RECAP

B649
Parallel Architectures and Programming
Parallel Programming

• A bag of tricks
  ★ less structured and categorized than “sequential” programming
  ★ less well understood in theoretical terms
  ★ still evolving

• Trends
  ★ at an inflection point?
  ★ legacy code and new code
  ★ productivity a major concern
PRODUCTIVITY IN HPC
High Productivity Computing Systems (HPCS)

Program Manager: Dr. Charles Holland

Mission:

Conduct a focused research and development program that creates a new generation of high productivity computing systems. These computing systems will comprise software tools, architectures, and hardware components. These systems will improve by orders of magnitude the effectiveness of humans and computers in solving the problems in the high performance computing domain.

Create economically viable high productivity computing systems for the national security and industrial user communities. These systems must have the following attributes:

- **Performance**: Improve the computational efficiency and reduce the execution time of critical national security applications.
- **Programmability**: Reduce cost and time of developing HPCS applications.
- **Portability**: Insulate HPCS application software from system specifics.
- **Robustness**: Improve reliability and reduce vulnerability to intentional attacks.
High Productivity Computing Systems

Providing a New Generation of Economically Viable
High Productivity Computing Systems

The DARPA High Productivity Computing Systems is focused on providing a new generation of economically viable high productivity computing systems for national security and for the industrial user community. HPCS program researchers have initiated a fundamental reassessment of how we define and measure performance, programmability, portability, robustness and ultimately, productivity in the HPC domain.

A Project Sponsored by

[Logos of various organizations]
Productivity and Performance

1 Introduction

The overall objective of programming support systems is to make it possible to produce software faster, without any degradation in software quality. However, it is essential that this goal must not be achieved at the cost of performance: programs written in a high-level language and intended to solve large problems on highly parallel machines must not be egregiously less efficient than the same applications written in a lower-level language. Because this has been a traditional stumbling block for high-level languages, metrics for productivity analysis must explore the trade-off between programming effort and performance.

To that end, we propose the use of two dimensionless ratios, relative power and relative efficiency, to measure

Productivity and Performance

Fig. 1 Power–efficiency graph.

Case Studies in HPC

<table>
<thead>
<tr>
<th></th>
<th>ASC Codes</th>
<th>MP Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td># of projects</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Environment</td>
<td>Academia (ASC-Alliance projects)</td>
<td>Mission Partners (DoD, DOE, NASA)</td>
</tr>
<tr>
<td>Classified</td>
<td>No</td>
<td>Some</td>
</tr>
<tr>
<td>Code size</td>
<td>200-600 KLOC</td>
<td>80-760 KLOC</td>
</tr>
<tr>
<td>Type</td>
<td>Coupled multi-physics applications</td>
<td>Single physics to coupled multi-physics and engineering</td>
</tr>
</tbody>
</table>

Table 1. Types of projects examined.

Case Studies in HPC

<table>
<thead>
<tr>
<th>Type</th>
<th>ASC Codes</th>
<th>MP Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Ongoing</td>
<td>Retrospective</td>
</tr>
<tr>
<td>Interviewees</td>
<td>Technical leads</td>
<td>Projects leads, project staff</td>
</tr>
</tbody>
</table>
| Overview | 1. Pre-interview questionnaire  
2. Telephone interview  
3. Generate summary document  
4. Send summary document for approval/comments  
5. Generate synthesis report across all projects  
6. Send synthesis report to all centers for approval/comments | 1. Identify project and sponsors  
2. Negotiate case study participation  
3. Pre-interview questionnaire  
4. On-site interview  
5. Initial list of findings  
6. Follow-up  
7. Write report |
| Focus    | • Product: attributes, machine target, history  
• Project organization: structure, staff, configuration management  
• Development activities: adding new features, testing, tuning, debugging, porting, effort distribution, bottlenecks, achieving performance  
• Programming models and productivity: choice of model, adoption of language, productivity measures | • Goals, requirements, deliverables  
• Project characteristics, structure, organization and risks  
• Code Characteristics  
• Staffing  
• Workflow Management  
• V&V, Testing  
• Success Measures  
• Lessons Learned |

Table 3. Methodology.

