Teaching people how to “think parallel”

Tim Mattson

Principal Engineer
Application Research Laboratory
Intel Corp
Disclaimer

- This is not an “Intel talk”.
- The views expressed in this presentation are my own and do not represent the views of Intel.
- Also note that these slides have not been formally reviewed to verify proper use of trademarks, references to third party products, or other legal-issues.
- So please, if there are any problems, blame me and not Intel.
Agenda

- Many core motivation slides
- Psychology of programming
- Design Patterns and Pattern Languages
- Berkeley patterns effort
Future processors:
How many cores is “many core”

Dual Core (2006)

Quad-Core (2007)

Tera-scale research chip:
1 SP TFLOP* for 97 W
~100 Million transistors

Think big ... dozens or even hundreds of cores is not too far down the road.

80-Core (2007)

*single precision TFLOP with the Stencil application kernel. Note: this is a research chip and not a product
If this is going to be a general purpose CPU, we need “all software” to be parallel. Where will the parallel software come from?

This is an architecture concept that may or may not be reflected in future products from Intel Corp.
We need to educate a new generation of parallel programmers

- Parallel Programming is difficult, error prone and only accessible to small cadre of experts “do it” ... Clearly we haven’t been doing a very good job at educating programmers.
- Consider the following:
  - All known programmers are human beings.
  - Hence, human psychology, not hardware design, should guide our work on this problem.

Teach Parallel programming around the needs of the programmer, not the needs of the computer.
Agenda

- Many core motivation slides
- Psychology of programming
- Design Patterns and Pattern Languages
- Berkeley patterns effort
Cognitive Psychology and human reasoning

- Human Beings are model builders
  - We build hierarchical complexes of mental models.
  - Understand sensory input in terms of these models.
  - When input conflicts with models, we tend to believe the models.
Programming and models

- Programming is a process of successive refinement of a problem over a hierarchy of models. [Brooks83]
- The models represent the problem at a different level of abstraction.
  - The top levels express the problem in the original problem domain.
  - The lower levels represent the problem in the computer’s domain.
- The models are informal, but detailed enough to support simulation.
Model based reasoning in programming

Models

- Specification
- Programming
- Computation
- Machine (AKA Cost Model)

Domain

- Problem Specific: polygons, rays, molecules, etc.
- OpenMP’s fork/join, Actors
- Threads – shared memory
- Processes – shared nothing
- Registers, ALUs, Caches, Interconnects, etc.
Programming process: getting started.

- The programmer starts by constructing a high level model from the specification.
- Successive refinement starts by using some combination of the following techniques:
  - The problem’s state is defined in terms of objects belonging to abstract data types with meaning in the original problem domain.
  - Key features of the solution are identified and emphasized. These features, sometimes referred to as "beacons" [Wiedenbeck89], emphasize key aspects of the solution.
The programming process

- Programmers use an informal, internal notation based on the problem, mathematics, programmer experience, etc.
  - Within a class of programming languages, the program generated is only weakly dependent on the language. [Robertson90] [Petre88]
- Programmers think about code in chunks or “plans”. [Rist86]
  - Low level plans code a specific operation: e.g. summing an array.
  - High level or global plans relate to the overall program structure.
Programming Process: Strategy + Opportunistic Refinement

- Common strategies are:
  - Backwards goal chaining: start at result and work backwards to generate sub goals. Continue recursively until plans emerge.
  - Forward chaining: Directly leap to plans when a problem is familiar. [Rist86]

- Opportunistic Refinement: [Petre90]
  - Progress is made at multiple levels of abstraction.
  - Effort is focused on the most productive level.
Programming Process: the role of testing

- Programmers test the emerging solution throughout the programming process.
- Testing consists of two parts:
  - Hypothesis generation: The programmer forms an idea of how a portion of the solution should behave.
  - Simulation: The programmer runs a mental simulation of the solution within the problem models at the appropriate level of abstraction. [Guindom90]
From psychology to software

- My central hypothesis:
  - A Design Pattern language provides the roadmap to apply results from the psychology of programming to software engineering:
    - Design patterns capture the essence of plans
    - The structure of patterns in a pattern language should mirror the types of models programmers use.
    - Connections between patterns must fit well with goal-chaining and opportunistic refinement.
Agenda

- Many core motivation slides
- Psychology of programming
- Design Patterns and Pattern Languages
- Berkeley patterns effort
Design Patterns and Pattern Languages

- A design pattern is:
  - A “solution to a problem in a context”.
  - A structured description of high quality solutions to recurring problems
  - A quest to encode expertise so all designers can capture that “quality without a name” that distinguishes truly excellent designs

- A pattern language is:
  - A structured catalog of design patterns that supports design activities as “webs of connected design patterns”.

