Algorithms Design for the Parallelization of Nested Loops

Florina M. Ciorba

Computing Systems Laboratory
Department of Informatics and Computer Technology
School of Electrical and Computer Engineering
National Technical University of Athens, Greece

February 29, 2008

Outline

Outline

Outline

What is Parallel Processing?

Definition (Parallelization)

Analyzing large (or heavy) sequential programs for parallelism and restructuring them to run efficiently on parallel and/or distributed systems.

The need for parallel processing arises in various scientific fields:

- computational fluid dynamics – all sorts of fluids
- molecular dynamics and astrodynamics – e.g. nuclear fusion simulations
- environmental modeling – atmosphere, land use, acid rain
- integrated complex simulations – e.g. weather forecasting, climate changes
- health and biological modeling – empirical models, DNA and protein analysis
- structural dynamics – civil and automotive

The Process of Parallelization

Consists of three steps:

Step 1 **Task decomposition**: decomposing the applications into tasks.

Step 2 **Dependence analysis**: analyzing the dependencies between the decomposed tasks

Step 3 **Task scheduling**: scheduling these tasks onto the target parallel or distributed system.

Definition (Task)

Generally, a task can range from a simple statement to basic blocks, loops or sequences of these. In this thesis, a task refers to one iteration of a nested DO (or FOR) loop.

The Process of Parallelization

Step 1: **Task decomposition** is influenced by the following factors:

Concurrency - applications can be *embarrassingly parallel* (all tasks can be executed concurrently) or *embarrassingly serial* (no two tasks can be executed concurrently).

Granularity - expresses the computational size of tasks after the decomposition. There are three types of granularity: *fine*, *medium* and *coarse*.

Application type - consisting of distinct steps or one iterative block of (regular or not) computations

Target system - shared-memory architectures usually support a fine grain decomposition (cheap communication); distributed-memory architecture usually require a coarse gain decomposition (expensive communication).

The Process of Parallelization

Step 2: **Dependence analysis** - there are two types of dependencies:
Data dependencies created by data transfer between tasks

<herein> flow (true) dependencies - one tasks writes and another reads a variable (create a precedence order for the execution of tasks)

- anti-dependencies - one task reads and another writes a variable
- output dependencies - both tasks write on that variable

Control dependencies describe the control structure of a program.

The Process of Parallelization

Step 3: **Task Scheduling** consists of:

Temporal assignment - or time schedule, refers to assigning a start time to each task

Spatial assignment - or mapping, refers to allocating the tasks to processors, which will execute them according to the time schedule.

Outline

Motivation

Why parallelize nested loops?

Because they constitute the most computational intensive part of a heavy application $\Rightarrow$ the most performance gain.

How does task scheduling relates to loop scheduling?

Loop scheduling is a particular case of task scheduling, in which each loop iteration is considered to be a task. Hence, DOACROSS loop scheduling refers to the problem of scheduling dependent tasks.

How easy/difficult is it to schedule tasks?

Task scheduling is an NP-complete problem. Many heuristics have been proposed. Good scheduling heuristic (static and/or dynamic) should be based on realistic assumptions (communication cost, number of processors, heterogeneity).

Outline

Problem Definition

Definition (Task Scheduling)

Given a set of tasks of a parallel computation, determine how the tasks can be assigned (both in space and time) to processing resources (scheduled on them) to satisfy certain optimality criteria.

Challenges

- minimizing execution time
- minimizing inter-processor communication
- load balancing tasks
- handling and/or recovering from failures
- meeting deadlines
- a combination of these

Addressing the Problem of Task Scheduling

Facts:

1. Task scheduling onto a set of homogeneous resources, considering interprocessor communication, and aiming to minimize the total execution time is NP-complete.
2. Things are worse ☹ for heterogeneous systems.

Problems to address:

Heterogeneity of processors, of communication links, irregularity of interconnection networks, non-dedicated platforms

Solutions:

- Optimal - there are no polynomial time optimal solutions ☹
- Heuristic methods - numerous (static/dynamic) scheduling heuristics have been proposed

Outline

Selected Bibliography – Static DOACROSS Loops

Fine grained heuristics aiming for *optimal time scheduling*

A. Consider unit execution time for each iteration and zero communication at each step (UET model):

1. Hyperplane method [Lamport, 1974]
2. [Moldovan and Fortes, 1986] applied the hyperplane method to find a linear optimal execution schedule using diophantine equations and [Shang and Fortes, 1991] using linear programming in subspaces
3. [Darte et al, 1991] proved that the hyperplane method is nearly optimal
4. [Koziris et al, 1996] yields the optimal time using the minimum number of processors

Selected Bibliography – Static DOACROSS Loops

Fine grained heuristics aiming for *optimal time scheduling*

B. Consider unit execution time and unit communication time at each step (UET-UCT model):

1. Polynomial time algorithm for scheduling in- and out-forrests [Varvarigou et al, 1996]
2. Polynomial time solutions for special cases of UET-UCT DAGs were proposed by [Jung et al, 1989], [Chretienne, 1992] and [Andronikos et al, 1997]
3. Find the optimal hypersurface for UET/UET-UCT loops [Papakonstantinou et al, 2001]

Selected Bibliography – Static DOACROSS Loops

Fine grained heuristics aiming for to *minimize the communication costs*

C. Consider arbitrary execution and communication costs

1. Minimize the communication cost by grouping neighboring iterations into chains [King et al, 1991], [Sheu and Chen, 1995], [Tsanakas et al, 2000], [Drositis et al, 2000]
2. [Papadimitriou and Yannakakis, 1988] proposed a heuristic that guaranteed the worst performance twice the optimum makespan, for a DAG with arbitrary computation and communication times

Selected Bibliography – Static DOACROSS Loops

Coarse grained heuristics with *arbitrary* computation and communication times

1. Most common loop transformation is tiling (or loop blocking) proposed by [Irigoin and Triolet, 1988]
2. [Wolf and Lam, 1991] minimize the communication between tiles
3. [Ramanujam and Sadayappan, 1992], [Boulet et al, 1994] and [Xue, 1997] study the problem of finding the optimal tile shape/size
4. [Goumas et al, 2006] study the problem of efficient code generation for non-rectangular tiling

Outline

What is Missing?

Methods to address:

- Maximization of resource utilization
- Minimization of the inter-processor communication cost
- Dynamic scheduling and load balancing
- Fault tolerance and reliability
- Scalability

for DOACROSS loops

Outline

How does this dissertation contribute to the field?

- Presents novel static methods for fine and coarse grained parallelization
- A dynamic scheduling algorithm for a special class of DOACROSS loops on shared and distributed memory systems [Ciorba et al, 2003], [Andronikos et al, 2004].

hereafter A dynamic multi-phase scheduling that extends several dynamic scheduling algorithms initially devised for DOALL loops and applied them to DOACROSS loops in heterogeneous systems [Ciorba et al, 2006], [Papakonstantinou et al, 2006].

hereafter Two general mechanisms for enhancing the performance of self-scheduling algorithms for DOACROSS loops on heterogeneous systems through synchronization and weighting [Ciorba et al, 2008].

- A theoretical model that determines the optimal synchronization frequency for the pipelined execution of DOACROSS loops on heterogeneous systems [Ciorba et al, 2007a].