Observations about Software Development for High End Computing

# HPC Case Studies: Cases

<table>
<thead>
<tr>
<th>Application Domain</th>
<th>FALCON</th>
<th>HAWK</th>
<th>CONDOR</th>
<th>EAGLE</th>
<th>NENE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Duration</strong></td>
<td>~ 10 years</td>
<td>~ 6 years</td>
<td>~ 20 years</td>
<td>~ 3 years</td>
<td>~ 25 years</td>
</tr>
<tr>
<td><strong># of Releases</strong></td>
<td>9 (production)</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td><strong>Staffing</strong></td>
<td>15 FTEs</td>
<td>3 FTEs</td>
<td>3-5 FTEs</td>
<td>3 FTEs</td>
<td>~10 FTEs (100’s of contributors)</td>
</tr>
<tr>
<td><strong>Customers</strong></td>
<td>&lt; 50</td>
<td>10s</td>
<td>100s</td>
<td>None</td>
<td>~ 100,000</td>
</tr>
<tr>
<td><strong>Code Size</strong></td>
<td>~ 405,000 LOC</td>
<td>~ 134,000 LOC</td>
<td>~200,000 LOC</td>
<td>&lt; 100,000 LOC</td>
<td>750,000 LOC</td>
</tr>
<tr>
<td><strong>Primary Languages</strong></td>
<td>F77 (24%), C++ (67%), C (18%)</td>
<td>C++ (67%), C (18%)</td>
<td>F77 (85%)</td>
<td>C++, Matlab</td>
<td>F77 (95%)</td>
</tr>
<tr>
<td><strong>Other Languages</strong></td>
<td>F90, Python, Perl, ksh/csh/sh</td>
<td>Python, F90</td>
<td>F90, C, Slang</td>
<td>Java Libraries</td>
<td>C</td>
</tr>
<tr>
<td><strong>Target Hardware</strong></td>
<td>Parallel Supercomputer</td>
<td>Parallel Supercomputer</td>
<td>PCs to Parallel Supercomputer</td>
<td>Embedded Hardware</td>
<td>PCs to Parallel Supercomputer</td>
</tr>
<tr>
<td><strong>Status</strong></td>
<td>Production</td>
<td>Production Ready</td>
<td>Production</td>
<td>Demonstration Code</td>
<td>Production</td>
</tr>
</tbody>
</table>

HPC Case Studies: Lessons

HPC Case Studies: Lessons

• Verification and validation is extremely difficult
HPC Case Studies: Lessons

- Verification and validation is extremely difficult
- Primary language does not change over time

HPC Case Studies: Lessons

- Verification and validation is extremely difficult
- Primary language does not change over time
- The use of higher-level languages is low

HPC Case Studies: Lessons

• Verification and validation is extremely difficult
• Primary language does not change over time
• The use of higher-level languages is low
• Developers prefer UNIX command-line over IDE

HPC Case Studies: Lessons

• Verification and validation is extremely difficult
• Primary language does not change over time
• The use of higher-level languages is low
• Developers prefer UNIX command-line over IDE
• Externally developed software is a risk

HPC Case Studies: Lessons

• Verification and validation is extremely difficult
• Primary language does not change over time
• The use of higher-level languages is low
• Developers prefer UNIX command-line over IDE
• Externally developed software is a risk
• Performance competes with other important goals
  ★ correctness, performance, portability, maintainability

HPC Case Studies: Lessons

• Verification and validation is extremely difficult
• Primary language does not change over time
• The use of higher-level languages is low
• Developers prefer UNIX command-line over IDE
• Externally developed software is a risk
• Performance competes with other important goals
  ★ correctness, performance, portability, maintainability
• Agile methodologies are better accepted
HPC Case Studies: Lessons

- Verification and validation is extremely difficult
- Primary language does not change over time
- The use of higher-level languages is low
- Developers prefer UNIX command-line over IDE
- Externally developed software is a risk
- Performance competes with other important goals
  - correctness, performance, portability, maintainability
- Agile methodologies are better accepted
- Multidisciplinary teams are important to success

HPC Case Studies: Lessons

- Verification and validation is extremely difficult
- Primary language does not change over time
- The use of higher-level languages is low
- Developers prefer UNIX command-line over IDE
- Externally developed software is a risk
- Performance competes with other important goals
  - correctness, performance, portability, maintainability
- Agile methodologies are better accepted
- Multidisciplinary teams are important to success
- Success or failure depends on keeping customers satisfied
Measuring Productivity in HPC

Figure 1: Process of Refining and Evaluating HPC Programmer Productivity Hypotheses.

