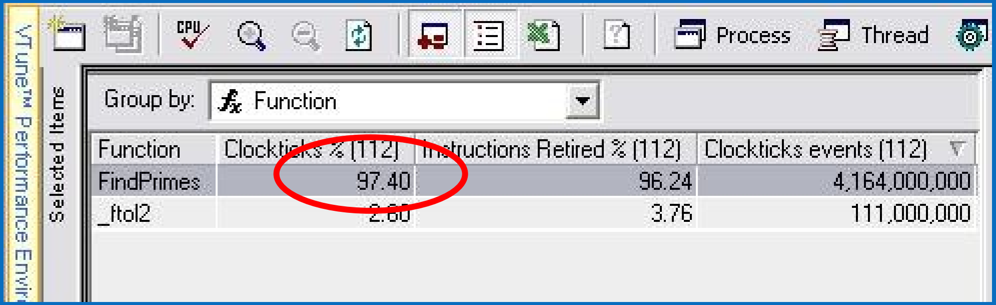
This is the level in the call tree where we need to thread

Identifies time consuming region – finds proper level in call-tree to thread

Example Thread Checker tool: Intel® Thread Checker

Analysis

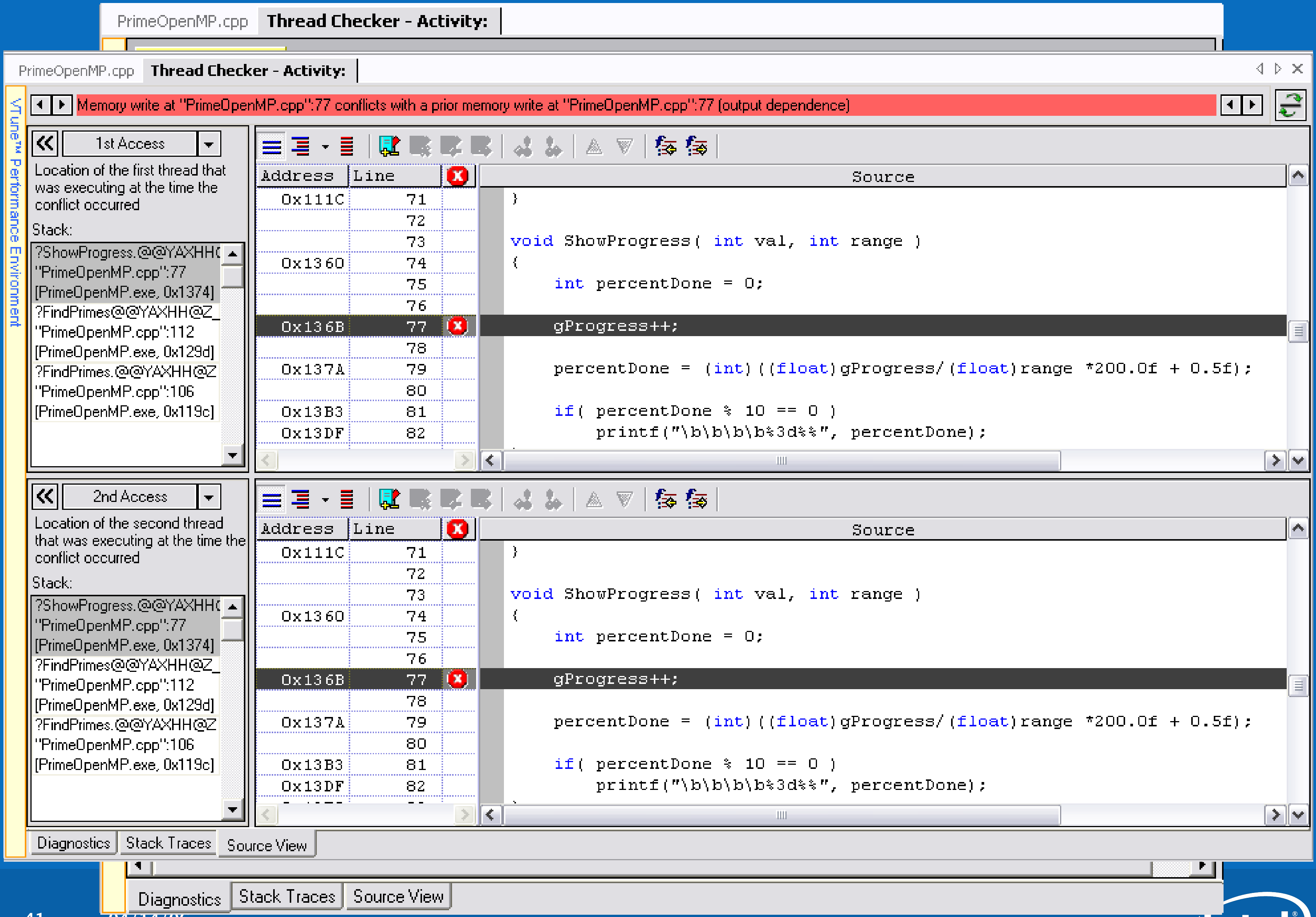
- Where to thread?
  - **`FindPrimes()`**
- Is it worth threading a selected region?