Outline

Outline

DOALL and DOACROSS Nested Loops

There are two types of nested loops:

DOALL	- the iterations are independent and can be executed in any order
DOACROSS	- there exist dependencies between the iterations which impose a certain execution order

Recall that nested loops constitute most computationally intensive part of a program

Recall that each iteration of a nested loop is considered to be a task

Outline

Graphical Representations of DOACROSS Loops

Applications with *DOACROSS loops* are represented by:

Directed Acyclic Graphs (DAGs) - the numbered vertices represent tasks and the edges (or arcs) represent the dependencies among the tasks

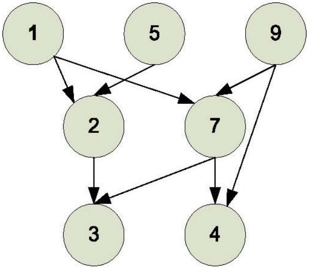


Figure: DAG representation of tasks and dependencies

Graphical Representations of DOACROSS Loops

or in our case by:

Cartesian Spaces - the points have coordinates and represent tasks and the directed vectors represent the dependencies among the tasks (e.g. precedence)

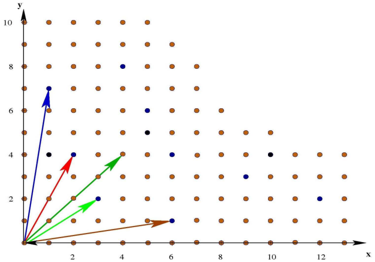


Figure: Cartesian representation of tasks and dependencies

Outline

Algorithmic Model - DOACROSS Loops

```
for (i1 = l1; i1 <= u1; i1++)
    for (i2 = l2; i2 <= u2; i2++)
        ...
            for (in = ln; in <= un; in++)
                S1(I);
                ...
                Sk(I);
            endfor
        ...
    endfor
endfor
```

- $J = \{\mathbf{I} \in \mathbb{N}^n | l_r \leq i_r \leq u_r, 1 \leq r \leq n\}$ - the Cartesian n-dimensional index space of a loop of depth n
- $|J| = \prod_{i=1}^{n}(u_i - l_i + 1)$ - the cardinality of J
- $S_i(\mathbf{I})$ - general program statements of the loop body
- $DS = \{\tilde{\mathbf{d}}_1, \ldots, \tilde{\mathbf{d}}_p\}$, $p \geq n$ - the set of dependence vectors
- By definition $\tilde{\mathbf{d}}_j > \mathbf{0}$, where $\mathbf{0} = (0, \ldots, 0)$ and $>$ is the *lexicographic* ordering
- $\mathbf{L} = (l_1, \ldots, l_n)$ - the initial point of J
- $\mathbf{U} = (u_1, \ldots, u_n)$ - the terminal point of J

Outline

What is Dynamic Scheduling?

Definition (Dynamic Scheduling)

In dynamic scheduling, only a few assumptions about the parallel program or the parallel system can be made before execution, and thus, scheduling decisions have to be made on-the-fly.

What is the goal?

To minimize the program completion time and minimize the scheduling overhead which constitutes a significant portion of the cost paid for running the dynamic scheduler.

Why do we need dynamic scheduling?

Dynamic scheduling is necessary when static scheduling may result in a highly imbalanced distribution of work among processors or when the inter-tasks dependencies are dynamic (e.g. due to changing system's behavior or changing application's behavior), thus precluding a static scheduling approach.

Outline

Dynamic Multi-Phase Scheduling

Motivation:

- Existing dynamic algorithms can not cope with dependencies, because they lack inter-slave communication
- If dynamic algorithms are applied to DOACROSS loops, in their original form, they yield a serial/invalid execution
- Static algorithms are not always efficient on heterogeneous systems

Contributions:

- Extended master-slave model with inter-slave communication
- A scheme that brings dynamic DOALL (coarse grained) loops scheduling schemes into the field of scheduling DOACROSS loops

Partitioning the Index Space with Self-Scheduling Algorithms

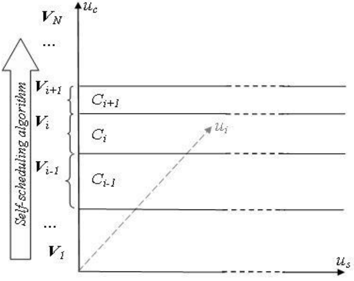


Figure: Index space partitioned with self-scheduling algorithms

- u_c - scheduling dimension
- u_s - synchronization dimension
- PE - processing element
- $P_1, \ldots, P_m$ - slave processors; P_0 - master processor
- N - the number of scheduling steps (the total number of chunks)
- C_i - chunk size at the i-th scheduling step
- V_i - the projection of C_i along scheduling dimension u_c
- $C_i = V_i \times \frac{\prod_{j=1}^{n} u_j}{u_c}$
- VP_k - virtual computing power of slave P_k (delivered speed)
- q_k - number of processes in the run-queue of slave P_k
- $A_k = \lfloor \frac{VP_k}{q_k} \rfloor$ - available computing power of slave P_k

Existing Self-Scheduling Algorithms for DOALL loops

- PSS [Polychronopoulos and Kuck, 1987] - Pure Self-Scheduling, $C_i = 1$
  ☺ good load balance ☹ excessive scheduling overhead
- CSS [Kruskal and Weiss, 1985] - Chunk Self-Scheduling, $C_i =$ constant > 1
  ☹ large chunks ⇒ load imbalance or ☹ small chunks ⇒ excessive scheduling overhead ∴ tradeoff required
- TSS [Tzen and Ni, 1993] - Trapezoid Self-Scheduling, $C_i = C_{i-1} - D$, where D decrement, the first chunk is $F = \frac{|J|}{2m}$ and the last chunk is $L = 1$
  ☺ reduces the need for synchronization and maintains reasonable load balance
- DTSS [Chronopoulos et al, 2001] - Distributed TSS, $C_i = A_k \times (F - D \times (S_{k-1} + (A_k - 1)/2))$, where: $S_{k-1} = A_1 + \ldots + A_{k-1}$, the first chunk is $F = \frac{|J|}{2A}$ and the last chunk is $L = 1$
  ☺ reduces synchronization, better load balance

Existing Self-Scheduling Algorithms for DOALL loops

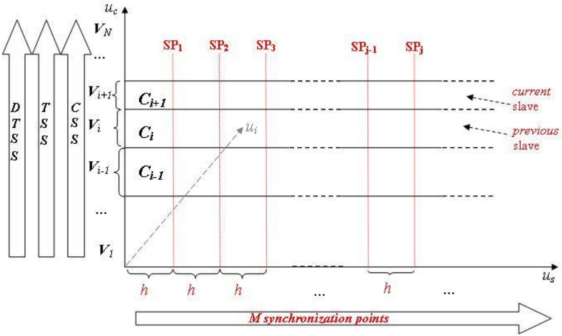
Table: Sample chunk sizes given for $|J| = 5000 \times 10000$ and $m = 10$

Algorithm	Chunk sizes
PSS	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ... 1
CSS	300 300 300 300 300 300 300 300 300 300 300 300 300 300 300 300 200
TSS (D=7)	277 270 263 256 249 242 235 228 221 214 207 200 193 186 179 172 165 158 151 144 137 130 123 116 109 102 73
DTSS (dedicated)	392 253 368 237 344 221 108 211 103 300 192 276 176 176 252 160 77 149 72 207 130 183 114 159 98 46 87 41 44
DTSS (non-dedicated)	263 383 369 355 229 112 219 107 209 203 293 279 265 169 33 96 46 89 86 83 80 77 74 24 69 66 31 59 56 53 50 47 44 20 39 20 33 30 27 24 21 20 20 20 20 20 20 20 20 8

- $|J| = 5000 \times 10000$ points
- $m = 10$ slaves
- ☹ CSS and TSS give the same chunk sizes both in dedicated and non-dedicated systems, respectively
- ☺ DTSS adjusts the chunk sizes to match the different A_k of slaves ($m/2$ were loaded)

Multi-Phase Scheduling

Self-Scheduling with Synchronization Points

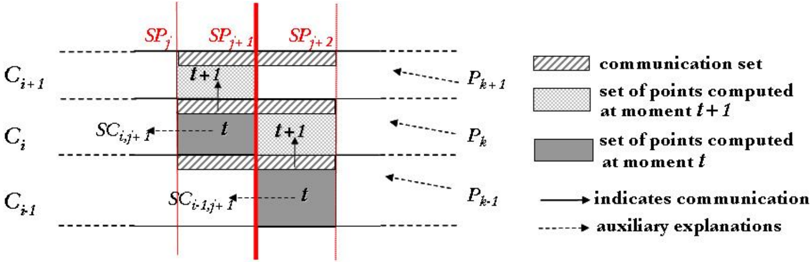


- Chunks are formed along the scheduling dimension, u_c
- *SP*s are inserted along the synchronization dimension, u_s

