Параллельное программирование

Технологии параллельного программирования

Средства параллельного программирование

- Более 100 средств для решения задач на параллельном компьютере
- Всегда стоит вопрос эффективности
- Модели программирования заточены под конкретную системы
- Выбор системы зависит от задачи

Популярные системы

- Мультипроцессоры
  - OpenMP(Open Multi-Processing)
- Мультикомпьютеры
  - MPI(Message Passing Interface)
- Гибридные архитектуры
  - OpenMP + MPI
  - OpenMP
  - MPI

Другие варианты

- Threads (C++11, posix, boost)
- Cilc
- Intel Threding Buildings Block
- Java fork-Join Framework
- .Net Task Parallel
- Parallel Patterns Library
- PVM
- MapReduce

Подходы

- Расширение последовательного языка
  - OpenMP
  - Cilk
  - UPC
- Библиотека к последоватльному языку
  - PThreads
  - MPI
  - TBB
- Языки со встроенной поддержкой многозадачности
  - C++11, C#, Java, Erlang

Критерии оценки программы

- Ясность
- Масштабируемость
- Эффективность
- Удобство сопровождения
- Согласованность с целtвой системой
- Переносимост
- Эквивалентность последовательной программе

Парадигмы параллельного программирования

- SPMD(Single Program Multiple Data)
- Loop Parallelism
- Master\Worker
- Fork\Join

- Тесно связаны
- Допускается комбинирование

Single Program Multiple Data

- Наиболее распространенный подход
- Все исполнители запускаю одну программу
- Каждый исполнитель имеет уникальных идентификатор
- Исполнители получают разные данные (в зависимости от иденфикатора)
- Редукция результатов

Пример. Интеграл

```
double liner(double x){
    return x;
}

int _tmain(int argc, _TCHAR* argv[])
{
    time_t start_time;
    time_t finish_time;

    int eteration_count = 10000000000000000; // количество итераций
    int a = 0;                          // левая граница интегрирования
    int b = 1;                          // правая граница интегрирования
    int x;

    time(&start_time);

    double step = (b - a) / eteration_count;
    double total = (liner(a) + liner(b))*step / 2;
    for (int i = 0; i < eteration_count; ++i)
    {
        x = a + i* step;
        total += liner(x)*step;
    }
    time(&finish_time);

    std::cout << "Result is: " << total << " Operation time: " << difftime(finish_time, start_time) << std::endl;
    return 0;
}
```

Пример. MPI

```
int _tmain(int argc, _TCHAR* argv[])
{
	time_t start_time;
	time_t finish_time;

	int eteration_count = 10000000000000000; // количество итераций
	int a = 0;                                          // левая граница интегрирования
	int b = 1;                                          // правая граница интегрирования
	int x;

	int i_start, i_end;

	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
	MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

	i_start = my_id * (num_steps / numprocs);
	i_end = i_start + (num_steps / numprocs);

	double total = 0;

	if (my_id == (numprocs - 1))
		i_end = num_steps;

	for (i = i_start; i< i_end; i++)
	{
		x = a + i* step;
		total += liner(x)*step;
	}


	MPI_Reduce(&sum, &total, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
	total+=(liner(a) + liner(b))*step / 2;

	if (my_id == 0)
		printf("Result = %.10 f\n", total);

	MPI_Finalize();

	return
}
```

Single Program Multiple Data

- Плюсы
  - Накладные расходы изолированы в начале и конце
  - Поддержка сложно координации между исполнителями
  - Возможность использовать на MIMD
- Минусы
  - Программа сильно отличается от последовательной
  - Сложная логика распределения данных и загрузки
  - Плохо подходит на динамической балансировки нагрузки

Loop Parallelism

- Вычислительно-интенсивная часть скрыта внутри алгоритма
- Ест готовая последовательная программа
- Итерации внутри цикла независимы
- Требуется распараллелить итерации, минимально модифицируя код исходной программы

