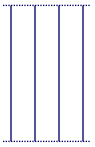
RTS – Provided Services

- **Common RTS provide a subset of the following** (not limited to)
  - Parallelism
    - Type of parallelism (API)
    - Threading Model (underlying implementation)
  - Communication
  - Synchronization
  - Consistency
  - Scheduling
  - Dynamic Load Balancing
  - Memory Management
  - Parallel I/O
- **Some functionalities are only provided as a thin abstraction layer on top of the OS service**

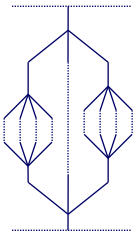
RTS – Flat Parallelism



Parallelism types – Flat Parallelism

- All threads of execution have the same status
  - No parent/child relationship
- Threads are active during the whole execution
- Usually constant number of threads of execution
- Well adapted for problems with large granularity
- Difficult to achieve load-balance for non-embarrassingly parallel applications
- E.g. MPI

RTS – Nested Parallelism

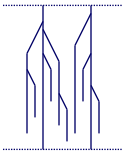


Parallelism types – Nested Parallelism

- Parallelism is hierarchal
  - Threads of execution can spawn new threads to execute their task
  - Exploits multiple levels of parallelism (e.g. nested parallel loops)
- Good affinity with heterogeneous architectures (e.g. clusters of SMPs)*
  - Allows the exploitation of different levels of granularity
- Natural fit for composed parallel data structures*
  - E.g. `p_vector< p_list< Type > >`
- E.g. OpenMP, Cilk, TBB

* Also for dynamic parallelism.

RTS – Dynamic Parallelism



Parallelism types – Dynamic Parallelism

- Threads of execution are dynamically created whenever new parallelism is available
  - Exploits any granularity of parallelism available
  - Necessary to achieve scalability for dynamic applications
- Improves load-balancing for dynamic applications
  - Work stealing
  - Thread migration
- Parallelism can be dynamically refined (e.g. mesh refinement*)
- E.g. STAPL, Charm++, AMPI, Chapel

* Can also be achieved by redistributing the data.

RTS – Threading Models (1:1)

1:1 threading model: (1 user-level thread mapped onto 1 kernel thread)

- Default kernel scheduling
  - Possibility to give hints to scheduler (e.g., thread priority levels)
  - Reduced optimization opportunities
- Heavy kernel threads
  - Creation, destruction and swapping are expensive
  - Scheduling requires to cross into kernel space
- E.g., Pthreads

RTS – Threading Models (M:1)

M:1 threading model: (M user-level threads mapped onto 1 kernel thread)

- Customizable scheduling
  - Enables scheduler-based optimizations (e.g. priority scheduling, good affinity with latency hiding schemes)
- Light user-level threads
  - Lesser threading cost
    - User-level thread scheduling requires no kernel trap
- Problem: no effective parallelism
  - User-level threads' execution serialized on 1 kernel thread
  - Often poor integration with the OS (little or no communication)
  - E.g., GNU Portable Threads

RTS – Threading Models (M:N)

M:N threading model: (M user-level threads mapped onto N kernel threads)

- Customizable scheduling
  - Enables scheduler-based optimizations (e.g. priority scheduling, better support for relaxing the consistency model …)
- Light user-level threads
  - Lesser threading cost
    - Can match N with the number of available hardware threads : no kernel-thread swapping, no preemption, no kernel over-scheduling …
    - User-level thread scheduling requires no kernel trap
  - Perfect and free load balancing within the node
    - User-level threads are cooperatively scheduled on the available kernel threads (they migrate freely).
- E.g., PM2/Marcel

RTS – Communication

- Systems usually provide low-level communication primitives
  - Not practical for implementing high-level libraries
  - Complexity of development leads to mistakes
- Often based on other RTS libraries
  - Layered design conceptually based on the historic ISO/OSI stack
  - OSI layer-4 (end-to-end connections and reliability) or layer-5 (inter-host communication)
  - Communication data is not structured
  - E.g., MPI, Active Message, SHMEM
- **Objective:** Provide structured communication
  - OSI layer-6 (data representation) – data is structured (type)
  - E.g., RMI, RPC

RTS – Synchronization

- Systems usually provide low-level synchronization primitives (e.g., semaphores)
  - Impractical for implementing high-level libraries
  - Complexity of development leads to mistakes
- Often based on other RTS libraries
  - E.g., POSIX Threads, MPI …
- Objective: Provide appropriate synchronization primitives
  - Shared Memory synchronization
    - E.g., Critical sections, locks, monitors, barriers …
  - Distributed Memory synchronization
    - E.g., Global locks, fences, barriers …

RTS – Consistency

- In shared memory systems
  - Use system's consistency model
  - Difficult to improve performance in this way
- In distributed systems: relaxed consistency models
  - Processor Consistency
    - Accesses from a processor on another's memory are sequential
    - Limited increase in level of parallelism
  - Object Consistency
    - Accesses to different objects can happen out of order
    - Uncovers fine-grained parallelism
      - Accesses to different objects are concurrent
      - Potential gain in scalability