Phase 1 Apply self-scheduling algorithms to the scheduling dimension

Phase 2 Insert synchronization points along the synchronization dimension

The Inter-slave Communication Scheme



- C_{i-1} is assigned to P_{k-1}, C_i assigned to P_k and C_{i+1} to P_{k+1}
- When P_k reaches SP_{j+1}, it sends to P_{k+1} only the data P_{k+1} requires (i.e., those iterations imposed by the existing dependence vectors)
- Next, P_k receives from P_{k-1} the data required for the current computation

Obs. Slaves do not reach a *SP* at the same time, which leads to a **pipelined execution**

Dynamic Multi-Phase Scheduling DMPS($\mathscr{A}$)

INPUT

Master(a) An n-dimensional DOACROSS loop
(b) The choice of the scheduling algorithm [CSS, TSS or DTSS]
(c) If CSS is chosen, then constant chunk size C_i
(d) The synchronization interval h
(e) The number of slaves m; in case of DTSS, the virtual power V_k of every slave

Master:

Init (M.a) Register slaves; in case of DTSS, slaves report their A_k
(M.b) Calculate F, L, N, D for TSS and DTSS; for CSS use given C_i

While there are unassigned chunks do:

(M.1) If a request arrives, put it in the queue
(M.2) Pick a request from the queue, and compute the next chunk size using CSS, TSS or DTSS
(M.3) Update the *current* and *previous* slave ids
(M.4) Send the id of the current slave to the previous one

Dynamic Multi-Phase Scheduling DMPS($\mathscr{A}$)

Slave P_k:

Init (S.a) Register with the master; in case of DTSS, report A_k

(S.b) Compute M according to the given h

(S.1) Send request to the master

(S.2) Wait for reply; if received chunk from master, go to step S.3, else go to **OUTPUT**

(S.3) While the next *SP* is not reached, compute chunk i

(S.4) If id of the *send-to* slave is known, go to step S.5, else go to step S.6.

(S.5) Send computed data to *send-to* slave

(S.6) Receive data from the *receive-from* slave and go to step S.3

OUTPUT

Master If there are no more chunks to be assigned, **terminate**

Slave P_k If no more tasks come from master, **terminate**

Dynamic Multi-Phase Scheduling DMPS($\mathscr{A}$)

Advantages:

- Can take as input *any* self-scheduling algorithm, without any modifications
- **Phase 2** is independent of **Phase 1**
- **Phase 1** deals with the heterogeneity & load variation in the system
- **Phase 2** deals with minimizing the inter-slave communication cost
- Suitable for any type of heterogeneous systems

Results
Experimental Setup

- The algorithms are implemented in C and C++
- MPI is used for master-slave and inter-slave communication
- The heterogeneous system consists of 10 machines:
  - 4 zealots: Intel Pentiums III, 1266 MHz with 1GB RAM, assumed to have $VP_k = 1.5$ (one of them is the master)
  - 6 kids: Intel Pentiums III, 500 MHz with 512MB RAM, assumed to have $VP_k = 0.5$
- Interconnection network is Fast Ethernet, at 100Mbit/sec
- Dedicated system: all machines are dedicated to running the program and no other loads are interposed during the execution
- Non-dedicated system: at the beginning of program's execution, a resource expensive process is started on some of the slaves, halving their A_k

Results
Experimental Setup

- Machinefile: zealot1(master),zealot2, kid1,zealot3, kid2,zealot4, kid3, kid4, kid5, kid6
- Three series of experiments for both dedicated & non-dedicated systems, for m = 3,4,5,6,7,8,9 slaves:
  1) DMPS(CSS)
  2) DMPS(TSS)
  3) DMPS(DTSS)
- Real-life application: Heat diffusion equation (similar results for Floyd-Steinberg - in thesis)
- Speedup is computed with: $S_p = \frac{min\{T_{P_1}, T_{P_2}, \ldots, T_{P_m}\}}{T_{PAR}}$ where

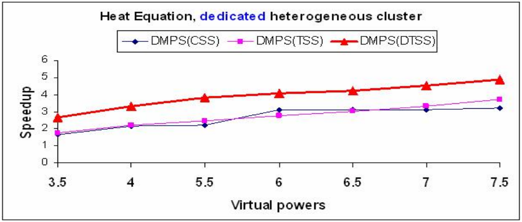
T_{P_k} - serial execution time on slave P_k, $1 \leq k \leq m$, and
T_{PAR} - parallel execution time (on m slaves)

Obs. In the plotting of S_p, VP is used instead of m on the x-axis

Results

Heat Diffusion Equation - Dedicated Heterogeneous System

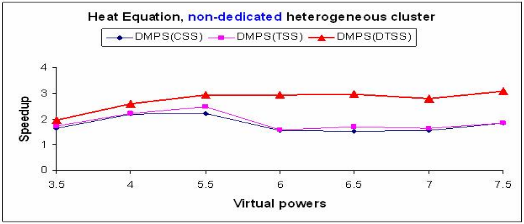
h	Dedicated	Parallel time (sec)						
	m	3	4	5	6	7	8	9
100	DMPS(CS)	2.32	1.75	1.73	1.23	1.21	1.21	1.18
	DMPS(TSS)	2.20	1.73	1.56	1.38	1.25	1.14	1.02
	DMPS(DTSS)	1.42	1.14	1.00	0.95	0.91	0.85	0.78
150	DMPS(CS)	2.31	1.74	1.71	1.21	1.22	1.21	1.18
	DMPS(TSS)	2.18	1.72	1.54	1.38	1.25	1.14	1.02
	DMPS(DTSS)	1.42	1.13	0.99	0.93	0.90	0.84	0.78
200	DMPS(CS)	2.30	1.74	1.73	1.22	1.23	1.22	1.19
	DMPS(TSS)	2.21	1.74	1.55	1.38	1.25	1.14	1.02
	DMPS(DTSS)	1.42	1.13	0.99	0.94	0.90	0.83	0.78



Results

Heat Diffusion Equation - Non-dedicated Heterogeneous System

h	Non-dedicated	Parallel time (sec)						
	m	3	4	5	6	7	8	9
100	DMPS(CS)	2.33	1.76	1.73	2.46	2.45	2.38	2.06
	DMPS(TSS)	2.20	1.74	1.56	2.52	2.56	2.18	2.10
	DMPS(DTSS)	1.95	1.45	1.30	1.31	1.33	1.38	1.25
150	DMPS(CS)	2.33	1.74	1.72	2.46	2.49	2.43	2.05
	DMPS(TSS)	2.19	1.72	1.54	2.42	2.23	2.31	2.06
	DMPS(DTSS)	1.94	1.47	1.30	1.30	1.28	1.36	1.23
200	DMPS(CS)	2.30	1.74	1.73	2.39	2.36	2.38	2.10
	DMPS(TSS)	2.22	1.75	1.56	1.79	2.32	2.10	2.02
	DMPS(DTSS)	1.96	1.44	1.29	1.29	1.27	1.32	1.21



Interpretation of Results

- Dedicated system:
  - All algorithms perform better on a dedicated system, than on a non-dedicated one ☺ expected!
  - DMPS(TSS) slightly outperforms DMPS(CSS) for parallel loops, because it provides better load balancing by reducing the chunk size
  - DMPS(DTSS) outperforms both other algorithms because it explicitly accounts for system's heterogeneity ☺ expected!
- Non-dedicated system:
  - DMPS(DTSS) stands out even more, since the other algorithms cannot handle extra load variations ☺ expected!
  - The speedup for DMPS(DTSS) increases in all cases ☺ expected!
- *h* must be chosen so as to maintain the comm/comp ratio $<< 1$, for every test case
- *h* is determined empirically or selected by the user
- However, small variations of the value of *h*, do not significantly affect the overall performance

Conclusions

- DOACROSS loops can now be dynamically scheduled on heterogeneous dedicated & non-dedicated systems
- Dynamic self-scheduling algorithms are also efficient for DOACROSS loops

What's missing?

