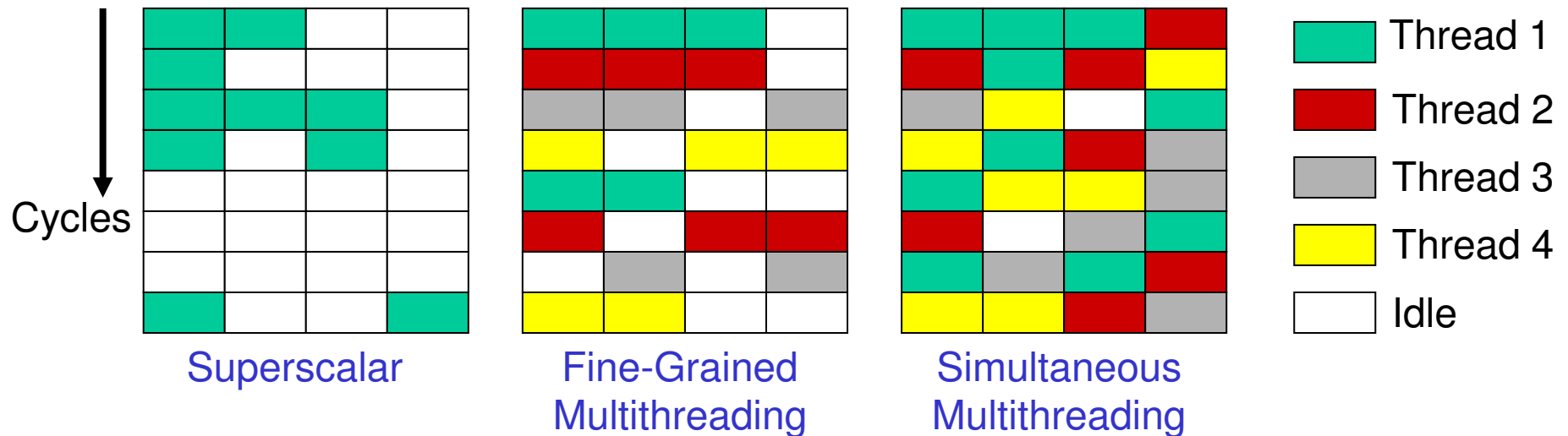


Thread-Level Parallelism

- Motivation:
 - a single thread leaves a processor under-utilized for most of the time
 - by doubling processor area, single thread performance barely improves
- Strategies for thread-level parallelism:
 - multiple threads share the same large processor → reduces under-utilization, efficient resource allocation
Simultaneous Multi-Threading (SMT)
 - each thread executes on its own mini processor → simple design, low interference between threads
Chip Multi-Processing (CMP) or multi-core