### Measuring Productivity in HPC

#### 4.2 Dependent Variables

Our studies measured the following as outcomes of the HPC development practices applied:

A primary measure of the quality of the HPC solution was the speedup achieved, that is, the relative execution time of a program running on a multi-processor system compared to a uniprocessor system. In this paper, all values reported for speedup were measured when the application was run on eight processors, as this was the largest number of processors that was feasible for use in our classroom environments.

A primary measure of cost was the amount of effort required to develop the final solution, for a given problem and given approach. The effort undertaken to develop a serial solution included the following activities: thinking/planning the solution, coding, and testing. The effort undertaken to develop a parallel solution included all of the above as well as tuning the parallel code (i.e. improving performance through optimizing the parallel instructions). HPC development for the studies presented in this paper was always done having a serial version available.

Another outcome variable studied was the code expansion factor of the solutions. In order to take full advantage of the parallel processors, HPC codes can be expected to include many more lines of code (LOC) than serial solutions. For example, message-passing approaches such as MPI require a significant amount of code to deal with communication across different nodes. The expansion factor is the ratio of LOC in a parallel solution to LOC in a serial solution of the same problem.

Finally, we look at the cost per LOC of solutions in each of the various approaches. This value is another measure (in person-hours) of the relative cost of producing code in each of the HPC approaches.

#### 4.3 Studies Described in this Paper

To validate our methodology, we selected a subset of the cells in the table for which we have already been able to gather sufficient data to draw conclusions (i.e., the gray-shaded cells in Table 1), where the majority of our data lies at this time.

Studies analyzed in this paper include:

- C0A1. This data was collected in Fall 2003, from a graduate-level course with 16 students. Subjects were asked to implement the "game of life" program in C on a cluster of PCs, first using a serial solution and then parallelizing the solution with MPI.

### Table 1: Matrix describing the problem space of HPC studies being run. Columns show the parallel programming model used. Rows show the assignment, grouped by communication pattern required. Each study is indicated with a label CxAy, identifying the participating class (C) and the assignment (A). Studies analyzed in this paper are grey-shaded.

<table>
<thead>
<tr>
<th>Nearest-Neighbor Type Problems</th>
<th>Serial</th>
<th>MPI</th>
<th>OpenMP</th>
<th>Co-Array Fortran</th>
<th>StarP</th>
<th>XMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game of Life</td>
<td>C3A3</td>
<td>C3A3</td>
<td>C0A1</td>
<td>C1A1</td>
<td>C3A3</td>
<td></td>
</tr>
<tr>
<td>Grid of Resistors</td>
<td>C2A2</td>
<td>C2A2</td>
<td>C2A2</td>
<td>C2A2</td>
<td></td>
<td>C2A2</td>
</tr>
<tr>
<td>Sharks &amp; Fishes</td>
<td>C6A2</td>
<td>C6A2</td>
<td>C6A2</td>
<td></td>
<td>C6A2</td>
<td></td>
</tr>
<tr>
<td>Laplace’s Eq.</td>
<td>C2A3</td>
<td></td>
<td>P2A3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWIM</td>
<td></td>
<td></td>
<td>C0A2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Broadcast Type Problems</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LU Decomposition</td>
<td></td>
<td></td>
<td>C4A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel Mat-vec</td>
<td></td>
<td></td>
<td></td>
<td>C3A4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quantum Dynamics</td>
<td></td>
<td></td>
<td></td>
<td>C7A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embarrassingly Parallel Type Problems</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C3A1</td>
<td>C3A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Miscellaneous Problem Types)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel Sorting</td>
<td>C3A2</td>
<td>C3A2</td>
<td>C3A2</td>
<td></td>
<td>C5A1</td>
<td></td>
</tr>
<tr>
<td>Array Compaction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Randomized Selection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C5A2</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Matrix describing the problem space of HPC studies being run. Columns show the parallel programming model used. Rows show the assignment, grouped by communication pattern required. Each study is indicated with a label CxAy, identifying the participating class (C) and the assignment (A). Studies analyzed in this paper are grey-shaded.

Measuring Productivity in HPC

Table 4: The mean and standard deviation of the total effort along with the number of subjects is shown for each programming model. All data sets are for C implementations of the Game of Life for data set C3A3.