- A generic add-on to other self-scheduling algorithms, such that they can all handle DOACROSS loops and account for system's heterogeneity without any modifications
- A model for predicting the optimal synchronization interval *h* and minimizing the communication

Outline

Enhancing Self-Scheduling Algorithms via Synchronization and Weighting

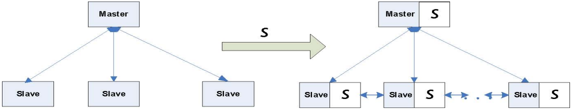
Motivation:

Existing self-scheduling algorithms need something to enable them to handle DOACROSS loops and something else to enable them to be efficient on heterogeneous systems

Contributions:

- A synchronization mechanism (the 'something')
- A weighting mechanism (the 'something else')

The Synchronization Mechanism $\mathscr{S}$



(a) Classic master-slave model

(b) Master-slave model with the stand alone synchronization component and inter-slaves communication links

- Enables self-scheduling algorithms to handle DOACROSS loops
- Provides:
  - The synchronization interval h along u_s: $h = \frac{U_s}{M}$
  - A framework for inter-slave communication (presented earlier)

Observations:

1. $\mathscr{S}$ is completely independent of the self-scheduling algorithm and does not enhance the load balancing capability of the algorithm
2. The synchronization overhead is compensated by the increase of parallelism $\Rightarrow$ overall performance improvement

The Synchronization Mechanism $\mathscr{S}$

Master

While there are unassigned chunks
{
1. Receive request from P_k
2. Calculate C_i according to $\mathscr{A}$
3. Serve request
}

$\mathscr{S}$-Master

While there are unassigned chunks
{
1. Receive request from P_k
2. Calculate C_i according to $\mathscr{A}$
3. Make P_k - current slave
4. Make P_{k-1} - previous slave
5. Send P_{k-1} the rank of P_k
6. Send P_k the rank of P_{k-1}
7. and serve request
}

Slave P_k

1. Make new request to Master
2. If request served
{
Compute chunk
}
3. Go to step 1

$\mathscr{S}$-Slave P_k

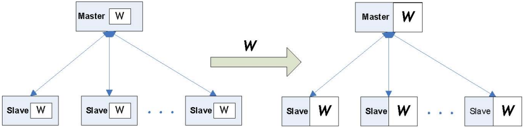
1. Make new request to Master
2. If request served
{
Receive partial results from P_{k-1}
Compute chunk
Send partial results to P_{k+1}
}
3. Go to step 1

$\mathscr{S}$ adds 3 components to the original algorithm $\mathscr{A}$:

1. transaction accounting (master)
2. receive part (slave)
3. transmit part (slave)

h is determined empirically or selected by the user and must be a trade-off between synchronization overhead and parallelism

The Weighting Mechanism $\mathscr{W}$



(a) Master-slave model with embedded work-weighting

(b) Master-slave model with the stand alone weighting component

- Enables self-scheduling algorithms to handle load variations and system heterogeneity
- Adjusts the amount of work (chunk size) given by the original algorithm $\mathscr{A}$ according to the current load of a processor and its nominal computational power

Observations:

1. $\mathscr{W}$ is completely independent of the self-scheduling algorithm and can be used alone on DOALL loops
2. The weighting overhead is insignificant (a $\star$ and a $/$ operation)
3. On a dedicated homogeneous system, $\mathscr{W}$ does not improve the performance and could be omitted

The Weighting Mechanism $\mathscr{W}$

Master

```
While there are unassigned chunks
 {
   1. Receive request from Pk
   2. Calculate Chunk according to A
   3. Serve Request
 }
```

$\mathscr{W}$-Master

```
While there are unassigned chunks
 {
   1. Receive request from Pk
   2. Calculate Ci according to A
   3. Apply W to compute Ĉi
   4. Serve request
 }
```

Slave P_k

```
1. Make new request to Master
2. If request served
   {
       Compute chunk
   }
3. Go to step 1
```

$\mathscr{W}$-Slave P_k

```
1. Make new request to Master
2. Report current load Qk
3. If request served
   {
       Compute chunk
   }
4. Go to step 1
```

$\mathscr{W}$ adds 2 components to the original algorithm $\mathscr{A}$:

1. chunk weighting (master)
2. run-queue monitoring (slave)

$\mathscr{W}$ calculates the chunk $\widehat{C_i}$ assigned to P_k as follows: $\widehat{C_i} = C_i \times \frac{VP_k}{q_k}$, where C_i is the chunk size given by the original self-scheduling algorithm $\mathscr{A}$.

Two More Existing Self-Scheduling Algorithms

For DOALL Loops

FSS [Hummel et al, 1992] – Factoring Self-Scheduling, assigns batches of equal chunks. $C_i = \lceil \frac{R_i}{\alpha * m} \rceil$ and $R_{i+1} = R_i - (m \times C_i)$, where the parameter α is computed (by a probability distribution) or is sub-optimally chosen $\alpha = 2$.
☺ few chunks adaptations ☹ difficult to determine the optimal parameter

GSS [Polychronopoulos and Kuck, 1987] – Guided Self-Scheduling, $C_i = R_i / m$, where R_i is the number of remaining iterations
☺ large chunks first ⇒ reduced communication ☺ small chunks last ⇒ balance the load among processors

Evaluation of $\mathscr{W}$

Table: Chunk sizes given by the original and weighted algorithms for the Mandelbrot set (irr. DOALL), index space size $|J| = 10000 \times 10000$ points and $m = 4$, $VP_1 = VP_3 = 1$, $VP_2 = VP_4 = 0.8$ and P_2 and P_4 were loaded

$\mathscr{A}$	Chunk sizes with $\mathscr{A}$ with respect to the processors' request order	Chunk sizes with $\mathscr{W}$-$\mathscr{A}$ with respect to the processors' request order	Par. time for $\mathscr{A}$	Par. time for $\mathscr{W}$-$\mathscr{A}$
CSS	$1250(P_1)$ $1250(P_2)$ $1250(P_3)$ $1250(P_4)$ $1250(P_3)$ $1250(P_1)$ $1250(P_3)$ $1250(P_1)$	$1250(P_1)$ $1250(P_3)$ $500(P_4)$ $500(P_2)$ $1250(P_3)$ $500(P_2)$ $500(P_4)$ $1250(P_1)$ $1250(P_3)$ $500(P_4)$ $1250(P_1)$	$120.775s$	$66.077s$
FSS	$1250(P_1)$ $1250(P_3)$ $1250(P_2)$ $1250(P_4)$ $625(P_3)$ $625(P_3)$ $625(P_1)$ $625(P_3)$ $390(P_1)$ $390(P_1)$ $390(P_3)$ $390(P_1)$ $244(P_3)$ $244(P_4)$ $244(P_1)$ $208(P_3)$	$1250(P_1)$ $1250(P_3)$ $500(P_2)$ $500(P_4)$ $812(P_3)$ $324(P_2)$ $324(P_4)$ $324(P_1)$ $324(P_3)$ $812(P_3)$ $630(P_1)$ $630(P_1)$ $630(P_4)$ $252(P_3)$ $176(P_1)$ $441(P_4)$ $441(P_2)$ $176(P_3)$ $123(P_1)$ $308(P_2)$ $308(P_4)$ $113(P_1)$	$120.849s$	$56.461s$

Slower slaves request work only once (they need more time to compute a chunk) and $\mathscr{W}$ compensates for this

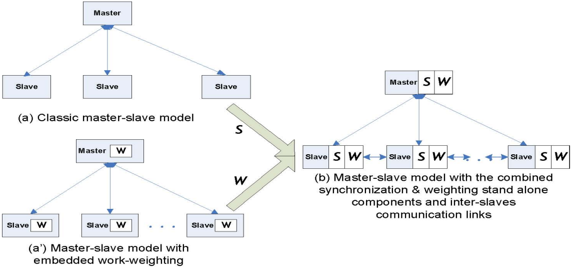
Evaluation of $\mathscr{W}$

Table: Chunk sizes given by the original and weighted algorithms for the Mandelbrot set (irr. DOALL), index space size $|J| = 10000 \times 10000$ points and $m = 4$, $VP_1 = VP_3 = 1$, $VP_2 = VP_4 = 0.8$ and P_2 and P_4 were loaded

