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

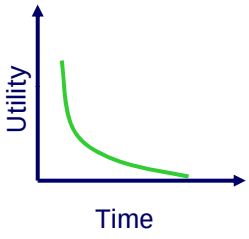
Productivity

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

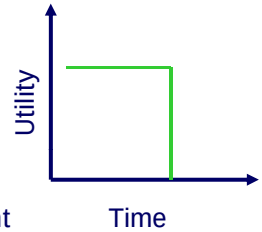
$$\Psi = \Psi(P, S, U)$$

Utility Functions

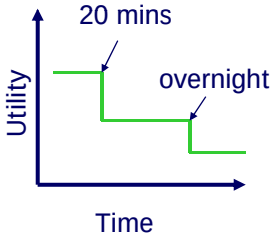
- Decreasing in time.



- Extreme example: deadline driven

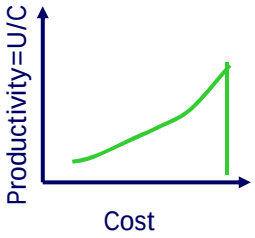


- Practical approximation: staircase



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

Component Transparency

- Opaque objects hide implementation details
  - raises level of abstraction
  - makes expansion difficult
- Transparent components
  - allow internal component reuse
  - example of working in programming model

```
int main() {
  pthread_t thread;
  pthread_attr_t attr;
  // …
}
```

```
template<class T>
class p_array : public p_container_indexed<T> {
  typedef p_container_indexed<T> base_type;
  size_t m_size;
  //…
};
```

Component Composition

Build a new component using building blocks.

```
template<typename View>
bool p_next_permutation(View& vw) {
  …
  reverse_view<View> rvw(vw);
  iter1 = p_adjacent_find(rvw);
  …
  iter2 = p_find_if(rvw, std::bind1st(pred,*iter1));
  …
  p_reverse(rvw);
  return true;
}
```

Programming Language

- Programming model language options:
  - provide a new language
  - extend an existing language
  - provide directives for an existing language
  - use an existing language

Fortress

```
component HelloWorld
  export Executable

  run()=do
    print "Hello, world!\n“
  end
end
```

Cilk

```
cilk void hello() {
  printf(“Hello, world!\n”);
}

int main() {
  spawn hello();
  sync;
}
```

Providing a new language

- Advantage
  - Complete control of level of abstraction
  - Parallel constructs embedded in language
- Disadvantage
  - Compiler required for every target platform
  - Developers must learn language

Fortress

```
component HelloWorld
  export Executable

  run()=do
    print "Hello, world!\n“
  end
end
```

Extending a language

- Advantage
  - Developers have less to learn
  - Complete control of level of abstraction
  - Parallel constructs embedded in syntax
- Disadvantage
  - Compiler required for every target system
  - Limited by constraints of base language

```
cilk void hello() {
  printf("Hello, world!\n");
}
int main() {
  spawn hello();
  sync;
}
```

Directives for a language

- Advantage
  - Developers have less to learn
  - Parallel constructs easily expressed in directives
  - Use available compilers if needed (no parallelization)
  - Specialized not necessarily needed on system
- Disadvantage
  - Compiler required for every target system
  - Higher levels of abstraction can't be achieved
  - Limited by constraints of base language
  - No composition

```
#pragma omp parallel for
for(int i=0; i<N; ++i) {
  C[i] = A[i]*B[i];
}
```

Library for a language

- Advantage
  - Developers learn only new API
  - Compilers available on more systems
- Disadvantage
  - Limited by constraints of base language

```
void* hello(void*) {
  printf("Hello, world!\n");
  pthread_exit(NULL);
}
```

```
int main() {
  pthread_t thread;
  pthread_attr_t attr;
  pthread_attr_init(&attr);

  pthread_create(&thread, &attr,
   hello, NULL);
}
```

Debuggable

Programming environments provide many options for debugging parallel applications.

Built-in	provides proprietary tools that utilize extra runtime information	Charm++
Tracing	provides hooks for tools to log state during execution	MPI, Charm++
Interoperability with standard tools	Leverage standard tools available on platform (e.g., gdb, totalview)	STAPL, TBB, Pthreads, MPI, OpenMP

Defect Management

- Reduce Defect Potential
  - Programming style reduces likelihood of errors
  - Use of container methods reduces out-of-bounds accesses

```
class tbb_work_function {
  void operator()(const blocked_range<size_t>& r) {
    for(size_t i = r.begin(); i != r.end(); ++i)
      C[i] = A[i]*B[i];
  }
};
```

- Provide Defect Detection
  - Components support options to detect errors at runtime
  - E.g., PTHREAD_MUTEX_ERRORCHECK enables detection of double-locking and unnecessary unlocking

Tunability

Programming environments support application optimization on a platform using:

- Performance Monitoring
  - Support measuring application metrics
- Implementation Refinement
  - Support for adaptive/automatic modification of application
  - Manual mechanisms provided to allow developer to implement refinement

Performance Monitoring

- Built-in support
  - Environment's components instrumented
  - Output of monitors enabled/disabled by developer
  - Components written by developer can use same instrumentation interfaces
- Interoperable with performance monitoring tools
  - Performance tools on a platform instrument binaries

Implementation Refinement

- Adjust implementation to improve performance
  - distribution of data in a container
  - scheduling of iterations to processors
- Adaptive/Automatic
  - Monitors performance and improves performance without developer intervention
  - Example: Algorithm selection in STAPL
- Manual mechanisms
  - Model provides methods to allow developer adjustment to improve performance
  - Example: Grain size specification to TBB algorithms

Machine Model

- Programming models differ in the amount and type of machine information available to user
  - TBB, Cilk, OpenMP: user unaware of number of threads
  - MPI: user required to write code as a function of the machine in order to manage data mapping
- Programming as a function of the machine
  - Lowers level of abstraction
  - Increases programming complexity

Interoperability with other models

- Projects would like to use multiple models
  - Use best fit for each application module
  - Modules need data from one another
- Models need flexible data placement requirements
  - Avoid copying data between modules
  - Copying is correct, but expensive
- Models need generic interfaces
  - Components can interact if interfaces meet requirements
  - Avoids inheriting complex hierarchy when designing new components