<table>
<thead>
<tr>
<th>Programming Model</th>
<th>Effort (person-hrs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>mean 4.4, sd 4.3, n=15</td>
</tr>
<tr>
<td>MPI</td>
<td>mean 10.7, sd 8.9, n=16</td>
</tr>
<tr>
<td>OpenMP</td>
<td>mean 5.0, sd 3.5, n=16</td>
</tr>
</tbody>
</table>

Productivity with MPI

Abstract—There is widespread belief in the computer science community that MPI is a difficult and time-intensive approach to developing parallel software. Nevertheless, MPI remains the dominant programming model for HPC systems, and many projects have made effective use of it. It remains unknown how much impact the use of MPI truly has on the productivity of computational scientists.

In this paper, we examine a mature, ongoing HPC project, the Flash Center at the University of Chicago, to understand how MPI is used and to estimate the time that programmers spend on MPI-related issues during development. Our analysis is based on an examination of the source code, version control history, and regression testing history of the software. Based on our study, we estimate that about 20% of the development effort is related to MPI. This implies a maximum productivity improvement of 25% for switching to an alternate parallel programming model.

Keywords: MPI, debugging, effort, productivity, case study

## Productivity with MPI

### TABLE I. SIZE OF CODE BASE

<table>
<thead>
<tr>
<th>Language</th>
<th>With PARAMESH</th>
<th>Without PARAMESH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SLOC</td>
<td>% of total</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>377,149</td>
<td>87.2%</td>
</tr>
<tr>
<td>C</td>
<td>29,058</td>
<td>6.7%</td>
</tr>
<tr>
<td>Parameter (FLASH)</td>
<td>16,566</td>
<td>3.8%</td>
</tr>
<tr>
<td>Config (FLASH)</td>
<td>3,841</td>
<td>0.9%</td>
</tr>
<tr>
<td>Make</td>
<td>2,247</td>
<td>0.5%</td>
</tr>
<tr>
<td>Perl</td>
<td>1,753</td>
<td>0.4%</td>
</tr>
<tr>
<td>Python</td>
<td>1,576</td>
<td>0.4%</td>
</tr>
<tr>
<td>Shell</td>
<td>475</td>
<td>0.1%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>432,665</strong></td>
<td><strong>100%</strong></td>
</tr>
</tbody>
</table>

Productivity with MPI

<table>
<thead>
<tr>
<th></th>
<th>Amount of MPI Code</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of files</td>
<td>SLOC</td>
<td></td>
</tr>
<tr>
<td><strong>With PARAMESH</strong></td>
<td>MPI</td>
<td>471</td>
<td>213,397 SLOC</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>2625</td>
<td>406,207 SLOC</td>
</tr>
<tr>
<td></td>
<td>Percentage</td>
<td>17.9%</td>
<td>52.5%</td>
</tr>
<tr>
<td><strong>Without PARAMESH</strong></td>
<td>MPI</td>
<td>145</td>
<td>23,335 SLOC</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>1925</td>
<td>159,779 SLOC</td>
</tr>
<tr>
<td></td>
<td>Percentage</td>
<td>7.5%</td>
<td>14.6%</td>
</tr>
</tbody>
</table>

Productivity with MPI: MPI Calls vs SLOC

Productivity with MPI: MPI Calls vs Commits

Productivity with MPI: MPI Calls vs Commits

TABLE III. COMMITS RELATED TO BUG-FIXING

<table>
<thead>
<tr>
<th></th>
<th># of commits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-MPI</td>
<td>2366 (69.9%)</td>
</tr>
<tr>
<td>MPI</td>
<td>1021 (30.1%)</td>
</tr>
<tr>
<td>Total</td>
<td>3387 (100%)</td>
</tr>
</tbody>
</table>