$\mathscr{A}$	Chunk sizes with $\mathscr{A}$ with respect to the processors' request order	Chunk sizes with $\mathscr{W}$-$\mathscr{A}$ with respect to the processors' request order	Par. time for $\mathscr{A}$	Par. time for $\mathscr{W}$-$\mathscr{A}$
GSS	2500(P_1) 1875(P_2) 1406(P_3) 1054(P_4) 791(P_3) 593(P_3) 445(P_3) 334(P_1) 250(P_3) 188(P_1) 141(P_3) 105(P_1) 80(P_3) 80(P_1) 80(P_3) 78(P_1)	2500(P_1) 1875(P_3) 562(P_2) 506(P_4) 455(P_4) 410(P_2) 923(P_3) 692(P_3) 519(P_1) 155(P_4) 140(P_2) 315(P_3) 94(P_4) 213(P_1) 160(P_3) 120(P_1) 90(P_3) 80(P_2) 80(P_1) 80(P_3) 31(P_1)	145.943s	58.391s
TSS	1250(P_1) 1172(P_3) 1094(P_2) 1016(P_4) 938(P_3) 860(P_1) 782(P_3) 704(P_1) 626(P_3) 548(P_4) 470(P_2) 392(P_1) 148(P_3)	1250(P_1) 1172(P_3) 446(P_2) 433(P_4) 1027(P_3) 388(P_4) 375(P_2) 882(P_1) 804(P_3) 299(P_4) 286(P_2) 660(P_1) 582(P_3) 504(P_1) 179(P_4) 392(P_3) 134(P_2) 187(P_1)	89.189s	63.974s

The performance gain of $\mathscr{W}$-$\mathscr{A}$ over $\mathscr{A}$ is quite significant

The Combined $\mathscr{S}\mathscr{W}$ Mechanisms



- $\mathscr{S}\mathscr{W}$ enable self-scheduling algorithms to handle DOACROSS loops on heterogeneous systems with load variations
- Synchronization points are introduced and chunks are weighted

Observations:

1. Since $\mathscr{S}$ does not provide any load balancing, it is most advantageous to use $\mathscr{W}$ to achieve it
2. The synchronization & weighting overheads are compensated by the performance gain

The Combined $\mathscr{SW}$ Mechanisms

Master

While there are unassigned chunks
{
1. Receive request from P_k
2. Calculate C_i according to $\mathscr{A}$
3. Serve request
}

$\mathscr{SW}$-Master

While there are unassigned chunks
{
1. Receive request from P_k
2. Calculate C_i according to $\mathscr{A}$

3. Apply $\mathscr{W}$ to compute $\hat{C}_i$
4. Make P_k - current slave
5. Make P_{k-1} - previous slave
6. Send P_{k-1} the rank of P_k
7. Send P_k the rank of P_{k-1}

8. Serve request
}

Slave P_k

1. Make new request to Master
2. If request served
{
Compute chunk
}
3. Go to step 1

$\mathscr{SW}$-Slave P_k

1. Make new request to Master
2. Report current load Q_k
3. If request served
{
Receive partial results from P_{k-1}
Compute chunk
Send partial results to P_{k+1}
}
4. Go to step 1

$\mathscr{SW}$ add 5 (3+2) components to the original algorithm $\mathscr{A}$:

1. chunk weighting (master)
2. transaction accounting (master)
3. run-queue monitoring (slave)
4. receive part (slave)
5. transmit part (slave)

Evaluation of the $\mathscr{SW}$ Mechanisms

Table: Chunk sizes given by the synchronized–only and synchronized–weighted algorithms for the Floyd-Steinberg DOACROSS loop, index space size $|J| = 10000 \times 10000$ points and $m = 4$, $VP_1 = VP_3 = 1$, $VP_2 = VP_4 = 0.8$ and P_2 and P_4 were loaded

$\mathscr{A}$	Chunk sizes with $\mathscr{S}$-$\mathscr{A}$ with respect to the processors' request order	Chunk sizes with $\mathscr{SW}$-$\mathscr{A}$ with respect to the processors' request order	Par. time for $\mathscr{S}$-$\mathscr{A}$	Par. time for $\mathscr{SW}$-$\mathscr{A}$
CSS	1250(P_1) 1250(P_3) 1250(P_2) 1250(P_4) 1250(P_1) 1250(P_3) 1250(P_2) 1250(P_4)	1250(P_1) 1250(P_3) 500(P_2) 500(P_4) 1250(P_1) 1250(P_3) 500(P_2) 500(P_4) 1250(P_1) 1250(P_3) 500(P_2)	27.335s	16.582s
FSS	1250(P_1) 1250(P_3) 1250(P_2) 1250(P_4) 625(P_1) 625(P_3) 625(P_2) 625(P_4) 390(P_1) 390(P_3) 390(P_2) 390(P_4) 244(P_1) 244(P_3) 244(P_2) 208(P_4)	1250(P_1) 1250(P_3) 500(P_2) 500(P_4) 812(P_1) 812(P_3) 324(P_2) 324(P_4) 630(P_1) 630(P_3) 252(P_2) 252(P_4) 488(P_1) 488(P_3) 195(P_2) 195(P_4) 378(P_1) 378(P_3) 151(P_2) 151(P_4) 40(P_1)	27.667s	16.556s

The slaves' request order is the same due to existing dependencies and synchronizations

Evaluation of the $\mathscr{S}\mathscr{W}$ Mechanisms

Table: Chunk sizes given by the synchronized–only and synchronized–weighted algorithms for the Floyd-Steinberg DOACROSS loop, index space size $|J| = 10000 \times 10000$ points and $m = 4$

$\mathscr{A}$	Chunk sizes with $\mathscr{S}$-$\mathscr{A}$ with respect to the processors' request order	Chunk sizes with $\mathscr{S}\mathscr{W}$-$\mathscr{A}$ with respect to the processors' request order	Par. time for $\mathscr{S}$-$\mathscr{A}$	Par. time for $\mathscr{S}\mathscr{W}$-$\mathscr{A}$
GSS	2500(P_1) 1875(P_3) 1406(P_2) 1054(P_4) 791(P_1) 593(P_3) 445(P_2) 334(P_4) 250(P_1) 188(P_3) 141(P_2) 105(P_4) 80(P_1) 80(P_3) 80(P_2) 78(P_4)	2500(P_1) 1875(P_3) 562(P_2) 506(P_4) 1139(P_1) 854(P_3) 256(P_2) 230(P_4) 519(P_1) 389(P_3) 116(P_2) 105(P_4) 237(P_1) 178(P_3) 80(P_2) 80(P_4) 108(P_1) 81(P_3) 80(P_2) 80(P_4) 25(P_1)	28.526s	18.569s
TSS	1250(P_1) 1172(P_2) 1094(P_3) 1016(P_4) 938(P_1) 860(P_2) 782(P_3) 704(P_4) 626(P_1) 548(P_2) 470(P_3) 392(P_4) 148(P_1)	509(P_2) 1217(P_1) 464(P_4) 1105(P_3) 420(P_2) 995(P_1) 376(P_4) 885(P_3) 332(P_2) 775(P_1) 288(P_4) 665(P_3) 244(P_2) 555(P_1) 200(P_4) 445(P_3) 156(P_2) 335(P_1) 34(P_4)	25.587s	14.309s

The performance gain of $\mathscr{S}$-$\mathscr{A}$ over $\mathscr{S}\mathscr{W}$-$\mathscr{A}$ is quite significant

Results

Experimental Setup

- The algorithms are implemented in C and C++
- MPI is used for master-slave and inter-slave communication
- The heterogeneous system consists of 13 nodes (1 master and 12 slaves):
  - 7 twins: Intel Pentiums III, 800 MHz with 256MB RAM, assumed to have $VP_k = 1$ (one of them is the master)
  - 6 kids: Intel Pentiums III, 500 MHz with 512MB RAM , assumed to have $VP_k = 0.8$
- Interconnection network is Fast Ethernet, at 100Mbit/sec
- Non-dedicated system: at the beginning of program's execution, a resource expensive process is started on some of the slaves, halving their A_k
- Machinefile: twin1 (master),twin2, kid1, twin3, kid2, twin4, kid3, twin5, kid4, twin6, kid5, twin7, kid6
- In all cases, the kids were overloaded