-
Design Patterns:
A silly example

- Name: Money Pipeline

- Context: **You want to get rich and all you have to work with is a C.S. degree and programming skills.** How can you use software to get rich?

- Forces: **The solution must resolve the forces:**
  - It must give the buyer something they believe they need.
  - It can’t be too good, or people won’t need to buy upgrades.
  - Every good idea is worth stealing -- anticipate competition.

- Solution: **Construct a money pipeline**
  - Create SW with enough functionality to do something useful most of the time. This will draw buyers into your money pipeline.
  - Promise new features to thwart competitors.
  - Use bug-fixes and a slow trickle of new features to extract money as you move buyers along the pipeline.
Design Patterns

- Some example design patterns:
  - **Iterator**: visit each object in a collection without exposing the object’s internal representation.
  - **Task Queue**: the well known approach for embarrassingly parallel problems. Master-worker algorithms are a common example.
  - **Facade**: Provide a unified interface to a set of distinct interfaces in a subsystem.

- Each pattern is written down in a specific format and includes all the information required to apply it for a particular problem.
Our Pattern Language’s structure:
Four design spaces in parallel software development

Original Problem
Finding Concurrency
Algorithm Structure
Tasks, shared and local data

Units of execution + new shared data
for extracted dependencies

Algorithm Structure

Supporting stuct. & impl. mech.

Corresponding source code

Program SPMD_Emb_Par()
{
    TYPE *tmp, *func();
    global_array Data(TYPE);
    global_array Res(TYPE);
    int N = get_num_procs();
    int id = get_proc_id();
    if (id==0) setup_problem(N, DATA);
    for (int I= 0; I<N; I=I+Num){
        tmp = func(I);
        Res.accumulate( tmp);
    }
}

Corresponding source code

Program SPMD_Emb_Par()
{
    TYPE *tmp, *func();
    global_array Data(TYPE);
    global_array Res(TYPE);
    int Num = get_num_procs();
    int id = get_proc.id();
    if (id==0) setup_problem(N, Data);
    for (int I= ID; I<N; I=I+Num){
        tmp = func(I, Data);
        Res.accumulate( tmp);
    }
}
Our Pattern Language’s Structure

A software design can be viewed as a series of refinements.

We consider the process in terms of 4 design spaces which add progressively lower level elements to the design.

<table>
<thead>
<tr>
<th>Design Space</th>
<th>The Evolving Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>FindingConcurrency</td>
<td>Tasks, shared data, partial orders</td>
</tr>
<tr>
<td>AlgorithmStructure</td>
<td>Thread/process structures, schedules</td>
</tr>
<tr>
<td>SupportingStructures</td>
<td>Source Code organization, Shared data</td>
</tr>
<tr>
<td>ImplementationMechanisms</td>
<td>Messages, synchronization, spawn</td>
</tr>
</tbody>
</table>
Decomposition
(Finding Concurrency)

Start with a specification that solves the original problem -- finish with the problem decomposed into tasks, shared data, and a partial ordering.
Concurrency Strategy
(Algorithm Structure)
Implementation strategy
(Supporting Structures)

High level constructs impacting large scale organization of the source code.

Program Structure
- SPMD
- Master/Worker
- Loop Parallelism
- Fork/Join

Data Structures
- Shared Data
- Shared Queue
- Distributed Array
Parallel Programming building blocks

Low level constructs implementing specific constructs used in parallel computing. Examples in Java, OpenMP and MPI.

These are not properly design patterns, but they are included to make the pattern language self-contained.
Let’s use Design patterns to help people “think parallel”

Now available at a bookstore near you!

A pattern language for parallel algorithm design with examples in MPI, OpenMP and Java.

This is our hypothesis for how programmers think about parallel programming.
Name: The Task Parallelism Pattern

- **Context:**
  - How do you exploit concurrency expressed in terms of a set of distinct tasks?

- **Forces**
  - Size of task – small size to balance load vs. large size to reduce scheduling overhead.
  - Managing dependencies without destroying efficiency.