Group by: Function

Function	Clockticks % [112]	Instructions Retired % [112]	Clockticks events [112]
FindPrimes	97.40	96.24	4,164,000,000
_ftol2	2.60	3.76	111,000,000

  - Appears to have minimal dependencies
  - Appears to be data-parallel
  - Consumes over 95% of the run time



PrimeOpenMP.cpp | **Thread Checker - Activity:**

Memory write at "PrimeOpenMP.cpp":77 conflicts with a prior memory write at "PrimeOpenMP.cpp":77 (output dependence)

1st Access

Location of the first thread that was executing at the time the conflict occurred

Stack:

```
?ShowProgress.@@YAXHH(
"PrimeOpenMP.cpp":77
[PrimeOpenMP.exe, 0x1374]
?FindPrimes@@YAXHH@Z_
"PrimeOpenMP.cpp":112
[PrimeOpenMP.exe, 0x129d]
?FindPrimes.@@YAXHH@Z
"PrimeOpenMP.cpp":106
[PrimeOpenMP.exe, 0x119c]
```

Address	Line	☒
0x111C	71	
	72	
	73	
0x1360	74	
	75	
	76	
0x136B	77	☒
	78	
0x137A	79	
	80	
0x13B3	81	
0x13DF	82	

Source

```
}

void ShowProgress( int val, int range )
{
    int percentDone = 0;

    gProgress++;

    percentDone = (int)((float)gProgress/(float)range *200.0f + 0.5f);

    if( percentDone % 10 == 0 )
        printf("\b\b\b\b%3d%%", percentDone);
```

2nd Access

Location of the second thread that was executing at the time the conflict occurred

Stack:

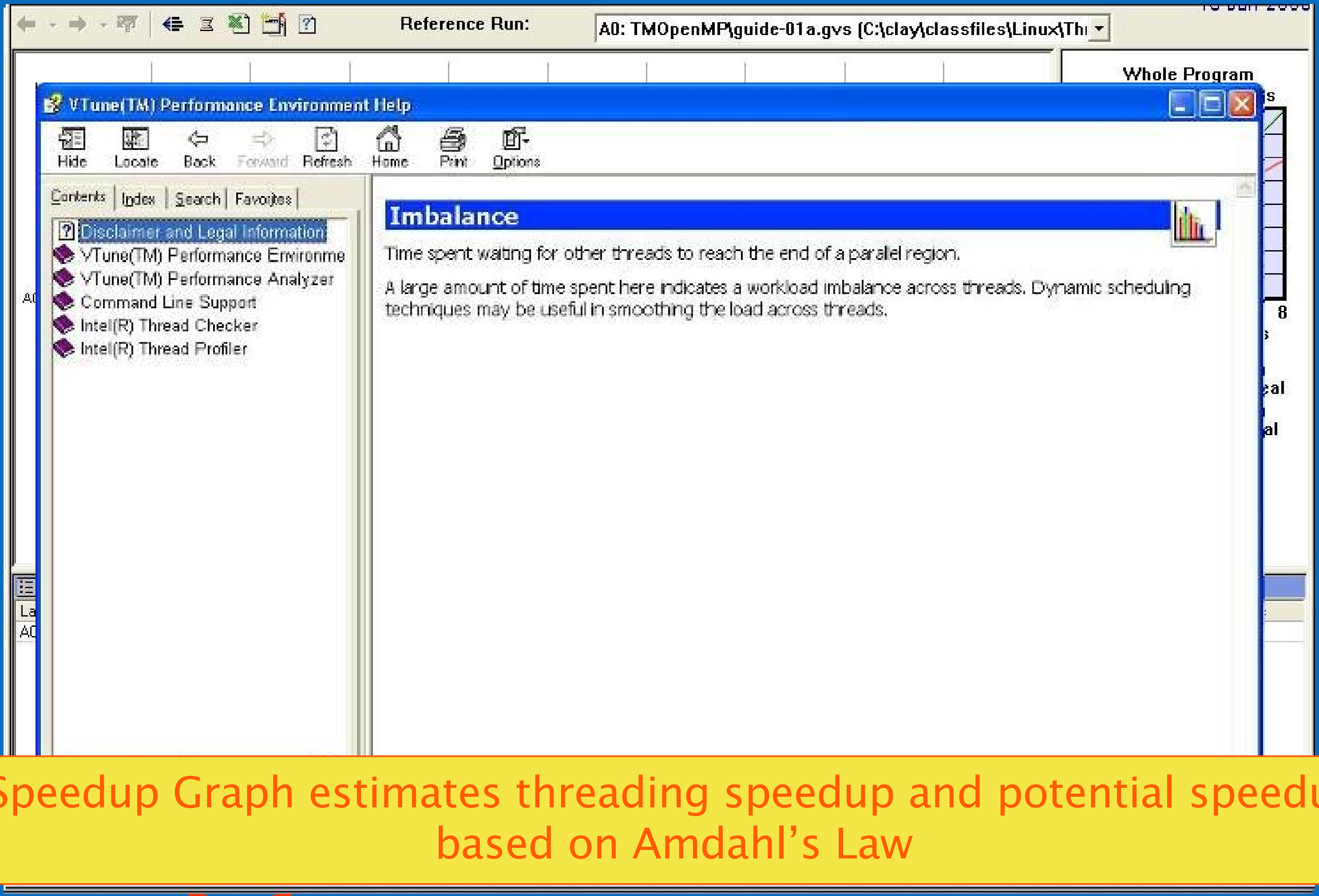
```
?ShowProgress.@@YAXHH(
"PrimeOpenMP.cpp":77
[PrimeOpenMP.exe, 0x1374]
?FindPrimes@@YAXHH@Z_
"PrimeOpenMP.cpp":112
[PrimeOpenMP.exe, 0x129d]
?FindPrimes.@@YAXHH@Z
"PrimeOpenMP.cpp":106
[PrimeOpenMP.exe, 0x119c]
```