TABLE IV. COMMITS ACROSS DEVELOPERS

<table>
<thead>
<tr>
<th>Developer</th>
<th># of source commits</th>
<th># of MPI commits</th>
<th>% MPI-related development</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>666 (16.2%)</td>
<td>112 (12.3%)</td>
<td>16.8%</td>
</tr>
<tr>
<td>B</td>
<td>630 (15.3%)</td>
<td>110 (12.1%)</td>
<td>17.5%</td>
</tr>
<tr>
<td>C</td>
<td>557 (13.6%)</td>
<td>162 (17.9%)</td>
<td>29.1%</td>
</tr>
<tr>
<td>D</td>
<td>484 (11.8%)</td>
<td>81 (8.9%)</td>
<td>16.7%</td>
</tr>
<tr>
<td>E</td>
<td>358 (8.7%)</td>
<td>36 (4.0%)</td>
<td>10.1%</td>
</tr>
<tr>
<td>F</td>
<td>286 (7.0%)</td>
<td>43 (4.7%)</td>
<td>15.0%</td>
</tr>
<tr>
<td>G</td>
<td>246 (6.0%)</td>
<td>41 (4.5%)</td>
<td>16.7%</td>
</tr>
<tr>
<td>H</td>
<td>241 (5.9%)</td>
<td>122 (13.5%)</td>
<td>50.6%</td>
</tr>
<tr>
<td>I</td>
<td>115 (2.8%)</td>
<td>81 (8.9%)</td>
<td>70.4%</td>
</tr>
<tr>
<td>J</td>
<td>102 (2.5%)</td>
<td>19 (2.1%)</td>
<td>18.6%</td>
</tr>
<tr>
<td>Total</td>
<td>4110</td>
<td>1220</td>
<td></td>
</tr>
</tbody>
</table>

Productivity with MPI: Code Layout

(with PARAMESH)

Productivity with MPI: Code Layout
(with PARAMESH)

Productivity with MPI: Conclusion

VII. Conclusion

In this paper, we have examined the role of MPI on a large-scale HPC code development project, and characterized the degree to which coding activities deal with MPI specifically. We addressed the issues inherent in any study of software engineering issues based on archaeological data by applying a rigorous case study methodology using triangulation: combining different measures to provide insight on the same topic. The general agreement among our measures (number of files, number of SLOC, number of commits, number of debugging activities) provides confidence that the results, showing that MPI-specific issues make up a small percentage of the overall coding activities, are indicative of real phenomena.

RECAP
Recap

• ILP
• Exploiting ILP
• Dynamic scheduling
• Thread-level Parallelism
• Memory Hierarchy
• Other topics through student presentations
• Parallel programming and productivity
ILP: Pipelining
Pipelining: Adding Latches
Pipelining: Adding Forwarding

- DADD R1, R2, R3
- DSUB R4, R1, R5
- AND R6, R1, R7
- OR R8, R1, R9
- XOR R10, R1, R11
Pipelining: Adding Branch Delay Slots

(a) From before
DADD R1, R2, R3
if R2 = 0 then
  Delay slot
becomes

if R2 = 0 then
  DADD R1, R2, R3

(b) From target
DSUB R4, R5, R6
if R1 = 0 then
  Delay slot
DADD R1, R2, R3
if R1 = 0 then
  DSUB R4, R5, R6

(c) From fall-through
DADD R1, R2, R3
if R1 = 0 then
  Delay slot
OR R7, R8, R9
DSUB R4, R5, R6

Extending the Basic Pipeline
Exploiting ILP Through Compiler Techniques

• Loop unrolling
• Making use of branch delayed slots
• Static branch prediction
• Loop fusion
• Unroll and jam
• ...

B649: Parallel Architectures and Programming, Spring 2009
Dynamic Branch Prediction

Address bits

bits to index BPB

Branch Prediction Buffer

0 1
1 0
...
Two-bit Branch Predictor
General n-bit Correlating Branch Predictors

Use Branch Target Buffers (BTBs) for caching branch targets.
Dynamic Scheduling: Tomasulo’s Approach
Tomasulo’s Approach: Observations

- RAW hazards handled by waiting for operands
- WAR and WAW hazards handled by register renaming
  - only WAR and WAW hazards between instructions currently in the pipeline are handled; is this a problem?
  - larger number of hidden names reduces name dependences
- CDB implements forwarding
Tomasulo’s Approach + Speculation

Fields in ROB
1. Instruction type
2. Destination
3. Value
4. Ready
Observations on Speculation

• Speculation enables precise exception handling
  ★ defer exception handling until instruction ready to commit

• Branches are critical to performance
  ★ prediction accuracy
  ★ latency of misprediction detection
  ★ misprediction recovery time

• Must avoid hazards through memory
  ★ WAR and WAW already taken care of (how?)
  ★ for RAW
    ★ don’t allow load to proceed if an active ROB entry has Destination field matching with A field of load
    ★ maintain program order for effective address computation (why?)
## Multiple Issue Processor Types