- **Solution**
  - Schedule tasks for execution with balanced load – use master worker, loop parallelism, or SPMD patterns.
  - Manage dependencies by:
    - removing them (replicating data),
    - transforming induction variables,
    - exposing reductions,
    - explicitly protecting (shared data pattern).
**Name: The SPMD Pattern**

- **Context:**
  - How do you structure a parallel program to make interactions between threads manageable yet easy to integrate with the core computation?

- **Forces**
  - Fewer programs are easier to manage, but complex algorithms often need very different instruction streams on each thread.
  - Balance the conflicting needs of scalability, maintainability, and portability.

- **Solution**
  - Use a single program for all the threads.
  - Keep it simple ... use the threads ID to select different pathways through the program.
  - Keep interactions between threads explicit and at a minimum.
Name: The Loop Parallelism Pattern

■ Context:
  □ How do you transform a serial program dominated by compute intensive loops into a parallel program without radically changing the semantics?

■ Forces
  □ An existing program implies expected output … and this must be preserved even as the programs execution changes due to parallelism.
  □ High performance requires restructuring data to optimize for cache … but this must be done carefully to avoid changing semantics.
  □ Assuring correctness suggests that the parallelization process should be incremental with each transformation subject to testing.

■ Solution
  □ Identify key compute intensive loops.
  □ Use directives/pragmas to parallelize these loops (i.e. at no point do you use the ID to manage loop parallelization by hand).
  □ Optimize incrementally testing at each step of the way – merging loops, modifying schedules, etc.
A simple pedagogical Example: The PI program

Numerical Integration

Mathematically, we know that:

\[
\int_{0}^{1} \frac{4.0}{1+x^2} \, dx = \pi
\]

We can approximate the integral as a sum of rectangles:

\[
\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi
\]

Where each rectangle has width \(\Delta x\) and height \(F(x_i)\) at the middle of interval \(i\).
OpenMP PI Program:

SPMD Pattern
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
    int i; double x, pi, sum[NUM_THREADS] = {0};
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        double x; int id, i;
        id = omp_get_thread_num();
        int nthreads = omp_get_num_threads();
        for (i=id; i< num_steps; i=i+nthreads) {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0; i<NUM_THREADS; i++) pi += sum[i] * step;
}
MPI Pi program

SPMD pattern

#include <mpi.h>
void main (int argc, char *argv[])
{
    int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
    step = 1.0/(double) num_steps ;
    MPI_Init(&argc, &argv) ;
    MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
    MPI_Comm_Size(MPI_COMM_WORLD, &numprocs);
    for (i=my_id;  i<num_steps; i=i+numprocs)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    sum *= step ;
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, 
               MPI_COMM_WORLD) ;
}
OpenMP PI Program:
Loop level parallelism pattern

```
#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2
void main ()
{
  int i;    double x, pi, sum =0.0;
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for private(x) reduction (+:sum)
  for (i=0;i< num_steps; i++){
    x = (i+0.5)*step;
    sum += 4.0/(1.0+x*x);
  }
  pi = sum[i] * step;
}
```

Loop Level Parallelism:
Parallelism expressed solely by (1) exposing concurrency, (2) managing dependencies, and (3) splitting up loops.
Agenda

- Many core motivation slides
- Psychology of programming
- Design Patterns and Pattern Languages
- Berkeley patterns effort
UCB’s Par Lab: Research Overview

Easy to write correct software that runs efficiently on manycore

Application Layer

Composition & Coordination Language (C&CL)

C&CL Compiler/Interpreter

Parallel Libraries
Parallel Frameworks

Legacy Code
Schedulers
Communication & Synch. Primitives

Efficiency Languages

Sketching
Autotuners

Legacy OS
OS Libraries & Services
Hypervisor

Intel Multicore/GPGPU
RAMP Manycore

Efficiency Language Compilers

OS
Arch.

Productivity Layer

Design Pattern Language

Personal Health
Image Retrieval
Hearing, Music
Speech
Parallel Browser

Easy to write correct software that runs efficiently on manycore
Influences on Our Pattern Language (OPL)
Choose your high level structure – what is the structure of my application? Guided expansion

Pipe-and-filter
Agent and Repository
Process Control
Event based, implicit invocation

Model-view controller
Bulk iterative
Map reduce
Layered systems
Arbitrary Static Task Graph

Identify the key computational patterns – what are my key computations? Guided instantiation

Graph Algorithms
Dynamic Programming
Dense Linear Algebra
Sparse Linear Algebra
Unstructured Grids
Structured Grids

Graphical models
Finite state machines
Backtrack Branch and Bound
N-Body methods
Circuits
Spectral Methods
Monte Carlo

Parallel algorithm strategy - what high level strategy do I use for my parallel algorithm? Guided re-organization

Task Parallelism
Recursive splitting

Data Parallelism
Pipeline

Geometric Decomposition
Discrete Event

Digital Circuits
Graph Partitioning

Implementation strategy – how do I express the algorithm in software? Guided programming

SPMD
Strict-data-par
Fork/Join
Actors

Master/worker
Loop Parallelism
BSP

Shared Queue
Shared Hash Table
Distributed Array
Shared Data

Data struc

Concurrent execution – how do we support the execution of the parallel algorithm? Guided HW mapping.