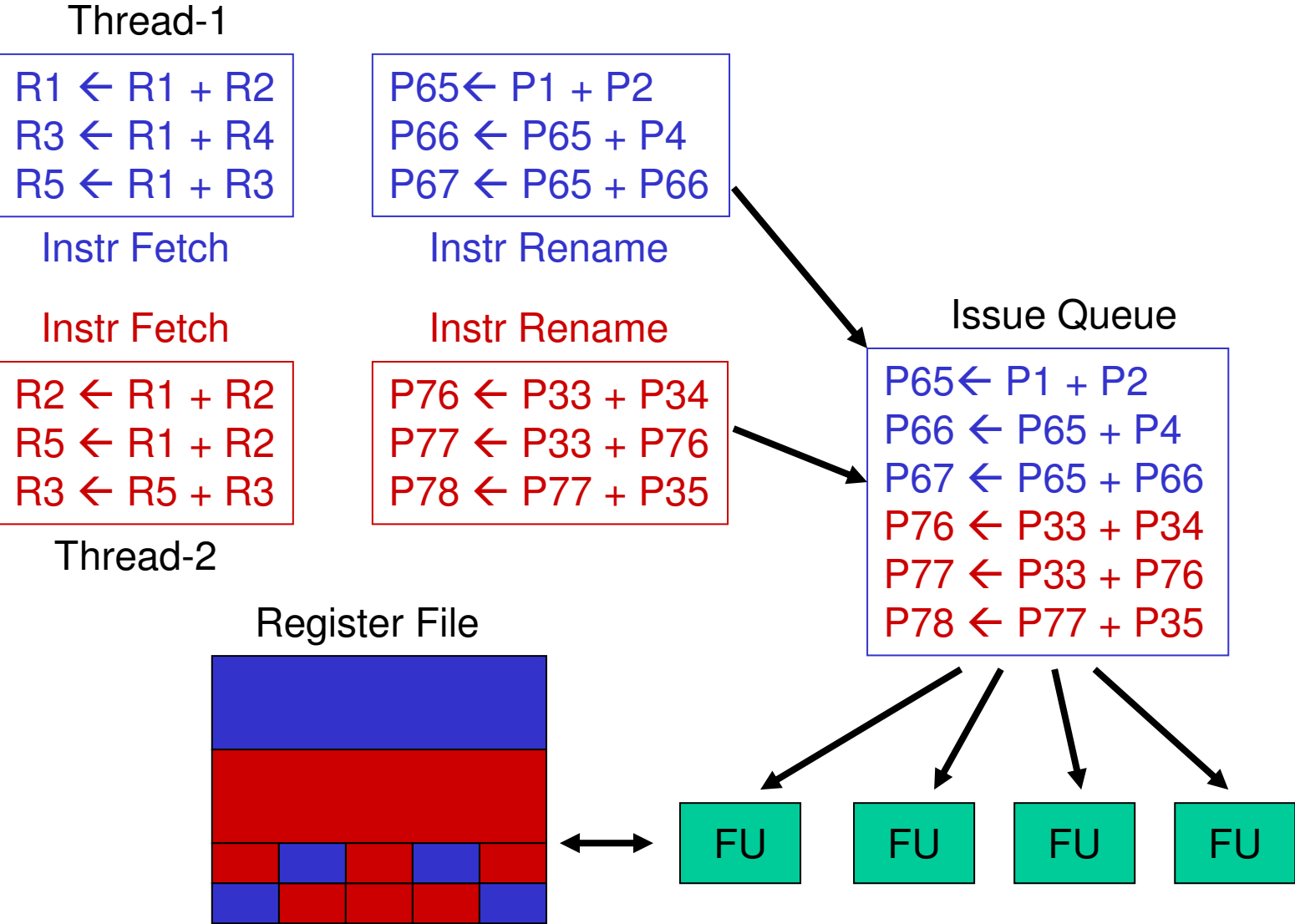
How are Resources Shared?

Each box represents an issue slot for a functional unit. Peak thruput is 4 IPC.



- Superscalar processor has high under-utilization – not enough work every cycle, especially when there is a cache miss
- Fine-grained multithreading can only issue instructions from a single thread in a cycle – can not find max work every cycle, but cache misses can be tolerated
- Simultaneous multithreading can issue instructions from any thread every cycle – has the highest probability of finding work for every issue slot

Resource Sharing



Performance Implications of SMT

- Single thread performance is likely to go down (caches, branch predictors, registers, etc. are shared) – this effect can be mitigated by trying to prioritize one thread
- While fetching instructions, thread priority can dramatically influence total throughput – a widely accepted heuristic (ICOUNT): fetch such that each thread has an equal share of processor resources
- With eight threads in a processor with many resources, SMT yields throughput improvements of roughly 2-4