RTS – Scheduling

- Available for RTS providing some user-level threading (M:1 or M:N)
- Performance improvement
  - Threads can be cooperatively scheduled (no preemption)
  - Swapping does not require to cross into kernel space
- Automatically handled by RTS
- Provide API for user-designed scheduling

RTS – Dynamic Load Balancing

- Available for RTS providing some user-level threading (M:1 or M:N)
- User-level threads can be migrated
  - Push: the node decides to offload part of its work on another
  - Pull: when the node idles, it takes work from others (work stealing)
- For the M:N threading model
  - Perfect load balance within the node (e.g., dynamic queue scheduling of user-level threads on kernel threads)
  - Free within the node (I.e., no additional cost to simple scheduling)

RTS – Memory Management

- RTS often provide some form of memory management
  - Reentrant memory allocation/deallocation primitives
  - Memory reuse
  - Garbage collection
  - Reference counting
- In distributed memory
  - Can provide Global Address Space
    - Map every thread's virtual memory in a unique location
  - Provide for transparent usage of RDMA engines

RTS – Parallel I/O

- I/O is often the bottleneck for scientific applications processing vast amounts of data
- Parallel applications require parallel I/O support
  - Provide abstract view to file systems
  - Allow for efficient I/O operations
  - Avoid contention, especially in collective I/O
- E.g., ROMIO implementation for MPI-IO
- Archive of current Parallel I/O research:
  http://www.cs.dartmouth.edu/pario/
- List of current projects:
  http://www.cs.dartmouth.edu/pario/projects.html

RTS – Portability / Abstraction

- Fundamental role of runtime systems
  - Provide unique API to parallel programming libraries/languages
  - Hide discrepancies between features supported on different systems
- Additional layer of abstraction
  - Reduces complexity
  - Encapsulates usage of low-level primitives for communication and synchronization
- Improved performance
  - Executes in user space
  - Access to application information allows for optimizations

References

- Hill, M. D. 1998. Multiprocessors Should Support Simple Memory-Consistency Models. Computer 31, 8 (Aug. 1998), 28-34. DOI=http://dx.doi.org/10.1109/2.707614
- Adve, S. V. and Gharachorloo, K. 1996. Shared Memory Consistency Models: A Tutorial. Computer 29, 12 (Dec. 1996), 66-76. DOI=http://dx.doi.org/10.1109/2.546611
- Dictionary on Parallel Input/Output, by Heinz Stockinger, February 1998.

Table of Contents

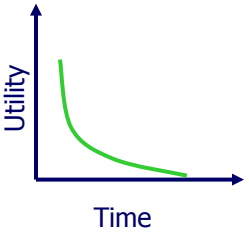
Productivity

- Reduce time to solution
  - Programming time + execution time
- Reduce cost of solution
- Function of:
  - Problem solved *P*
  - System used *S*
  - Utility function *U*

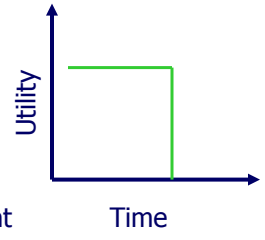
$$\Psi = \Psi(P, S, U)$$

Utility Functions

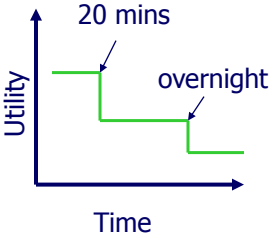
- Decreasing in time.



- Extreme example: deadline driven

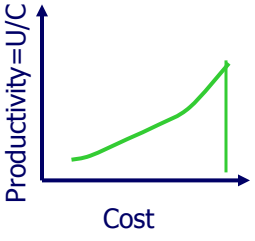


- Practical approximation: staircase



Simple Example

- Assume deadline-driven Utility and decreasing Cost



- Max productivity achieved by solving problem just fast enough to match deadline
- Need to account for uncertainty

Programming Model Impact

- Features try to reduce development time
  - Expressiveness
  - Level of abstraction
  - Component Reuse
  - Expandability
  - Base language
  - Debugging capability
  - Tuning capability
  - Machine model
  - Interoperability with other languages
- Impact on performance examined separately

Expressive

Programming model's ability to express solution in:

- The closest way to the original problem formulation
- A clear, natural, intuitive, and concise way
- In terms of other solved (sub)problems

Definition from http://lml.ls.fi.upm.es/~jjmoreno/expre.html

Level of Abstraction

- Amount of complexity exposed to developer

MATLAB

```
% a and b are matrices
c = a * b;
```

STAPL

```
// a and b are matrices
Matrix<double> c = a * b;
```

C

```
/* a and b are matrices */
double c[10][10];
int i, j, k;
for(int i=0; i<10; ++i) {
  for(int k=0; k<10; ++k) {
    for(int j=0; j<10; ++j) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Component Reuse

- Goal: Increase reuse to reduce development time
- Programming model provides component libraries

STAPL pContainers and pAlgorithms

```
p_vector<double> x(100);
p_vector<double> y(100);

p_generate(x, rand);
p_generate(y, rand);

double result = p_inner_product(x,y);
```

Expandable

- Programming model provides a subset of components needed for a parallel application.
- Expansion enabled by:
  - Transparent components
  - Compositional construction