CSP
Task Graph
SIMD

Thread Pool
Speculation
data flow

Adv. program
counters

Msg pass
coll comm
mutual exclusion
Pt-2-pt sync
coll sync
Trans. Mem

Data struct
Coordination
OPL Version 2.0 http://parlab.eecs.berkeley.edu/wiki/patterns/patterns

Applications

Choose your high level structure – what is the structure of my application? Guided expansion
- Pipe-and-filter
- Model-view controller
- Agent and Repository
- Bulk iterative
- Process Control
- Map reduce
- Event based, implicit invocation
- Layered systems
- Arbitrary Static Task Graph

Identify the key computational patterns – what are my key computations? Guided instantiation
- Graph Algorithms
- Dynamic Programming
- Dense Linear Algebra
- Sparse Linear Algebra
- Unstructured Grids
- Structured Grids
- Backtrack Branch and Bound
- N-Body methods
- Spectral Methods
- Monte Carlo

Parallel algorithm strategy - what high level strategy do I use for my parallel algorithm? Guided re-organization
- Task Parallelism
- Data Parallelism
- Recursive splitting
- Pipeline

Algorithmic strategy
- Decomposition
- Digital Circuits
- Graph Partitioning

Implementation strategy – how do I express the algorithm in software? Guided programming
- SPMD
- Master/worker
- Loop Parallelism
- BSP
- Parallel procedures
- Speculation
- Data flow

Source Code
- Shared Queue
- Shared Hash Table
- Distributed Array
- Shared Data

Execution
- Message passing
- Coll. comm. mutual exclusion
- Pt-2-pt sync
- Trans. Mem

Concurrent execution – how do we support the execution of the parallel algorithm? Guided HW mapping.
- CSP
- Task Graph
- SIMD
- Thread Pool
- Speculation
- data flow

Adv. program counters

Coordination
- Adv. program counters
- Coll. comm. mutual exclusion
- Pt-2-pt sync
- Trans. Mem
But patterns are not an end in themselves: Design patterns inform creation of software frameworks

1. Domain Experts + Application frameworks → End-user, application programs

2. Domain literate programming gurus (1% of the population) + Parallel programming frameworks → Application frameworks

3. Parallel programming gurus (1-10% of programmers) → Parallel programming frameworks

The hope is for Domain Experts to create parallel code with little or no understanding of parallel programming.

Leave hardcore “bare metal” efficiency layer programming to the parallel programming experts
To make our example concrete ...

- **Reference Platform:**
  - a multi-socket system with 4 sockets each of which has a quad-core CPU.
    - Assume cores in a socket share a last level cache.
    - No shared address space between sockets (message passing)
  - Expect that over the lifetime of the software, the number of sockets will stay fixed, but the number of cores will increase to 64/socket.

- **Efficiency layer**
  - Distributed memory between sockets
    - MPI
  - Shared Address space on a socket
    - Pthreads
    - OpenMP
Spectral Methods computational pattern

- **Name:**
  - Spectral Methods

- **Problem:**
  - Some problems are easier to solve if you first transform the representation ... e.g. from the time domain to the frequency (spectral) domain. How do we solve these problems in parallel?

- **Context:**
  - PDE or image processing problem that is easier to solve following a transform.