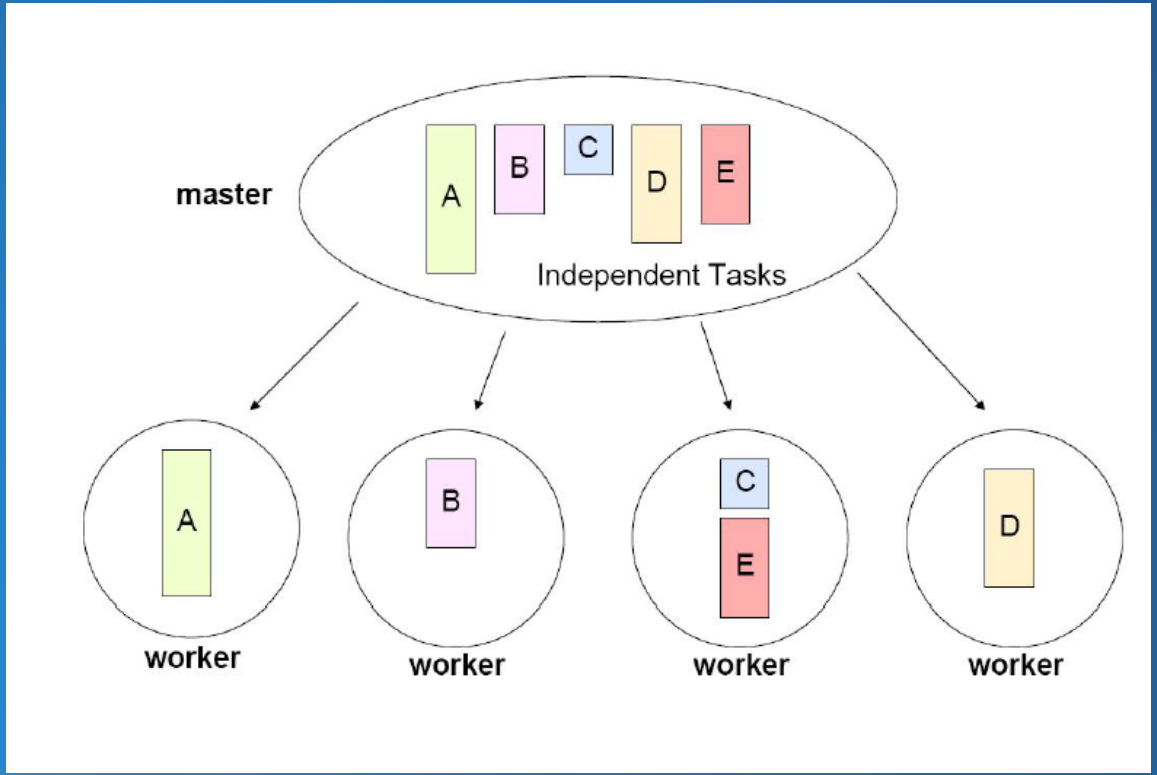
Пример (OpenMP)

```
double liner(double x){
    return x;
}

int _tmain(int argc, _TCHAR* argv[])
{
    time_t start_time;
    time_t finish_time;

    int eteration_count = 10000000000000000; // количество итераций
    int a = 0;                          // левая граница интегрирования
    int b = 1;                          // правая граница интегрирования

    time(&start_time);

    double total = 0;
    double x;
    double step = (float)(b - a) / eteration_count;

    total = (liner(a) + liner(b))*step / 2;
    #pragma omp parallel
    #pragma omp for private(x) reduction(+:total)
    for (int i = 0; i < eteration_count; ++i){
        x = a + i*step;
        total += liner(x)*step;
    }

    time(&finish_time);
    std::cout << "Result is: " << total << " Operation time: " << difftime(finish_time, start_time) << std::endl;
    return 0;
}
```

Loop Parallelism

- Плюсы
  - Минимальные модификации последовательной программы
  - Инкрементальное распараллеливание
  - Эквивалентность последовательной программе
- Минусы
  - Ориентация на системы с общей памятью(SMP)
  - Оптимизация доступа к памяти может потребовать рефакторинга
  - Ограничение масштабируемости.

Master/Worker



Master/Worker

- Требуется динамическая балансировка нагрузки между группой исполнителей
  - Сложность заданий меняется сильно и непредсказуемо
  - Вычисления не сводятся к простым циклам
  - Исполнителя могут отличаться друг от друга

Master/Worker

- Плюсы
  - Автоматических балансировка нагрузки
  - Хорошо работает для независимых задач
  - Подходи для разных платформ
- Минусы
  - Не подходит для зависимых задач

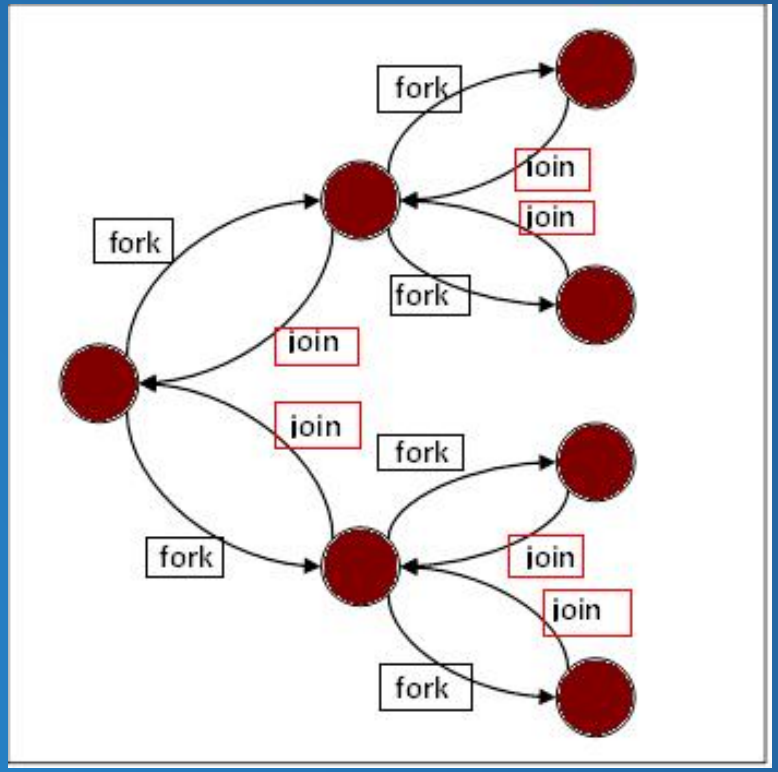
Рекурсивный Fork/Join

Разделяй и властвуй

Походы

- Каждому заданию по исполнителю
- Пул исполнителей
  - Очередь заданий
  - Work stealing

Рекурсивный Fork/Join



Рекурсивный Fork/Join

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
        int x, y;

        x = spawn fib (n-1);
        y = spawn fib (n-2);

        sync;

        return (x+y);
    }
}
```