Multi-Programmed Speedup

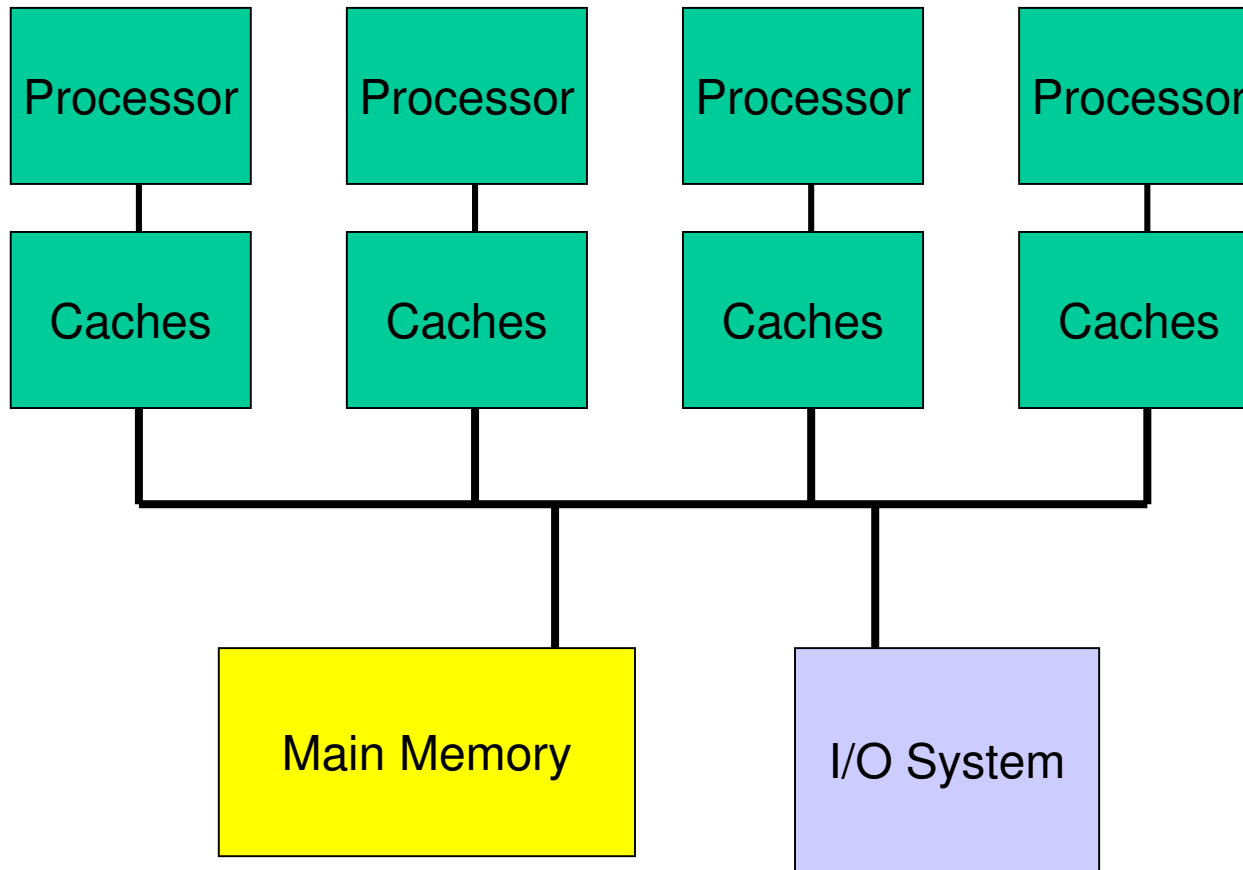
Benchmark	Best Speedup	Worst Speedup	Avg Speedup
gzip	1.48	1.14	1.24
vpr	1.43	1.04	1.17
gcc	1.44	1.00	1.11
mcf	1.57	1.01	1.21
crafty	1.40	0.99	1.17
parser	1.44	1.09	1.18
eon	1.42	1.07	1.25
perlbnk	1.40	1.07	1.20
gap	1.43	1.17	1.25
vortex	1.41	1.01	1.13
bzip2	1.47	1.15	1.24
twolf	1.48	1.02	1.16
wupwise	1.33	1.12	1.24
swim	1.58	0.90	1.13
mgrid	1.28	0.94	1.10
aplu	1.37	1.02	1.16
mesa	1.39	1.11	1.22
galgel	1.47	1.05	1.25
art	1.55	0.90	1.13
quake	1.48	1.02	1.21
facerec	1.39	1.16	1.25
ammp	1.40	1.09	1.21
lucas	1.36	0.97	1.13
fma3d	1.34	1.13	1.20
sixtrack	1.58	1.28	1.42
apsi	1.40	1.14	1.23
Overall	1.58	0.90	1.20