- **Solution:**
  - Apply a discrete transform to the PDE turning differential operators into algebraic operators.
  - Solve the resulting system of algebraic or ordinary differential equations.
  - Inverse transform the solution to return to the original domain.
Spectral method: example

- Solve Poisson’s equation:

\[
\left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) f(x, y) = g(x, y)
\]

- Apply the 2D Discrete Fourier transforms:

\[
f = \sum a_{j,k} e^{2\pi i (jx + ky)} \quad g = \sum b_{j,k} e^{2\pi i (jx + ky)}
\]

- Differential equation becomes an algebraic equation:

\[
4\pi^2 \sum -a_{j,k} (j^2 + k^2) e^{2\pi i (jx + ky)} = \sum b_{j,k} e^{2\pi i (jx + ky)}
\]

\[
a_{j,k} = -\frac{b_{j,k}}{4\pi^2 (j^2 + k^2)}
\]

See lecture 6 on meshes for more
PLPP Patterns: Finding Concurrency

- **Task Decomposition:**
  - 1D FFTS making up the 2D FFT
  - Computing $a_{j,k}$ from $b_{j,k}$

- **Data Decomposition**
  - Column based decomposition to support FFT tasks
  - Any decomposition will do for $a_{j,k}$ computation

- **Grouping tasks**
  - Each phase of 2D FFT
  - $a_{j,k}$ update

- **Data dependency**
  - Transpose (all to all) between phases of 2D FFT

- **Design evaluation**
  - May want to parallelize 1D FFT if more concurrency is needed
  - Not needed for modest core count, but as number of cores increase, this could be very important.
The Algorithm Structure Design Space

Let’s focus on the data decomposition.

- Organize By Flow of Data
  - Regular?
  - Irregular?
  - Pipeline
  - Event Based Coordination

- Organize By Tasks
  - Linear?
  - Task Parallelism

- Organize By Data
  - Linear?
  - Recursive?
  - Geometric Decomposition
  - Recursive Data

**Name:**
- Geometric Decomposition

**Problem:**
- How can an algorithm be organized around a data structure that has been decomposed into concurrently updatable “chunks”?
Spectral Method: Algorithm Structure

- Column decomposition ... blocks of columns mapped to sockets.
The Supporting Structures
Design Space

Large column blocks per process managed by MPI … one process per socket

Exploit cores on a socket with OpenMP

Program Structure
- SPMD
- Master/Worker
- Loop Parallelism
- Fork/Join

Data Structures
- Shared Data
- Shared Queue
- Distributed Array
Spectral Method: Supporting Structure

```c
int Spec_solver(ID, Nue, N) {
    int col_ind[] = col_block(ID);
    barrier();
    transform(1, ID, col_ind);
    barrier();
    solve(col_ind);
    barrier();
    transform(-1, ID, col_ind);
    return;
}
```

```
ID = 0
ID = 1
ID = 2
ID = 3
```

solve(col_ind) <<<< and likewise for transform>>>>
```c
{ 
    #pragma omp parallel loop numthreads(num_cores)
    loop int I over col_ind {
        doit(I);
    }
    return
}
```
As core counts grow ...

- As core counts grow, you may want to parallelize 1D FFTS.
FFT algorithm

- Divide and conquer for \( N \log(N) \) algorithm

![FFT Algorithm Diagram]

- Probably would use “fork-join” with Pthreads ... but this would be hidden in library routines.
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
    int i, start;
    double x, sum = 0.0;
    start = *(int *) arg;
    step = 1.0/(double) num_steps;
    for (i=start; i<= num_steps; i=i+NUM_THREADS){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    EnterCriticalSection(&hUpdateMutex);
    global_sum += sum;
    LeaveCriticalSection(&hUpdateMutex);
}

void main ()
{
    double pi; int i;
    DWORD threadID;
    int threadArg[NUM_THREADS];
    for(i=0; i<NUM_THREADS; i++)   threadArg[i] = i+1;
    InitializeCriticalSection(&hUpdateMutex);
    for (i=0; i<NUM_THREADS; i++){
        thread_handles[i] = CreateThread(0, 0,
            (LPTHREAD_START_ROUTINE) Pi,
            &threadArg[i], 0, &threadID);
    }
    WaitForMultipleObjects(NUM_THREADS,
        thread_handles, TRUE,INFINITE);
    pi = global_sum * step;
    printf(" pi is %f \n",pi);
}
Summary

- Programming education needs to focus on the human angle ... teach people how to “think parallel”.
  - Major research universities should have joint computer science /psychology faculty.
- We are designing our research program around a design pattern language to help us keep this human connection explicit.
- Much work remains to be done:
  - Review the pattern language and help it evolve so it represents a more broad consensus view.
  - Work with us to test this pattern language on real programmers to confirm that “its right”.
  - Derive programming frameworks based on this pattern language ... to help programmers turn their thinking into software.
Join us in working on the pattern language

- ... a workshop for pattern writers and people eager to help them to build a consensus pattern language of parallel programming
  - http://www.upcrc.illinois.edu/workshops/paraplop09
References