Results
Experimental Setup

- Three series of experiments on the non-dedicated system, for m = 4,6,8,10,12 slaves:
  1) for the synchronization mechanism $\mathscr{S}$
  2) for the weighting mechanism $\mathscr{W}$
  3) for the combined mechanisms $\mathscr{S}\mathscr{W}$
- Two real-life applications: Floyd-Steinberg (regular DOACROSS), and Mandelbrot (irregular DOALL)
  (*Similar results for Hydro – in thesis*)
- Reported results are averages of 10 runs for each case
- The chunk size for CSS was: $C_i = \frac{U_c}{2 \times m}$
- The number of synchronization points was: $M = 3 \times m$
- Lower and upper thresholds for the chunk sizes (table below)
- 3 problem sizes - some analyzed here, some in thesis

Problem size	**small**	**medium**	**large**
Floyd-Steinberg	5000×15000	10000×15000	15000×15000
upper/lower threshold	500/10	750/10	1000/10
Mandelbrot	7500×10000	10000×10000	12500×12500

Experiment 1

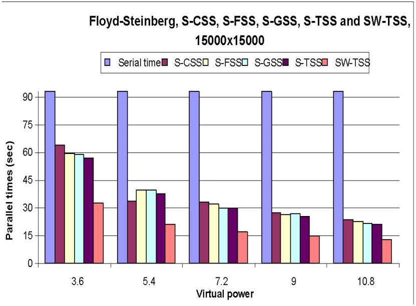
Speedups of the synchronized–only algorithms for Floyd-Steinberg

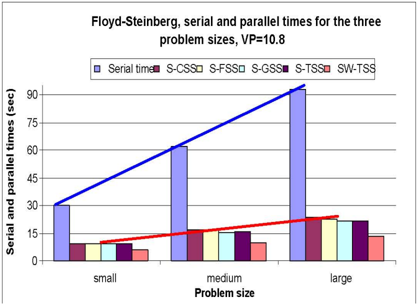
Test case	VP	$\mathscr{S}$-CSS	$\mathscr{S}$-FSS	$\mathscr{S}$-GSS	$\mathscr{S}$-TSS	$\mathscr{SW}$-TSS
Floyd-Steinberg	**3.6**	1.45	1.57	1.59	1.63	2.86
	5.4	2.76	2.35	2.33	2.47	4.35
	7.2	2.81	2.92	3.09	3.10	5.39
	9	3.41	3.50	3.49	3.70	6.27
	10.8	3.95	4.07	4.27	4.34	7.09

- The serial time was measured on the fastest slave type, i.e., twin
- $\mathscr{S}$-CSS, $\mathscr{S}$-FSS, $\mathscr{S}$-GSS and $\mathscr{S}$-TSS give significant speedups
- $\mathscr{SW}$-TSS gives an even greater speedup over all synchronized–only algorithms ☺ expected!

Experiment 1

Parallel times of the synchronized–only algorithms for Floyd-Steinberg





Serial times increase faster than parallel times as the problem size increases ⇒ larger speedups for larger problems ☺ anticipated!

Experiment 2

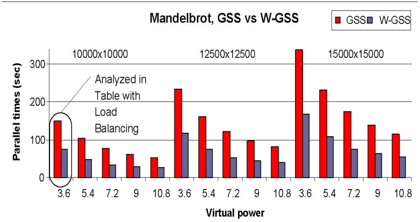
Gain of the weighted over non-weighted algorithms for Mandelbrot

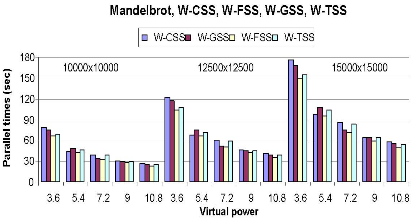
Test case	Problem size (large)	**VP**	CSS vs $\mathscr{W}$-CSS	GSS vs $\mathscr{W}$-GSS	FSS vs $\mathscr{W}$-FSS	TSS vs $\mathscr{W}$-TSS
Mandelbrot	15000×15000	**3.6**	27%	50%	18%	33%
		5.4	38%	54%	37%	34%
		7.2	45%	57%	53%	31%
		9	49%	54%	52%	35%
		10.8	46%	52%	54%	33%
Confidence interval (95%)	Overall 42 ± 3 %		40 ± 6 %	53 ± 6 %	42 ± 8 %	33 ± 4 %

- Gain is computed as $\frac{T_{\mathscr{A}} - T_{\mathscr{W}-\mathscr{A}}}{T_{\mathscr{A}}}$
- GSS has the best overall performance gain

Experiment 2

Parallel times of the weighted algorithms for Mandelbrot





The performance difference of the weighted algorithms is *much smaller* than that of their non-weighted versions ☺ anticipated!

Experiment 2

Load balancing obtained with $\mathscr{W}$ for Mandelbrot

Table: Load balancing in terms of total number of iterations per slave and computation times per slave, GSS vs $\mathscr{W}$-GSS.

Slave	GSS # Iterations (10^6)	GSS Comp. time (sec)	$\mathscr{W}$-GSS # Iterations (10^6)	$\mathscr{W}$-GSS Comp. time (sec)
twin2	56.434	34.63	55.494	62.54
kid1	18.738	138.40	15.528	62.12
twin3	10.528	39.37	15.178	74.63
kid2	14.048	150.23	13.448	61.92

$\mathscr{W}$-GSS achieves better load balancing and smaller parallel time

Experiment 3

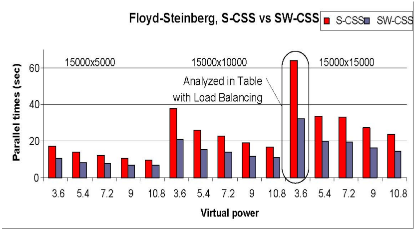
Gain of the synchronized–weighted over synchronized–only algorithms for Floyd-Steinberg

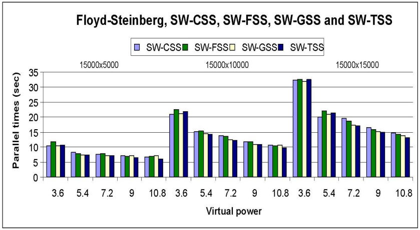
Test case	Problem size	**VP**	$\mathscr{S}$-CSS vs $\mathscr{SW}$-CSS	$\mathscr{S}$-GSS vs $\mathscr{SW}$-GSS	$\mathscr{S}$-FSS vs $\mathscr{SW}$-FSS	$\mathscr{S}$-TSS vs $\mathscr{SW}$-TSS
Floyd-Steinberg	15000×10000	**3.6**	**50%**	46%	45%	43%
		5.4	41%	48%	44%	43%
		7.2	41%	42%	41%	42%
		9	39%	43%	40%	41%
		10.8	38%	36%	38%	39%
Confidence interval (95%)	Overall 40 ± 1 %		39 ± 2 %	40 ± 3 %	40 ± 2 %	41 ± 2 %

- Gain is computed as $\frac{T_{\mathscr{S}-\mathscr{A}} - T_{\mathscr{SW}-\mathscr{A}}}{T_{\mathscr{S}-\mathscr{A}}}$
- CSS has the highest performance gain 50%

Experiment 3

Parallel times of the synchronized–weighted and synchronized–only algorithms for Floyd-Steinberg





The performance difference of the synchronized–weighted algorithms is *much smaller* than that of their synchronized–only versions
☺ anticipated!

Experiment 3

Load balancing obtained with $\mathscr{SW}$ for Floyd-Steinberg

Table: Load balancing in terms of total number of iterations per slave and computation times per slave, $\mathscr{S}$-CSS vs $\mathscr{SW}$-CSS

Test	Slave	# Iterations (10^6)	Comp. time (sec)	# Iterations (10^6)	Comp. time (sec)
		$\mathscr{S}$-CSS	$\mathscr{S}$-CSS	$\mathscr{SW}$-CSS	$\mathscr{SW}$-CSS
Floyd-Steinberg	twin2	59.93	19.25	89.90	28.88
	kid1	59.93	62.22	29.92	30.86
	twin3	59.93	19.24	74.92	24.06
	kid2	44.95	46.30	29.92	29.08

$\mathscr{SW}$-CSS achieves better load balancing and smaller parallel time than its synchronized–only counterpart ☺ anticipated!