Address	Line	☒
0x111C	71	
	72	
	73	
0x1360	74	
	75	
	76	
0x136B	77	☒
	78	
0x137A	79	
	80	
0x13B3	81	
0x13DF	82	

Source

```
}

void ShowProgress( int val, int range )
{
    int percentDone = 0;

    gProgress++;

    percentDone = (int)((float)gProgress/(float)range *200.0f + 0.5f);

    if( percentDone % 10 == 0 )
        printf("\b\b\b\b%3d%%", percentDone);
```

Example Thread Profiler tool: Intel® Thread Profiler for OpenMP



Reference Run:
A0: TMOpenMP\guide-01a.gvs (C:\clay\classfiles\Linux\Th
Whole Program
VTune(TM) Performance Environment Help
Hide
Locate
Back
Forward
Refresh
Home
Print
Options
Contents
Index
Search
Favorites
Disclaimer and Legal Information
VTune(TM) Performance Environme
VTune(TM) Performance Analyzer
Command Line Support
Intel(R) Thread Checker
Intel(R) Thread Profiler

Imbalance

Time spent waiting for other threads to reach the end of a parallel region.

A large amount of time spent here indicates a workload imbalance across threads. Dynamic scheduling techniques may be useful in smoothing the load across threads.

Speedup Graph estimates threading speedup and potential speedup based on Amdahl's Law

Memory Caching / Performance on multi-core systems

- To maximize software performance on multi-core systems, core configurations and memory cache design has to be considered.
- Process resources are shared by threads, and synchronized for data access.
- Multi-core processors share caches, and processor maintains cache coherency
- Cache Memory, and System Memory contains replicated data, and data state is monitored by Cache HW. Cache lines are used for data transfer

Memory Caching – Considerations for maximizing Performance

- Use locking primitives to get true sharing of data between threads, with data synchronization
- Keep few active threads to access data area
- Replicate data copies for use by multi–threads
- Threads feedback data to single thread for updating the shared data
- Create threads sharing data on cores that share cache. Use processor affinity to assign tasks to cores.
- False sharing can degrade performance, so organize data efficiently

OS Support / SW tools

- LINUX* 2.6.16 Kernels have complete support for multi-core (detects cores and enables them), and 2.6.16 -mm tree has multi-core scheduler optimizations too
- Intel® C++ Compiler for Linux*, / Windows*
  - Supports OpenMP*, Auto-parallelism, designed to support and optimize for dual-core and multi-core processors
- Intel® Thread Checker
  - Pinpoints notorious threading bugs like data races, stalls, and deadlocks
- Intel® Thread Profiler
  - Identifies performance issues in threaded applications, and pinpoints performance bottle-necks affecting execution time
- Intel® VTune Performance Analyzer
  - Identifies and characterizes performance issues.

Summary

- Multi-Core processors enable true thread level parallelism with great Energy Efficient Performance, and Scalability
- To utilize the full potential of multi-core processors, SW applications will need to move from a single to a multi-threaded model.
- Optimization techniques like OpenMP*, Auto-parallelization, cache coherency are key to maximizing performance
- A SW application should not just be threaded, but should be designed to be a well-threaded application for maximizing performance on multi-core processors.

Unleash the power of multi-core!

BACK-UP