<table>
<thead>
<tr>
<th>Common name</th>
<th>Issue structure</th>
<th>Hazard detection</th>
<th>Scheduling</th>
<th>Distinguishing characteristic</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar (static)</td>
<td>dynamic</td>
<td>hardware</td>
<td>static</td>
<td>in-order execution</td>
<td>mostly in the embedded space: MIPS and ARM</td>
</tr>
<tr>
<td>Superscalar (dynamic)</td>
<td>dynamic</td>
<td>hardware</td>
<td>dynamic</td>
<td>some out-of-order execution, but no speculation</td>
<td>none at the present</td>
</tr>
<tr>
<td>Superscalar (speculative)</td>
<td>dynamic</td>
<td>hardware</td>
<td>dynamic with speculation</td>
<td>out-of-order execution with speculation</td>
<td>Pentium 4, MIPS R12K, IBM Power5</td>
</tr>
<tr>
<td>VLIW/LIW</td>
<td>static</td>
<td>primarily software</td>
<td>static</td>
<td>all hazards determined and indicated by compiler (often implicitly)</td>
<td>most examples are in the embedded space, such as the TI C6x</td>
</tr>
<tr>
<td>EPIC</td>
<td>primarily static</td>
<td>primarily software</td>
<td>mostly static</td>
<td>all hazards determined and indicated explicitly by the compiler</td>
<td>Itanium</td>
</tr>
</tbody>
</table>
Dyn. Scheduling+Multiple Issue+Speculation

• Design parameters
  ★ two-way issue (two instruction issues per cycle)
  ★ pipelined and separate integer and FP functional units
  ★ dynamic scheduling, but not out-of-order issue
  ★ speculative execution

• Task per issue: assign reservation station and update pipeline control tables (i.e., control signals)

• Two possible techniques
  ★ do the task in half a clock cycle
  ★ build wider logic to issue any pair of instructions together

• Modern processors use both (4 or more way superscalar)
Distributed-Memory Multiprocessors
Other Ways to Categorize Parallel Programming

- data vs task parallel
- client-server vs p-to-p / master-slave vs symm.
- course vs fine grained
- SPMD vs MPMD
- threads vs producer-consumer
- tightly vs loosely coupled
- shared mem vs msg passing
- recursive vs iterative
- data vs task parallel
Write Invalidate Cache Coherence Protocol for Write-Back Caches
Distributed Memory + Directories
Directory-Based Cache Coherence

Invalid

CPU read hit

Send read miss message

Invalidate

Cache read

CPU write hit

CPU write miss

Send invalidate message

Send read miss message

CPU read

CPU write

Send read miss message

Send invalidate message

CPU read

CPU write

Modified (read/write)

CPU write hit

CPU read hit

Data write back

Write miss

Fetch invalidate

Shared (read only)

Data value reply; Share = {P}

Read miss

Write miss

Data value reply; Share = {P}; data value reply

Shared (read only)

Read miss

Write miss

Fetch data value reply; Share = {P}

Data value reply

Data write back

Uncached

Exclusive (read/write)

Write miss

Fetch/Invalidate

Data value reply; Share = {P}

Read miss

Write miss

Data value reply; Share = {P}

Read miss

Write miss

Data value reply; Share = {P}
Other Topics

• x86 assembly programming
• VLIW / EPIC
• Vector processors
• Embedded systems
• Scientific applications
• GPUs and GPGPUs
• Interconnection networks
• Multi-stage interconnection networks
• Parallel graphs
WHAT’S NEXT?
Future

• Continued importance of parallel programming
  ★ challenge: how to program multiprocessors
  ★ role of programming languages and compilers

• Convergence or specialization?
  ★ “standardization” of general purpose architecture
  ★ migration of “special-purpose” CPUs for general use
Landscape of Parallel Computing Research: A View from Berkeley

The recent switch to parallel microprocessors is a milestone in the history of computing. A multidisciplinary group of researchers here in Berkeley has been meeting since Spring 2005 to discuss this change from the conventional wisdom. Our white paper summarizes our learnings from these discussions. This wiki is a meeting place for us as a research community to explore the future of parallel processing. The video interview with Dave Patterson, Krste Asanovic and Kurt Keutzer, or Dave Patterson’s presentation at a recent Distinguished Colloquium here at Berkeley are great introductions to the Berkeley View project. Here are the slides from a related talk by Dave Patterson.

We believe that much can be learned by examining the success of parallelism at the extremes of the computing spectrum, namely embedded computing and high performance computing. This led us to