Conclusions

- DOACROSS loops can be dynamically scheduled using $\mathscr{S}$
- Self-scheduling algorithms are quite efficient on heterogeneous dedicated & non-dedicated systems using $\mathscr{W}$
- $\mathscr{S}\mathscr{W}$ Self-scheduling algorithms are even more efficient on heterogeneous dedicated & non-dedicated systems

Outline

Summary

- If the target platform is well identified and stable, strive to
  1. accurately model the hierarchical structure, and
  2. design well suited scheduling algorithms
- If the target platform is not stable enough or if it evolves too fast, then
  - dynamic schedulers are the ONLY option
- But, to reduce the scheduling overhead
  - inject static knowledge into the dynamic schedulers
- If the target platform is a distributed system
  - coarse grain methods are required to schedule tasks on distributed and heterogeneous systems

☺ DOACROSS loops can now be dynamically scheduled on heterogeneous dedicated & non-dedicated systems using $\mathscr{S}$

☺ $\mathscr{SW}$ self-scheduling algorithms are even more efficient on heterogeneous dedicated & non-dedicated systems

Future Work

1. Implementing the dynamic scheduling and load balancing algorithms presented here on Grid computing environments
2. Design a fault tolerant mechanism for the scheduling DOACROSS loops to increase system reliability and maximize resource utilization in distributed systems
3. Employ the scheduling algorithms presented earlier to perform large scale computation (containing both DOALL and DOACROSS loops) on computational grids
4. Use the scheduling algorithms presented earlier to schedule and load balance divisible loads (i.e. loads that can be modularly divided into precedence constrained loads)

Publications Resulted From This Work

Accepted journal publications:

J.1 T. Andronikos, F. M. Ciorba, P. Theodoropoulos, D. Kamenopoulos and G. Papakonstantinou, ”Cronus: A platform for parallel code generation based on computational geometry methods”, *Journal of Systems and Software*, 2008.

J.2 F. M. Ciorba, I. Riakiotakis, T. Andronikos, G. Papakonstantinou and A. T. Chronopoulos, ”Enhancing self-scheduling algorithms via synchronization and weighting”, *Journal of Parallel and Distributed Computing*, 2008.

J.3 G. Papakonstantinou, I. Riakiotakis, T. Andronikos, F. M. Ciorba and A. T. Chronopoulos, ”Dynamic Scheduling for Dependence Loops on Heterogeneous Clusters”, *Neural, Parallel & Scientific Computations*, 2006.

J.4 F. M. Ciorba, T. Andronikos and G. Papakonstantinou, ”Adaptive Cyclic Scheduling of Nested Loops”, *HERMIS International Journal*, 2006.

Journal publications under review:

UR.1 I. Riakiotakis, F. M. Ciorba, T. Andronikos, G. Papakonstantinou and A. T. Chronopoulos, ”Optimal Synchronization Frequency for Dynamic Pipelined Computations on Heterogeneous Systems”, *Journal of Cluster Computing*.

UR.2 F. M. Ciorba, I. Riakiotakis, T. Andronikos, G. Papakonstantinou and A. T. Chronopoulos, ”Studying the impact of synchronization frequency on scheduling tasks with dependencies in heterogeneous systems”, *Journal of Performance Evaluation*.

Publications Resulted From This Work

Accepted conference publications:

C.1 F. M. Ciorba, I. Riakiotakis, T. Andronikos, A. T. Chronopoulos, and G. Papakonstantinou, “Optimal Synchronization Frequency for Dynamic Pipelined Computations on Heterogeneous Systems”, *IEEE CLUSTER 2007*, 2007.

C.2 F. M. Ciorba, I. Riakiotakis, T. Andronikos, A. T. Chronopoulos, and G. Papakonstantinou, “Studying the impact of synchronization frequency on scheduling tasks with dependencies in heterogeneous systems”, *IEEE & ACM PACT ’07*, 2007.

C.3 I. Riakiotakis, F. M. Ciorba, T. Andronikos, and G. Papakonstantinou, “Self-Adapting Scheduling for Tasks with Dependencies in Stochastic Environments”, *IEEE CLUSTER 2006, HeteroPar ’06 Workshop*, 2006.

C.4 F. M. Ciorba, T. Andronikos, I. Riakiotakis, A. T. Chronopoulos, and G. Papakonstantinou, “Dynamic Multi Phase Scheduling for Heterogeneous Clusters”, *IEEE IPDPS’06*, 2006.

C.5 F. M. Ciorba, T. Andronikos, I. Drositis, G. Papakonstantinou, “Reducing Communication via Chain Pattern Scheduling”, *IEEE NCA’05*, 2005.

C.6 F. M. Ciorba, T. Andronikos, G. Papakonstantinou, “Adaptive Cyclic Scheduling of Nested Loops”, *HERCMA’05*, 2005.

C.7 T. Andronikos, F. M. Ciorba, P. Theodoropoulos, D. Kamenopoulos and G. Papakonstantinou, “Code Generation for General Loops Using Methods from Computational Geometry”, *IASTED PCDS 2004*, 2004.

C.8 F. M. Ciorba, T. Andronikos, D. Kamenopoulos, P. Theodoropoulos and G. Papakonstantinou, “Simple Code Generation for Special UDLs”, *BCI’03*, 2003.

C.9 T. Andronikos, M. Kalathas, F. M. Ciorba, P. Theodoropoulos and G. Papakonstantinou, “An Efficient Scheduling of Uniform Dependence Loops”, *HERCMA’03*, 2003.

C.10 T. Andronikos, M. Kalathas, F. M. Ciorba, P. Theodoropoulos, G. Papakonstantinou and P. Tsanakas, “Scheduling nested loops with the least number of processors”, *IASTED AI 2003*, 2003.

Thank you for your attention!

Questions?

Mandelbrot

```
for (hy=1; hy<=hyres; hy++)  { /* scheduling dimension */
    for (hx=1; hx<=hxres; hx++)  {
        cx = (((float)hx)/((float)hxres)-0.5)/magnify*3.0-0.7;
        cy = (((float)hy)/((float)hyres)-0.5)/magnify*3.0;
        x = 0.0; y = 0.0;
        for (iteration=1; iteration<itermax; iteration++)  {
            xx = x*x-y*y+cx;
            y = 2.0*x*y+cy;
            x = xx;
            if (x*x+y*y>100.0)  iteration = 999999;
        }
        if (iteration<99999)  color(0,255,255);
        else color(180,0,0);
    }
}
```

Heat Diffusion Equation

```
for (l=1; l<loop; l++) {
    for (i=1; i<width; i++){  /* synchronization dimension */
        for (j=1; j<height; j++){ /* scheduling dimension */
            A[i][j] = 1/4*(A[i-1][j] + A[i][j-1]
                   + A'[i+1][j] + A'[i][j+1]);
        }
    }
}
```

Floyd-Steinberg Error Dithering

```
for (i=1; i<width; i++){ /* synchronization dimension */
    for (j=1; j<height; j++){ /* scheduling dimension */
            I[i][j] = trunc(J[i][j]) + 0.5;
            err = J[i][j] - I[i][j]*255;
            J[i-1][j] += err*(7/16);
            J[i-1][j-1] += err*(3/16);
            J[i][j-1] += err*(5/16);
            J[i-1][j+1] += err*(1/16);
    }
}
```

Modified LL23 - Hydrodynamics kernel

```
for (l=1; l<=loop; l++) { /* synchronization dimension */
    for (j=1; j<5; j++)  {
        for (k=1; k<n; k++){ /* chunk dimension */
            qa = za[l-1][j+1][k]*zr[j][k] + za[l][j-1][k]*zb[j][k] +
                 za[l-1][j][k+1]*zu[j][k] + za[l][j][k-1]*zv[j][k] +
                 zz[j][k];
            za[l][j][k] += 0.175 * (qa - za[l][j][k] );
        }
    }
}
```

Bibliography

Lamport, L.
The Parallel Execution of DO Loops.
Comm. of the ACM, 37(2):83–93, 1974.