- sixtrack and eon do not degrade their partners (small working sets?)
- swim and art degrade their partners (cache contention?)
- Best combination: swim & sixtrack
worst combination: swim & art
- Static partitioning ensures low interference – worst slowdown is 0.9

Multiprocs -- Memory Organization - I

- Centralized shared-memory multiprocessor or Symmetric shared-memory multiprocessor (SMP)
- Multiple processors connected to a single centralized memory – since all processors see the same memory organization → uniform memory access (UMA)
- Shared-memory because all processors can access the entire memory address space
- Can centralized memory emerge as a bandwidth bottleneck? – not if you have large caches and employ fewer than a dozen processors

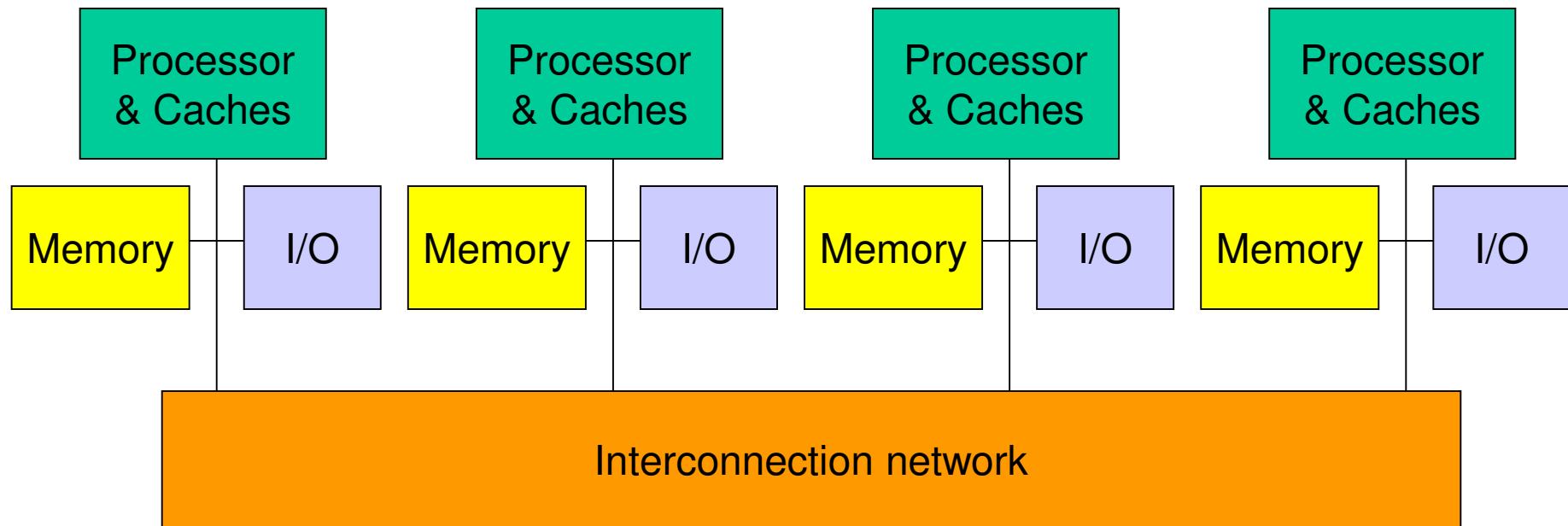
SMPs or Centralized Shared-Memory



Multiprocs -- Memory Organization - II

- For higher scalability, memory is distributed among processors → distributed memory multiprocessors
- If one processor can directly address the memory local to another processor, the address space is shared → distributed shared-memory (DSM) multiprocessor
- If memories are strictly local, we need messages to communicate data → cluster of computers or multicomputers
- Non-uniform memory architecture (NUMA) since local memory has lower latency than remote memory