Moldovan, D. I. and Fortes, J.
Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays.
IEEE Transactions on Computers, C-35(1):1–11, 1986.

Shang, W. and Fortes, J.A.B.
Time Optimal Linear Schedules for Algorithms with Uniform Dependencies.
IEEE Transactions on Computers, 40(6):723–742, 1991.

Bibliography (cont.)

Darte, A. and Khachiyan, L. and Robert, Y.
Linear Scheduling is Nearly Optimal.
Par. Proc. Letters, 1(2):73–81, 1991.

Koziris, N. and Papakonstantinou, G. and Tsanakas, P.
Optimal time and efficient space free scheduling for nested loops.
The Computer Journal, 39(5):439–448, 1996.

Varvarigou, T. and Roychowdhury, V. P. and Kailath, T. and Lawler, E.
Scheduling In and Out Forrests in the Presence of Communication Delays.
IEEE Trans. of Par. and Dist. Comp., 7(10):1065–1074, 1996.

Bibliography (cont.)

Jung, H. and Kirousis, L. and Spirakis, P.
Lower Bounds and Efficient Algorithms for Multiprocessor Sceduling of DAGs with Communication Delays.
1st ACM SPAA, 1989.

Chretienne, P.
Task Scheduling with Interprocessor Communication Delays.
European Journal of Operational Research, 57:348–354, 1992.

Andronikos, T. and Koziris, N. and Tsiatsoulis, Z. and Papakonstantinou, G. and Tsanakas, P.
Lower Time and Processor Bounds for Efficient Mapping of Uniform Dependence Algorithms into Systolic Arrays.
Journal of Parallel Algorithms and Applications, 10(3-4):177–194, 1997.

Bibliography (cont.)

Papakonstantinou, G. and Andronikos, T. and Drositis, I.
On the Parallelization of UET/UET-UCT Loops.
Neural, Parallel & Scientific Computations, 2001.

King, C.-T., Chou, W.-H., and Ni, L.
Pipelined Data-Parallel Algorithms: Part II Design.
IEEE Trans. on Parallel and Distributed Systems, 2(4):430–439, 1991.

J.-P. Sheu and T.-S. Chen
Partitioning and Mapping Nested Loops for Linear Array Multicomputers.
Journal of Supercomputing, 9:183–202, 1995.

Bibliography (cont.)

Tsanakas, P. and Koziris, N. and Papakonstantinou, G.
Chain Grouping: A Method for Partitioning Loops onto Mesh-Connected Processor Arrays.
IEEE Transactions on Parallel and Distributed Systems, 11(9):941–955, 2000.

Drositis, I. and Goumas, G. and Koziris, N. and Tsanakas, P. and Papakonstantinou
Evaluation of Loop Grouping Methods based on Orthogonal Projection Spaces.
Proc. Int. Conf. on Par. Proc. ICPP-2000, 469–476, 2000.

Bibliography (cont.)

Papadimitriou, C. and Yannakakis, M.
Toward an Architecture-Independent Analysis of Parallel Algorithms.
SIAM J. of Comp., Ext. Abstract in Proc. STOC 1988, 19:322–328, 1988.

Wolf, M. E. and Lam, M. S.
A data locality optimizing algorithm.
PLDI '91: Proc. of the ACM SIGPLAN 1991 Conf. on Progr. Lang. Design and Impl., 30-44, 1991.

J. Ramanujam and P. Sadayappan
Tiling Multidimensional Iteration Spaces for Multicomputers.
Journal of Parallel and Distributed Computing, 16(2):108–230, 1992.

Bibliography (cont.)

Boulet, P. and Darte, A. and Risset, T. and Robert, Y.
(Pen)-ultimate tiling.
Integration, the VLSI Journal, 17:33–51, 1994.

Jingling Xue
On Tiling as a Loop Transformation.
Parallel Processing Letters, 7(4):409-424, 1997.

Goumas, G. and Drosinos, N. and Athanasaki, M. and Koziris, N.
Message-Passing Code Generation for Non-rectangular Tiling Transformations
Parallel Computing, 32(11), 2006.

Bibliography (cont.)

Andronikos, T. and Kalathas, M. and Ciorba, F. M. and Theodoropoulos, P. and Papakonstantinou, G.
An Efficient Scheduling of Uniform Dependence Loops.
6th Hellenic European Research on Computer Mathematics and its Applications (HERCMA'03), 2003.

Andronikos, T. and Kalathas, M. and Ciorba, F.M. and Theodoropoulos, P. and Papakonstantinou, G.
Scheduling nested loops with the least number of processors.
Proc. of the 21st IASTED Int'l Conference APPLIED INFORMATICS, 2003.

Ciorba, F. M. and Andronikos, T. and Papakonstantinou, G.
Adaptive Cyclic Scheduling of Nested Loops.
7th Hellenic European Research on Computer Mathematics and its Applications (HERCMA'05), 2005.

Bibliography (cont.)

Ciorba, F. M. and Andronikos, T. and Drositis, I. and Papakonstantinou, G. and Tsanakas, P.
Reducing the Communication Cost via Chain Pattern Scheduling.
4th IEEE Conference on Network Computing and Applications (NCA'05), 2005.

Ciorba, F. M. and Andronikos, T. and Kamenopoulos, D. and Theodoropoulos, P. and Papakonstantinou, G.
Simple code generation for special UDLs.
1st Balkan Conference in Informatics (BCI'03), 2003.

Bibliography (cont.)

Andronikos, T. and Ciorba, F. M. and Theodoropoulos, P. and Kamenopoulos D. and Papakonstantinou, G.
Code Generation For General Loops Using Methods From Computational Geometry.
IASTED Parallel and Distributed Computing and Systems Conference (PDCS'04), 2004.

Ciorba, F. M. and Andronikos, T. and Riakiotakis, I. and Chronopoulos, A. T. and Papakonstantinou, G.
Dynamic Multi Phase Scheduling for Heterogeneous Clusters.
20th International Parallel and Distributed Processing Symposium (IPDPS'06), 2006.

Bibliography (cont.)

Papakonstantinou, G. and Riakiotakis, I. and Andronikos, T. and Ciorba, F. M. and Chronopoulos, A. T.
Dynamic Scheduling for Dependence Loops on Heterogeneous Clusters.
Neural, Parallel & Scientific Computations, 14(4):359–384, 2006.

Ciorba, F. M. and Riakiotakis, I. and Andronikos, T. and Papakonstantinou, G. and Chronopoulos, A. T.
Enhancing self-scheduling algorithms via synchronization and weighting.
J. Parallel Distrib. Comput., 68(2):246–264, 2006.

Bibliography (cont.)

Ciorba, F. M. and Riakiotakis, I. and Andronikos, T. and Chronopoulos, A. T. and Papakonstantinou, G.
Studying the Impact of Synchronization Frequency on Scheduling Tasks with Dependencies in Heterogeneous Sytems.
16th International Conference on Parallel Architectures and Compilations Techniques (PACT '07), 2007.

C.P. Kruskal and A. Weiss
Allocating independent subtasks on parallel processors.
IEEE Trans. on Software Eng., 11(10):1001–1016, 1985.

T.H. Tzen and L.M. Ni.
Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers.
IEEE Trans. on Par. and Dist. Sys., 4(1):87–98, 1993.

Bibliography (cont.)

- A.T. Chronopoulos, R. Andonie, M. Benche and D. Grosu.
  A Class of Distributed Self-Scheduling Schemes for Heterogeneous Clusters.
  3rd IEEE Int. Conf. on Cluster Computing, 2001.
- S.F. Hummel, E. Schonberg and L.E. Flynn.
  Factoring: A Method for Scheduling Parallel Loops
  Comm. of the ACM, 35(8):90–101, 1992.
- C.D. Polychronopoulos and D.J. Kuck.
  Guided self-scheduling: A practical self-scheduling scheme for parallel supercomputers
  IEEE Trans. on Computer, C-36(12): 1425–1439, 1987.