Distributed Memory Multiprocessors



Shared-Memory Vs. Message-Passing

Shared-memory:

- Well-understood programming model
- Communication is implicit and hardware handles protection
- Hardware-controlled caching

Message-passing:

- No cache coherence → simpler hardware
- Explicit communication → easier for the programmer to restructure code
- Sender can initiate data transfer

Ocean Kernel

```
Procedure Solve(A)
begin
  diff = done = 0;
  while (!done) do
    diff = 0;
    for i ← 1 to n do
      for j ← 1 to n do
        temp = A[i,j];
        A[i,j] ← 0.2 * (A[i,j] + neighbors);
        diff += abs(A[i,j] - temp);
      end for
    end for
    if (diff < TOL) then done = 1;
  end while
end procedure
```

Shared Address Space Model

```
int n, nprocs;
float **A, diff;
LOCKDEC(diff_lock);
BARDEC(bar1);

main()
begin
  read(n); read(nprocs);
  A ← G_MALLOC();
  initialize (A);
  CREATE (nprocs,Solve,A);
  WAIT_FOR_END (nprocs);
end main
```

```
procedure Solve(A)
  int i, j, pid, done=0;
  float temp, mydiff=0;
  int mymin = 1 + (pid * n/nprocs);
  int mymax = mymin + n/nprocs -1;
  while (!done) do
    mydiff = diff = 0;
    BARRIER(bar1,nprocs);
    for i ← mymin to mymax
      for j ← 1 to n do
        ...
      endfor
    endfor
    LOCK(diff_lock);
    diff += mydiff;
    UNLOCK(diff_lock);
    BARRIER (bar1, nprocs);
    if (diff < TOL) then done = 1;
    BARRIER (bar1, nprocs);
  endwhile
```

Message Passing Model

```
main()
  read(n); read(nprocs);
  CREATE (nprocs-1, Solve);
  Solve();
  WAIT_FOR_END (nprocs-1);

procedure Solve()
  int i, j, pid, nn = n/nprocs, done=0;
  float temp, tempdiff, mydiff = 0;
  myA ← malloc(...)
  initialize(myA);
  while (!done) do
    mydiff = 0;
    if (pid != 0)
      SEND(&myA[1,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      SEND(&myA[nn,0], n, pid+1, ROW);
    if (pid != 0)
      RECEIVE(&myA[0,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
```

```
    for i ← 1 to nn do
      for j ← 1 to n do
        ...
      endfor
    endfor
  endfor
  if (pid != 0)
    SEND(mydiff, 1, 0, DIFF);
    RECEIVE(done, 1, 0, DONE);
  else
    for i ← 1 to nprocs-1 do
      RECEIVE(tempdiff, 1, *, DIFF);
      mydiff += tempdiff;
    endfor
    if (mydiff < TOL) done = 1;
    for i ← 1 to nprocs-1 do
      SEND(done, 1, I, DONE);
    endfor
  endif
endwhile
```

Course outline (Pacheco; GGKK; Quinn)

- Motivation (1;1;1)
- How to quantify performance improvement (2.6; 5; 7)
- **Parallel hardware architecture (2.2-2.3; 2,4; 2)**
- Parallel programming frameworks
 - Pthreads for shared memory (4; 7; -)
 - OpenMP for shared memory (5; 7.10; 17)
 - MPI for distributed memory (3; 6; 4)
 - CUDA/OpenCL for GPU,
 - Hadoop/Spark/Mapreduce for distributed systems
- Parallel program verification
- Parallel algorithm design
- Some case studies